

## ❖ 2. JavaScript Fundamentals – Part 1 → 5. Values and Variables

Using camelCase to declare variables

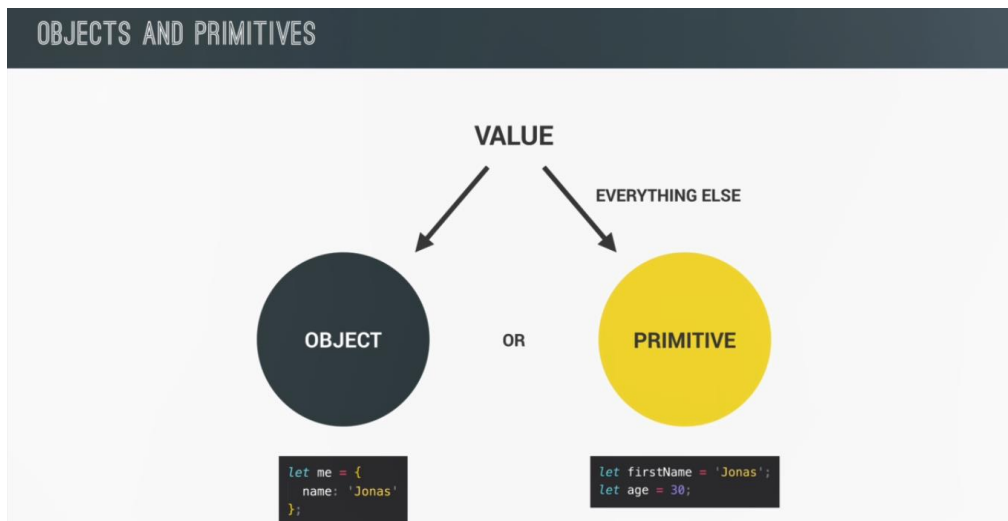
Here are rules JavaScript has for naming variables:

- **Variable** names cannot contain spaces.
- **Variable** names must begin with a letter, an underscore (`_`) or a dollar sign (`$`).
- **Variable** names can only contain letters, numbers, underscores, or dollar signs.
- **Variable** names are case-sensitive.
- **Don't use names that are too short.** Simple one-letter names or names that don't make sense are not a good option when naming variables.
- **Use more than one word to name your variable.** This will ensure your variable name is precise.
- **When using more than one word in your variable names, always put the adjective to the left.** For example, this is correct: `var greenGrass`.
- **Pick a style for names with more than one word, and stick to it.** The two most common ways to join words to create a name are camelCase and using an underscore (`_`). JavaScript is flexible — either method works.

### JavaScript Reserved Words

abstract	arguments	await*	boolean
break	byte	case	catch
char	class*	const	continue
debugger	default	delete	do
double	else	enum*	eval
export*	extends*	false	final
finally	float	for	function
goto	if	implements	import*
in	instanceof	int	interface
let*	long	native	new
null	package	private	protected
public	return	short	static
super*	switch	synchronized	this
throw	throws	transient	true
try	typeof	var	void
volatile	while	with	yield

## ❖ 2. JavaScript Fundamentals – Part 1 → 7. Data Types



### Primitive data types :

## THE 7 PRIMITIVE DATA TYPES

1. **Number:** Floating point numbers 🖱 Used for decimals and integers `let age = 23;`
2. **String:** Sequence of characters 🖱 Used for text `let firstName = 'Jonas';`
3. **Boolean:** Logical type that can only be true or false 🖱 Used for taking decisions `let fullAge = true;`
4. **Undefined:** Value taken by a variable that is not yet defined ('empty value') `let children;`
5. **Null:** Also means 'empty value'
6. **Symbol (ES2015):** Value that is unique and cannot be changed *[Not useful for now]*
7. **BigInt (ES2020):** Larger integers than the Number type can hold

👉 **JavaScript has dynamic typing:** We do *not* have to manually define the data type of the value stored in a variable. Instead, data types are determined **automatically**.

## ❖ 2. JavaScript Fundamentals – Part 1 → 8. let, const and var

sn	Var	let	Const
01.	We have before ES6 and now also	It becomes in ES6 (ES2015)	It becomes in ES6 (ES2015)
02.	It can be redeclared	It <b>can't</b> be redeclared	It can't be redeclared
03.	It has Global Scope and function Scope or local Scope, so it can be accesse inside the function if it is defined inside the function. And it can be accessible outside of a functin and inside of a function also, if it is defined outside of a function	It has Block Scope, so only accessible inside that block where it is defined	It has Block Scope, so only accessible inside that block where it is defined

	<a href="#">var</a>	<a href="#">let</a>	<a href="#">const</a>
Redeclarable	✓	✗	✗
Reassignable	✓	✓	✗
Global property	✓	✗	✗
Block scope	✗	✓	✓
Hoisted	✓	✗	✗

### Basic Operators

- += \*= \*\* -- ++ .....
- Operator Precedence

Operator	Operation	Order of Precedence	Order of Evaluation
++	Increment	1	R -> L
--	Decrement	1	R -> L
-	Negation	1	R -> L
!	NOT	1	R -> L
*, /, %	Multiplication, division, modulus	2	L -> R
+, -	Addition, subtraction	3	L -> R
+	Concatenation	3	L -> R
<, <=	Less than, less than, or equal	4	L -> R
>, >=	Greater than, greater than, or equal	4	L -> R
==	Equal	5	L -> R
!=	Not equal	5	L -> R
===	Identity	5	L -> R
!==	Non-identity	5	L -> R
&&	AND	6	L -> R
	OR	6	L -> R
?:	Ternary	7	R -> L
=	Assignment	8	R -> L
+=, -=, and so on.	Arithmetic assignment	8	R -> L

## ❖ 2. JavaScript Fundamentals – Part 1 → 12. Strings and Template Literals

```
// Strings and Template Literals
const firstName = 'Jonas';
const job = 'teacher';
const birthYear = 1991;
const year = 2037;

const jonas = "I'm " + firstName + ', a ' + (year - birthYear) + ' year old ' + job + '!';
console.log(jonas);

const jonasNew = `I'm ${firstName}, a ${year - birthYear} year old ${job}!`;
console.log(jonasNew);

console.log(`Just a regular string...`);
console.log('String with \n\
multiple \n\
lines');

console.log(`String
multiple
lines`);
```

## ❖ 2. JavaScript Fundamentals – Part 1 → 13. Taking Decisions if else Statements

## ❖ 2. JavaScript Fundamentals – Part 1 → 15. Type Conversion and Coercion

In JavaScript:

- **type conversion** : when we manually convert from one type to another
- **type coercion** : coercion is when JavaScript automatically converts types behind the scenes for us that's necessary in some situation, but it happens implicitly, completely hidden from us.

```
// type conversion
const inputYear = '1991';
console.log(Number(inputYear), inputYear); //1991 '1991'
console.log(Number(inputYear) + 18); //2009

console.log(Number('Jonas')); // NaN (not a number actually means invalid number.)
console.log(typeof NaN); //number (It's still a number of somehow, but it's an invalid one.)

console.log(String(23), 23); // 23 23 (first one is string type, second is number)
```

So we have different types here, right? We have a string, a number and another string. it works this way because of type of coercion.

```
// type coercion
console.log('I am ' + 23 + ' years old'); //I am 23 years old
console.log('23' - '10' - 3); //10
console.log('23' + '10' + 3); //23103
console.log('23' / '2'); //11.5
console.log('23' > '18'); //true

let n = '1' + 1; // '11' ("+" operator convert number to string)
n = n - 1; //("-" convert string to number)
console.log(n); //10
```

## ❖ 2. JavaScript Fundamentals – Part 1 → 16. Truthy and Falsy Values

falsy values are values that are not exactly false, but will become false when we try to convert them into a Boolean.

// 5 falsy values: 0, "", undefined, null, NaN

```
// 5 falsy values: 0, "", undefined, null, NaN
console.log(Boolean(0)); //false
console.log(Boolean(undefined)); //false
console.log(Boolean('Jonas')); //true
console.log(Boolean({})); //true
console.log(Boolean("")); //false

**the Boolean() function used for test, never did this in your life
```

so the conversion to boolean is always implicit, not explicit, or in other words is always typed coercion that JavaScript does automatically behind the scenes and it happens in two scenarios:

- First, when using logical operators,
- and second in a logical context, like for example, in the condition of an if else statement.

## ❖ 2. JavaScript Fundamentals – Part 1 → Equality Operators: == vs. ===

the difference is that this one here with the three equals is called the **strict equality** operator.

```
// Equality Operators: == vs. ===
const age = '18';
if (age === 18) console.log('You just became an adult :D (strict)');

if (age == 18) console.log('You just became an adult :D (loose)');
const favourite = Number(prompt("What's your favourite number?"));
```

```

console.log(favourite);
console.log(typeof favourite);

if (favourite === 23) { // 22 === 23 -> FALSE
  console.log('Cool! 23 is an amazing number!')
} else if (favourite === 7) {
  console.log('7 is also a cool number')
} else if (favourite === 9) {
  console.log('9 is also a cool number')
} else {
  console.log('Number is not 23 or 7 or 9')
}

if (favourite !== 23) console.log('Why not 23?');

```

### prompt

```

var person = prompt("Please enter your name", "Harry Potter");

if (person != null) {
  document.getElementById("demo").innerHTML =
  "Hello " + person + "! How are you today?";
}

```

## ❖ 2. JavaScript Fundamentals – Part 1 → 19. Logical Operators

&&		!
----	--	---

## ❖ JavaScript Fundamentals – Part 1 → 21. The switch Statement

switch statement which is an alternative way of writing a complicated if/else statement

```

const day = 'friday';

switch (day) {
  case 'tuesday':
    console.log('Prepare theory videos');
    break;
  case 'wednesday':
  case 'thursday':
    console.log('Write code examples');
    break;
  case 'friday':
    console.log('Record videos');
    break;
  case 'saturday':

```

```

case 'sunday':
  console.log('Enjoy the weekend :D');
  break;
default:
  console.log('Not a valid day!');
}

```

## ❖ 2. JavaScript Fundamentals – Part 1 → 22. Statements and Expressions

✓ An expression is a piece of code that produces a value.

```

3 + 4
1991
true && false && !false

```

✓ And the statement is like a bigger piece of code that is executed and which does not produce a value on itself. For example if else is a statement

```

if (23 > 10) {
  const str = '23 is bigger';
}
const me = 'Jonas';
console.log(`I'm ${2037 - 1991} years old ${me}`);

```

JavaScript expects statements and expressions in different places. For example, in a template literal, we can only insert expressions, but not statements.

## ❖ 2. JavaScript Fundamentals – Part 1 → 23. The Conditional (Ternary) Operator

conditional operator allows us to write something similar to an if/else statement but all in one line.

```

// The Conditional (Ternary) Operator
const age = 23;
const drink = age >= 18 ? 'wine 🍷' : 'water 💧';
console.log(drink);

```

ternary operator is an expression, so we can use it in a template literal. (unlike a normal if/else statement)

## 2. JavaScript Fundamentals – Part 1 → 25. JavaScript Releases ES5, ES6+ and ESNext


### A BRIEF HISTORY OF JAVASCRIPT

- 1995** ➤ Brendan Eich creates the **very first version of JavaScript in just 10 days**. It was called Mocha, but already had many fundamental features of modern JavaScript!  
- 1996** ➤ Mocha changes to LiveScript and then to JavaScript, in order to attract Java developers. However, **JavaScript has almost nothing to do with Java** 🙄
- Microsoft launches IE, **copying JavaScript from Netscape** and calling it JScript; 
- 1997** ➤ With a need to standardize the language, ECMA releases ECMAScript 1 (ES1), the first **official standard for JavaScript** (ECMAScript is the standard, JavaScript the language in practice); 
- 2009** ➤ ES5 (ECMAScript 5) is released with lots of great new features;
- 2015** ➤ ES6/ES2015 (ECMAScript 2015) was released: **the biggest update to the language ever!**
- ECMAScript changes to an **annual release cycle** in order to ship less features per update 🙏
- 2016 – ∞** ➤ Release of ES2016 / ES2017 / ES2018 / ES2019 / ES2020 / ES2021 / ... / ES2089 😄

### BACKWARDS COMPATIBILITY: DON'T BREAK THE WEB!

```
// ES1 Code
function add(n) {
  var x = 5 + add.arguments[0];
  return x;
}
```

1997



**BACKWARDS COMPATIBLE**


Modern JavaScript Engine

2020

**DON'T BREAK THE WEB!**

Modern JavaScript Engine

2020



**NOT FORWARDS COMPATIBLE**

```
// ES2089 Code 🤔
c int add n <=> int 5 + n
```

2089

- Old features are **never** removed;
- Not really new versions, just **incremental updates** (releases)
- Websites keep working **forever!**

how can we use modern JavaScript today? Because browsers that users are using might be old and there's no forwards compatibility. Right? So to answer the question, how we can use modern JavaScript today, we need to consider two distinct scenarios, **development** and **production**.



- So **the development phase** is simply when you're building the site or application on your computer. To ensure you can use the latest JavaScript features in this phase. All you have to do is to use the most Up ToDate version of the Google Chrome browser.
- The second scenario is **production**, which is when your web application is finished. You deploy it on the internet and it's then running in your users' browsers. And this is where problems might appear, because this is the part that we actually can't control. We cannot control which browser the user uses. And we also can't assume that all our users always use the latest browsers, right. Now, the solution to this problem is to basically convert these modern JavaScript versions back to ES5 using a process called **transpiling**. and also **polyfilling**. We will use a tool called Babel later in the course to transpile our code.

because you're using the most Up ToDate browser during development, transpiling back to ES5 is only necessary after your app is developed and you want to ship it to your users  
How different JavaScript releases can be used today:

- So first off ES5 is of course fully supported in all browsers today, all the way down to internet Explorer nine from 2011. So we can assume that ES5 is safe to be used at this point, which is the reason why we use it as a target for transpiling.

Now about the newer releases, ES6, ES7 and all the way to ES2020, as of mid-2020, they are actually quite well supported already in all modern browsers. And we usually call all the current versions together,

ES6 plus. So right now that's from ES6 to ES2020, and basically all together, they are the modern JavaScript. Now it's in this modern JavaScript. So in this ES6 plus where transpiling comes in,

as I mentioned earlier.

*if you want to stay up to date with what features are currently supported in which browser you can check out the ES6 compatibility table.*


Next, there are also the future releases of the language like ES2021, ES2022 and so on. And these future releases together are many times called **ESNext**. Now, why is this even relevant? Well, because most browsers actually start implementing new features even before they enter the official ECMAScript specification. That's possible because as new features are proposed, they have to go through four stages, starting with stage one, where they are first admitted all the way to stage four, at which point they enter the language officially. But when a feature is at stage three, browsers can be pretty sure it will eventually pass to stage four. And so they're gonna start implementing that feature while still in stage three. And there is a lot more to be said about this.

You can find tons of information about us online. If you want to learn more. And a great place to start is actually once more, this compatibility table, that's up here on the slide.

ES6 compatibility table: <https://kangax.github.io/compat-table/es6/>

# HOW TO USE MODERN JAVASCRIPT TODAY

 **During development:** Simply use the latest Google Chrome!

 **During production:** Use Babel to transpile and polyfill your code (converting back to ES5 to ensure browser compatibility for all users).



<http://kangax.github.io/compat-table>

ES5

- ☞ Fully supported in all browsers (down to IE 9 from 2011);
- ☞ Ready to be used today 🍌

ES6/ES2015




ES2020

- ☞ **ES6+:** Well supported in all **modern** browsers;
- ☞ No support in **older** browsers;
- ☞ Can use **most** features in production with transpiling and polyfilling 😊

ES2021 – ∞

- ☞ **ESNext:** Future versions of the language (new feature proposals that reach Stage 4);
- ☞ Can already use **some** features in production with transpiling and polyfilling.

 Will add new videos

(As of 2020)

**BABEL**

## ❖ 3. JavaScript Fundamentals – Part 2 → 2. Activating Strict Mode

strict mode is a special mode that we can activate in JavaScript, which makes it easier for us to write a secure JavaScript code.

- ✓ Write in first line of JS file
- ✓ Just comments before this are allowed
- ✓ We actually can also activate strict mode, only for a specific function or a specific block. But I don't really see the point in doing that

strict mode makes it easier for us developers to avoid accidental errors. So basically, it helps us introduce the bugs into our code and that's because of 2 reasons.

- First, strict mode forbids us to do certain things
- Second, it will actually create visible errors for us in certain situations in which without strict mode JavaScript will simply fail silently without letting us know that we did a mistake.

EX:

```
'use strict'  
let hasDriversLicense = false;  
const passTheE = true;  
if (passTheE) hasDriverLicense = true; //miss the 's' (in strict mode we get error but in other  
wat we didn't see any error  
if (hasDriversLicense) console.log('you can drive')
```

- another thing that strict mode does is to introduce a short list of variable names that are reserved for features that might be added to the language a bit later.

```
'use strict'
```

```
let interface = 'Audio'; // we get an error (interface is a reserved word)
```

### ❖ 3. JavaScript Fundamentals – Part 2 → 3. Functions

Well in the most simple form a function is simply a piece of code that we can reuse over and over again in our code.

### 3. JavaScript Fundamentals – Part 2 → 7. Reviewing Functions

**FUNCTIONS REVIEW: 3 DIFFERENT FUNCTION TYPES**

- Function declaration**  
Function that can be used before it's declared
- Function expression**  
Essentially a function value stored in a variable
- Arrow function**  
Great for a quick one-line functions. Has no `this` keyword (more later...)

```
function calcAge(birthYear) {  
  return 2037 - birthYear;  
}  
  
const calcAge = function (birthYear) {  
  return 2037 - birthYear;  
};  
  
const calcAge = birthYear => 2037 - birthYear;
```

Three different ways of writing functions, but they all work in a similar way: receive **input** data, **transform** data, and then **output** data.

in fact in JavaScript, functions are actually just values. (fact an expression and expressions produce values.)

**Parameters:** Function *names* (param1,param2){----}

**Arguments:** *names*(arg1,arg2)

main practical difference is that we can actually call function declarations before they are defined in the code (because hoisting).

*Personally, I prefer to use function expressions because this then forces me into a nice structure where I have to define all the functions first at the top of the code and only then I can call them.*

### ❖ 3. JavaScript Fundamentals – Part 2 → 9. Introduction to Arrays

1. **Arrays:** only primitive values, are immutable. But an Array is **not** a primitive value. So, we can mutate(change) the array even though they were declared with `const`. but cannot change whole array

```
const friends = ['Michael', 'Steven', 'Peter']; // (using [] called the literal syntax )
console.log(friends); // 'Michael', 'Steven', 'Peter'

friends[2] = 'Jay';
console.log(friends); // 'Michael', 'Steven', 'Jay'
```

### 3. JavaScript Fundamentals – Part 2 → Basic Array Operations (Methods):

JS has some built in functions that can apply directly on arrays and these are called methods

- .includes() // true/false
- .indexOf() // index of searched item / -1 if not find
- .push() // adds one or more elements to the **end** of an array and returns the new length of the array.
- .pop() //remove **last** element (returned the element)
- .shift() //remove elements from the **beginning** of the array (shift method also returns removed elements)
- .unshift() //adds elements to the **beginning** of the array (unshift method also returns the length of the new array)

```
// Basic Array Operations (Methods)
const friends = ['Michael', 'Steven', 'Peter'];
// Add elements
const newLength = friends.push('Jay'); // friends array will be changed
console.log(friends); // 'Michael', 'Steven', 'Peter', 'Jay'
console.log(newLength); // 4

friends.unshift('John');
console.log(friends); // 'John', 'Michael', 'Steven', 'Peter', 'Jay'

// Remove elements
friends.pop(); // Last
const popped = friends.pop();
console.log(popped); // 'Peter'
console.log(friends); // 'John', 'Michael', 'Steven'

friends.shift(); // First
console.log(friends); // 'Michael', 'Steven'

console.log(friends.indexOf('Steven')); // 1
console.log(friends.indexOf('Bob')); // -1

friends.push(23);
console.log(friends.includes('Steven')); // true
```

```

console.log(friends.includes('Bob')); // false
console.log(friends.includes(23)); // true

if (friends.includes('Steven')) {
  console.log('You have a friend called Steven'); //You have a friend called Steven
}

```

### ❖ 3. JavaScript Fundamentals – Part 2 → 12. Introduction to Objects

2. **Objects:** in arrays, there is no way of giving these elements a name. And so we can't reference them by name, but only by their order number to solve that problem, we have another data structure in JavaScript, which is objects.

```

const jonasArray = [
  'Jonas',
  'Schmedtmann',
  2037 - 1991,
  'teacher',
  ['Michael', 'Peter', 'Steven']
];

const jonas = {
  firstName: 'Jonas',
  lastName: 'Schmedtmann',
  age: 2037 - 1991,
  job: 'teacher',
  friends: ['Michael', 'Peter', 'Steven']
};

```

\*\*The big difference between objects and arrays, is that in objects, the order of these values does not matter at all when we want to retrieve them.

### ❖ 3. JavaScript Fundamentals – Part 2 → 13. Dot vs. Bracket Notation

```

const jonas = {
  firstName: 'Jonas',
  lastName: 'Schmedtmann',
  age: 2037 - 1991,
  job: 'teacher',
  friends: ['Michael', 'Peter', 'Steven']
};

console.log(jonas);

console.log(jonas.lastName);
console.log(jonas['lastName']);

const nameKey = 'Name';
console.log(jonas['first' + nameKey]);

```

```
console.log(jonas['last' + nameKey]);
```

```
console.log(jonas.'last' + nameKey) // error 'last' here is an unspected string, the reason why we need the brackets notation and dot notation
```

dot and [] operator precedence is: left to right

### ❖ 3. JavaScript Fundamentals – Part 2 → 14. Object Methods

Objects can hold arrays, objects function is just a value then that means that we can create a key value pair in which the value is a function.

any **function** that is attached to an object is called a method.

```
const jonas = {
  firstName: 'Jonas',
  lastName: 'Schmedtmann',
  birthYear: 1991,
  job: 'teacher',
  friends: ['Michael', 'Peter', 'Steven'],
  hasDriversLicense: true,

  calcAge1: function (birthYear) {
    return 2037 - birthYear;
  },

  calcAge2: function () { //using this keyword
    // console.log(this);
    return 2037 - this.birthYear;
  },
  //instead of using "jonas.calcage()" multiple time, just calculate the age once, then store it in the object, and then when we need it at a later point, we can then just retrieve the age as a property from the object.
  calcAge: function () { // using this to store value in the object
    this.age = 2037 - this.birthYear;
    return this.age;
  },

  getSummary: function () {
    return `${this.firstName} is a ${this.calcAge()}-year old ${jonas.job}, and he has ${this.hasDriversLicense ? 'a' : 'no'} driver's license.`
  }
};

console.log(jonas.calcAge());
```

“**This**” variable is basically equal to **the object on which the method is called**, or in other words, it is equal to the object calling the method.

### ❖ 3. JavaScript Fundamentals – Part 2 → 16. Iteration The for Loop

```
for (let rep = 1; rep <= 30; rep++) {  
  console.log(`Lifting weights repetition ${rep} 🏋️`);  
}
```

### ❖ 3. JavaScript Fundamentals – Part 2 → 17. Looping Arrays, Breaking and Continuing

\*\*Parts of a for: for(Counter ; condition ; updating the counter)

- with “**continue**” we can exit the current iteration of the loop.
- “**break**” completely terminate the whole loop (not the current iteration)

```
// Looping Arrays, Breaking and Continuing
```

```
const jonas = [  
  'Jonas',  
  'Schmedtmann',  
  2037 - 1991,  
  'teacher',  
  ['Michael', 'Peter', 'Steven'],  
  true  
];  
const types = [];  
  
// console.log(jonas[0])  
// console.log(jonas[1])  
// ...  
// console.log(jonas[4])  
// jonas[5] does NOT exist  
  
for (let i = 0; i < jonas.length; i++) {  
  // Reading from jonas array  
  console.log(jonas[i], typeof jonas[i]);  
  
  // Filling types array  
  // types[i] = typeof jonas[i];  
  types.push(typeof jonas[i]);  
}  
  
console.log(types);
```

```
const years = [1991, 2007, 1969, 2020];
```

```

const ages = [];

for (let i = 0; i < years.length; i++) {
  ages.push(2037 - years[i]);
}
console.log(ages);

// continue and break
console.log('--- ONLY STRINGS ---')
for (let i = 0; i < jonas.length; i++) {
  if (typeof jonas[i] !== 'string') continue;

  console.log(jonas[i], typeof jonas[i]);
}

console.log('--- BREAK WITH NUMBER ---')
for (let i = 0; i < jonas.length; i++) {
  if (typeof jonas[i] === 'number') break;

  console.log(jonas[i], typeof jonas[i]);
}

```

### ❖ 3. JavaScript Fundamentals – Part 2 → 18. Looping Backwards and Loops in Loops

First, we will loop over an array backwards, and then second, we will also create a loop

```

for (let i = jonas.length - 1; i >= 0; i--) {
  console.log(i, jonas[i]);
}

for (let exercise = 1; exercise < 4; exercise++) {
  console.log(`----- Starting exercise ${exercise}`);

  for (let rep = 1; rep < 6; rep++) {
    console.log(`Exercise ${exercise}: Lifting weight repetition ${rep} 🏋️`);
  }
}

```

### ❖ 3. JavaScript Fundamentals – Part 2 → 19. The while Loop

```

// The while Loop
for (let rep = 1; rep <= 10; rep++) {
  console.log(`Lifting weights repetition ${rep} 🏋️`);
}

```



```
let rep = 1;
while (rep <= 10) {
  console.log(`WHILE: Lifting weights repetition ${rep} 🏋️`);
  rep++;
}
```

```
let dice = Math.trunc(Math.random() * 6) + 1;
```

```
while (dice !== 6) {
  console.log(`You rolled a ${dice}`);
  dice = Math.trunc(Math.random() * 6) + 1;
  if (dice === 6) console.log('Loop is about to end...');
}
```

## ❖ 5. Developer Skills & Editor Setup → 3. Setting up Prettier and VS Code

Prettier configs:

1. Install prettier
2. File/Preferences/Settings => set "default formatter" to prettier
3. File/Preferences/Settings => check the "formatonsave"
4. Create ".prettierrc" file

Use prettier docs for config commands

```
{
  "singleQuote": true,
  "arrowParens": "avoid"
}
```

\*\*\* using this to ignore Prettier for a line

```
// prettier-ignore
```

VS code user snippets:

1. File/Preferences/User snippets/New Global snippets file  
Uncomment the last part and set it to :

```
"Print to console": {
  "scope": "javascript,typescript",
  "prefix": "cl",
  "body": [
    "console.log();",
  ],
  "description": "Log output to console"
}
```

## ❖ 5. Developer Skills & Editor Setup → 4. Installing Node.js and Setting Up a Dev Environment

Live reload:

1. Using VSCode live server extension
2. Using node.js
  - a. Install node.js
  - b. Use terminal -> check "node -v" (if you see any ver number node installed)
  - c. npm install liver-server -g
  - d. live-server

*if Execution Policy on your computer is Restricted :*  
*(get error:ps1 cannot be loaded because running scripts is disabled on this system )*  
 ♣ open PowerShell as admin  
 ♣ Set-ExecutionPolicy Unrestricted

❖ **5.Developer Skills & Editor Setup → 9. Debugging with the Console and Breakpoints**

```
console.log('log');//
console.warn('warn');// warn
console.error('error');// error
console.table('sampleObject');// log sample object in a table style
```

❖ **7. JavaScript in the Browser DOM and Events Fundamentals → 4. What's the DOM and DOM Manipulation**

## WHAT IS THE DOM?

DOM

DOCUMENT OBJECT MODEL: STRUCTURED REPRESENTATION OF HTML DOCUMENTS. ALLOWS JAVASCRIPT TO ACCESS HTML ELEMENTS AND STYLES TO MANIPULATE THEM.

Change text, HTML attributes, and even CSS styles

Tree structure, generated by browser on HTML load

❖ **7. JavaScript in the Browser DOM and Events Fundamentals → 6. Handling Click Events**

```
document.querySelector('tag').addEventListener('click', function () {
```

```
// do something  
}
```

## ❖ 7. JavaScript in the Browser DOM and Events Fundamentals → 8. Manipulating CSS Styles

```
document.querySelector('body').style.backgroundColor = 'red';
```

- Use “.style.attribute” to Manipulate the CSS of the element
- Use camelCase to write multi-word CSS attributes
- The value always is string

## ❖ 7. JavaScript in the Browser DOM and Events Fundamentals → 14. Handling an Esc Keypress Event

```
document.addEventListener('keydown', function (e) {  
  // console.log(e.key);  
  
  if (e.key === 'Escape' && !modal.classList.contains('hidden')) {  
    closeModal();  
  }  
});
```


## ❖ 8. How JavaScript Works Behind the Scenes → 3. An High-Level Overview of JavaScript

### DECONSTRUCTING THE MONSTER DEFINITION

High-level

- Garbage-collected
- Interpreted or just-in-time compiled
- Multi-paradigm
- Prototype-based object-oriented
- First-class functions
- Dynamic
- Single-threaded
- Non-blocking event loop

Any computer program needs resources:

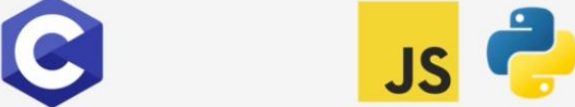


LOW-LEVEL

HIGH-LEVEL

Developer has to manage resources manually

Developer does NOT have to worry, everything happens automatically



JS is garbage-collection, which is basically an algorithm inside the JavaScript engine, which automatically removes old, unused objects from the computer memory in order not to clog

it up with unnecessary stuff. So it's a little bit like JavaScript has a cleaning guy who cleans our memory from time to time so that we don't have to do it manually in our code.

## DECONSTRUCTING THE MONSTER DEFINITION

High-level

Garbage-collected

Interpreted or just-in-time compiled

Multi-paradigm

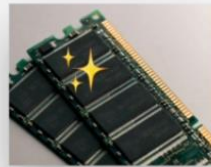
Prototype-based object-oriented

First-class functions

Dynamic

Single-threaded

Non-blocking event loop



Cleaning the memory so we don't have to

The computer's processor only understands zeros and ones, that's right. Ultimately, every single program needs to be written in 0 and 1, which is also called machine code. We simply write human-readable JavaScript code, but this code eventually needs to be translated to machine code. And that step can be either compiling or interpreting.

## DECONSTRUCTING THE MONSTER DEFINITION

High-level

Garbage-collected

Interpreted or just-in-time compiled

Multi-paradigm

Prototype-based object-oriented

First-class functions

Dynamic

Single-threaded

Non-blocking event loop

```
document.querySelector(".again").addEventListener("click", () => {
  document.querySelector(".message").textContent = "Start guessing...";
  document.querySelector(".number").textContent = "?";
  document.querySelector(".guess").value = "";
  score = 20;
  document.querySelector(".score").textContent = score;
  number = Math.floor(Math.random() * 20) + 1;
});
```

Abstraction over 0s and 1s

↓  
CONVERT TO MACHINE CODE = COMPILING

```
0111010101110101001001110101110101011100010101100010
0101010111010101110101110000001110010111011101100
1101001000001001011101010101010111010101010101010
000111010010010011101010111010101110010101111010
1101010101010101110101110101000101010101010101010
11101000100011101000010101110001010001010111010101
0101010101000101010111010101011101010101010101011
11010101011101010001010111010101011101010100010101
1100011101110101011101001010101010101010101010101
011010101010101010101011101011101011100111001110
1101010111010101110101010101010101010101010101010
```

More about this **Later in this Section** 📌

In programming, a paradigm is an approach and an overall mindset of structuring our code, which will ultimately direct the coding style and technique in a project that uses a certain paradigm. three popular paradigms are;

- **Procedural**
- **object-oriented**
- **functional programming**

we can classify paradigms as imperative or as declarative

many languages are only procedural or only object-oriented or only functional, but JavaScript does all of it.

## DECONSTRUCTING THE MONSTER DEFINITION

**High-level**

**Garbage-collected**

**Interpreted or just-in-time compiled**

**Multi-paradigm**

Prototype-based object-oriented

First-class functions

Dynamic

Single-threaded

Non-blocking event loop

👉 **Paradigm:** An approach and mindset of structuring code, which will direct your coding style and technique.

The one we've been using so far

- 1 Procedural programming
- 2 Object-oriented programming (OOP)
- 3 Functional programming (FP)

👍 Imperative vs. 👎 Declarative

More about this later in **Multiple Sections** 👉

about the object-oriented nature of JavaScript, it is a prototype-based, object-oriented approach. what does that mean? first, almost everything in JavaScript is an object, except for primitive values such as numbers, strings, et cetera. But arrays, for example, are just object.

Now, have you ever wondered why we can create an array and then use the push method on it, for example? Well, it's because of prototypal inheritance. Basically, we create arrays from an array blueprint, which is like a template and this is called the prototype. This prototype contains all the array methods and the arrays that we create in our code then inherit the methods from the blueprint so that we can use them on the arrays.

# DECONSTRUCTING THE MONSTER DEFINITION

High-level

Garbage-collected

Interpreted or just-in-time compiled

Multi-paradigm

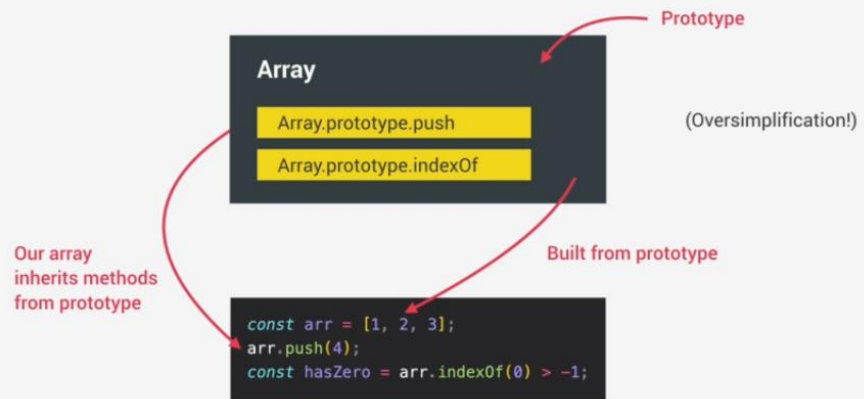
Prototype-based object-oriented

First-class functions

Dynamic

Single-threaded

Non-blocking event loop



More about this in Section **Object Oriented Programming** 🖱️

Udacity

# DECONSTRUCTING THE MONSTER DEFINITION

High-level

Garbage-collected

Interpreted or just-in-time compiled

Multi-paradigm

Prototype-based object-oriented

First-class functions

Dynamic

Single-threaded

Non-blocking event loop

- 🖱️ In a language with **first-class functions**, functions are simply **treated as variables**. We can pass them into other functions, and return them from functions.

```
const closeModal = () => {  
  modal.classList.add("hidden");  
  overlay.classList.add("hidden");  
};  
  
overlay.addEventListener("click", closeModal);
```

Passing a function into another function as an argument: First-class functions!

More about this in Section **A Closer Look at Functions** 🖱️

Udacity

# DECONSTRUCTING THE MONSTER DEFINITION

High-level

Garbage-collected

Interpreted or just-in-time compiled

Multi-paradigm

Prototype-based object-oriented

First-class functions

Dynamic

Single-threaded

Non-blocking event loop

☛ Dynamically-typed language:

No data type definitions. Types becomes known at runtime

```
let x = 23;
let y = 19;
x = "Jonas";
```

Data type of variable is automatically changed

~~C++~~ ~~Java~~ ~~Ruby~~

let's now finally talk about the single-thread and the non-blocking event loop concurrency model. this is a really complex topic And therefore, I'm not gonna go into specifics here just yet, but instead I will just define some things here.

First, what actually is a concurrency model? Well, it's just a fancy term that means how the JavaScript engine handles multiple tasks happening at the same time. But why do we need that? Well, because JavaScript itself runs in one single-thread, which means that it can only do one thing at a time and therefore we need a way of handling multiple things happening at the same time. in computing, a thread is like a set of instructions that is executed in the computer's CPU. So basically, the thread is where our code is actually executed in a machine's processor. All right. But what if there is a long-running task, like fetching data from a remote server? Well, it sounds like that would block the single thread where the code is running, right? But of course we don't want that. What we want is so-called non-blocking behavior and how do we achieve that? Well, by using a so-called event loop. The event loop takes long-running tasks, executes them in the background and then puts them back in the main thread once they are finished. And this is, in a nutshell, JavaScript's non-blocking event loop concurrency model with a single thread.

It sounds like a mouthful for sure but in the end, it really just compresses to this.

# DECONSTRUCTING THE MONSTER DEFINITION

High-level

Garbage-collected

Interpreted or just-in-time compiled

Multi-paradigm

Prototype-based object-oriented

First-class functions

Dynamic

Single-threaded

Non-blocking event loop

☛ **Concurrency model:** how the JavaScript engine handles multiple tasks happening at the same time.

↓ Why do we need that?

☛ JavaScript runs in one **single thread**, so it can only do one thing at a time.

↓ So what about a long-running task?

☛ Sounds like it would block the single thread. However, we want non-blocking behavior!

↓ How do we achieve that? (Oversimplification!)

☛ By using an **event loop**: takes long running tasks, executes them in the "background", and puts them back in the main thread once they are finished.

More about this **Later in this Section** 📌

© 2015

## ❖ 8. How JavaScript Works Behind the Scenes → 4. The JavaScript Engine and Runtime

JavaScript engine is simply a computer program that executes JavaScript code.

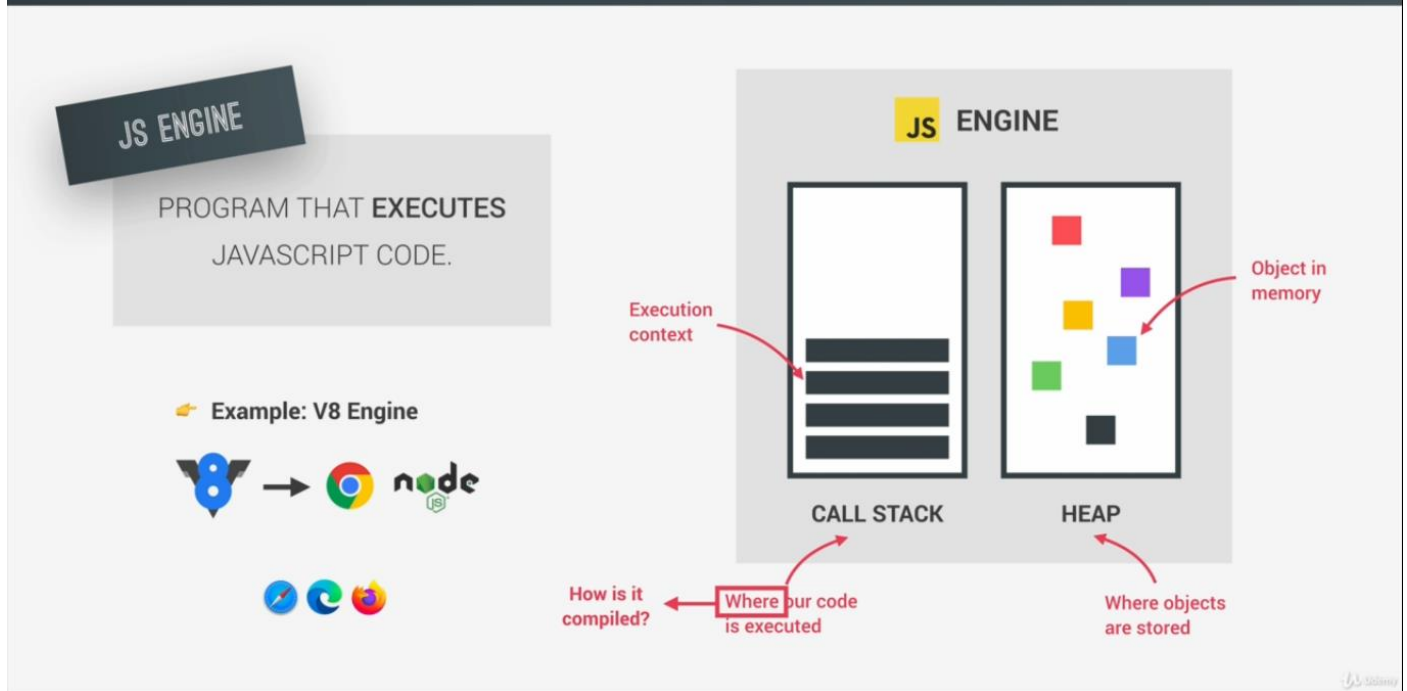
every browser has its own JavaScript engine but probably the most well known engine is Google's V-Eight. The V eight engine powers Google Chrome, but also Node.js which is that JavaScript runtime, the one that we can use to build server side applications with JavaScript, so outside of any browser.

any JavaScript engine always contains a call stack and a heap. The call stack is where our code is actually executed using something called execution contexts. Then the heap is an unstructured memory pool which stores all the objects that our application needs.

now the question is how the code is compiled to machine code so that it actually can be executed afterwards.



# WHAT IS A JAVASCRIPT ENGINE?



But first we need to make a quick computer science side note here and talk about the difference between compilation and interpretation. We learned that the computer's processor only understands zeros and ones and that's therefore every single computer program ultimately needs to be converted into this machine code and this can happen using compilation or interpretation.

So in compilation, the entire source code is converted into machine code at once. And this machine code is then written into a portable file that can be executed on any computer. First, the machine code is built and then it is executed in the CPU so in the processor. And the execution can happen way after the compilation of course.

For example, any application that you're using on your computer right now has been compiled before and you're now executing it way after it's compilation.

Now, on the other hand in interpretation, there is an interpreter which runs through the source code and executes it line by line.

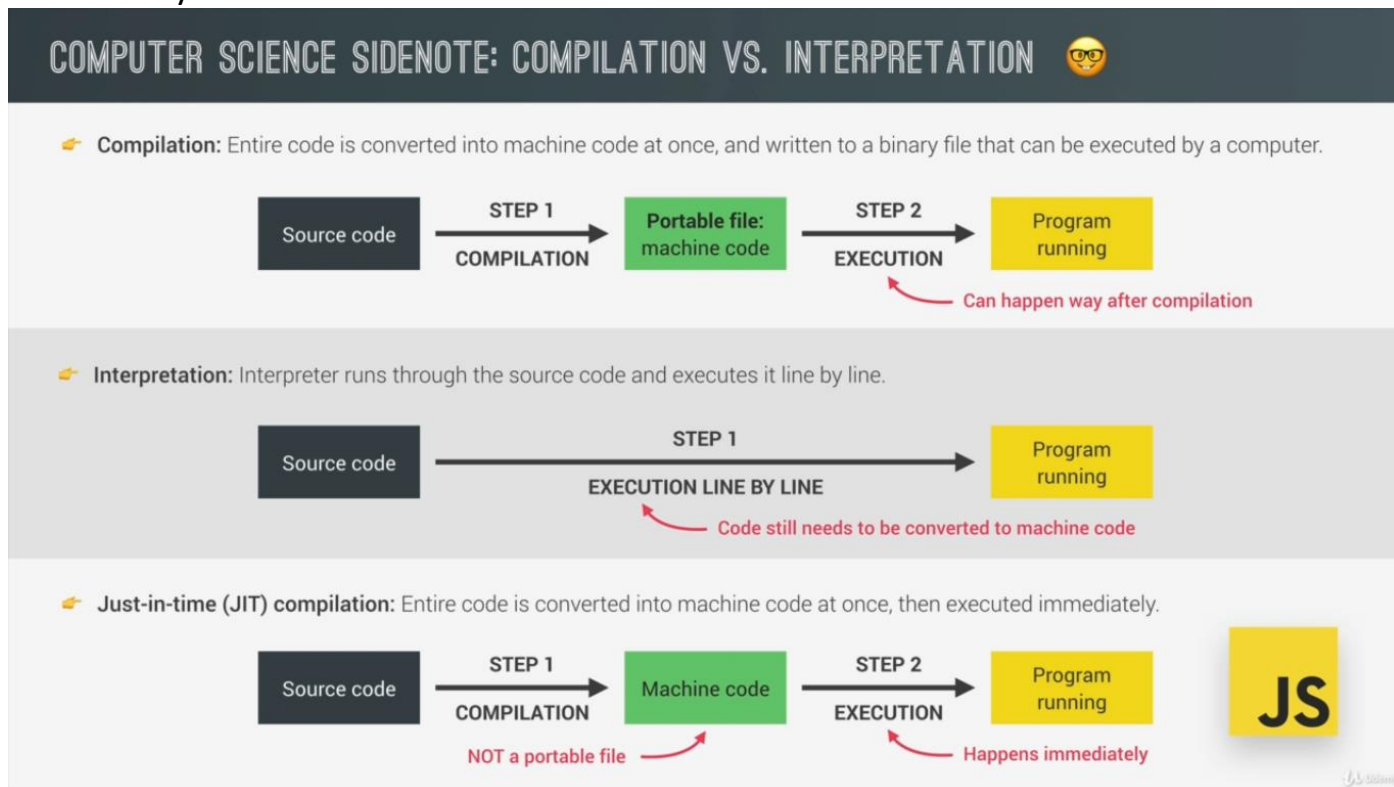
So here we do not have the same two steps as before. Instead the code is read and executed all at the same time.

Of course the source code still needs to be converted into machine code, but it simply happens right before it's executed and not ahead of time. Now JavaScript used to be a purely interpreted language but the problem with interpreted languages is that they are much, much slower than compiled languages. This used to be okay for JavaScript, but now with modern JavaScript and fully fledged web applications that we built and use today, low performance is no longer acceptable.

Just imagine you were using Google maps in your browser and you were dragging the map and each time you dragged it would take one second for it to move.

That would be completely unacceptable, right? Now many people still think that JavaScript is an interpreted language but that's not really true anymore. So instead of simple interpretation modern JavaScript engine now use a mix between compilation and interpretation which is called just-in-time compilation.

This approach basically compiles the entire code into machine code at once and then executes it right away. So we still have the two steps of regular ahead of time compilation but there is no portable file to execute. And the execution happens immediately after a compilation. And this is perfect for JavaScript as it's really a lot faster than just executing code line by line.



So as a piece of JavaScript code enters the engine the first step is to parse the code which essentially means to read the code. During the parsing process, the code is parsed into a data structure called the abstract syntax tree or AST.

This works by first splitting up each line of code into pieces that are meaningful to the language like the `const` or `function` keywords, and then saving all these pieces into the tree in a structured way.

This step also checks if there are any syntax errors and the resulting tree will later be used to generate the machine code.

Now let's say we have a very simple program. All it does is to declare a variable like this, and this is what the AST for just this one line of code looks like. So we have a variable declaration which should be a constant with the name `X` and the value of `23`.

And besides that there is a lot of other stuff here, as you can see.

So just imagine what it would look like for a large real application. And of course you don't need to know what an AST looks like. This is just for curiosity okay.

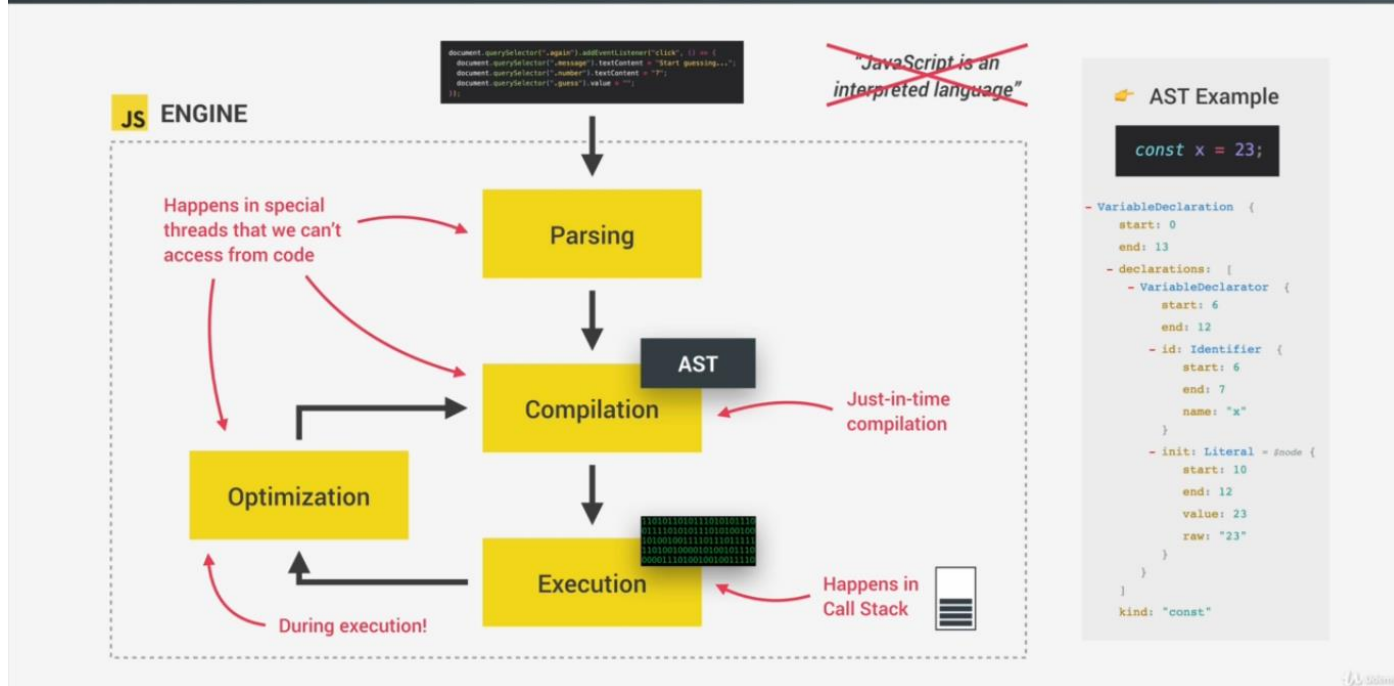
Now sometimes I get asked if this tree has anything to do with the DOM tree and the answer is a very clear no. So this tree has absolutely nothing to do with the DOM. It is not related in any way. It's just a representation of our entire code inside the engine. Anyway, the next step is compilation which takes the generated AST and compiles it into machine code just as we learned in the previous slide. This machine code then gets executed right away because remember modern JavaScript engines use just-in-time compilation.

And remember execution happens in the JavaScript engines call stack but we will dig deeper into this in the next lecture. All right, so far so good. We have our code running so we can finish here, Right? Well, not so fast because modern JavaScript engines actually have some pretty clever optimization strategies.

What they do is to create a very unoptimized version of machine code in the beginning just so that it can start executing as fast as possible. Then in the background, this code is being optimized and recompiled during the already running program execution. And this can be done most of the times and after each optimization the unoptimized code is simply swept for the new more optimized code without ever stopping execution of course.

And this process is what makes modern engines such as the V8 so fast and all this parsing, compilation and optimization happens in some special threads inside the engine that we cannot access from our code. So completely separate from the main thread that is basically running into call stack executing our own code. Now different engines implement in slightly different ways, but in a nutshell this is what modern just-in-time compilation looks like for JavaScript. And the next time someone tells you JavaScript is an interpreted language, you just show them this slide so that they can learn how it really works.

## MODERN JUST-IN-TIME COMPILATION OF JAVASCRIPT



Alright, so we looked at the JavaScript engine and how it works behind the scenes in quite some detail. Now to round off this lecture let's also take a look at what a JavaScript runtime is.

And in particular, the most common one, which is the browser and by doing this, we can get the bigger picture of how all the pieces fit together when we use JavaScript.

And so this is a really important slide. So we can imagine a JavaScript runtime as a big box or a big container which includes all the things that we need in order to use JavaScript in this case, in the browser. And to heart of any JavaScript, runtime is always a JavaScript engine. So exactly the one we've been talking about. That's why it makes sense to talk about engines and runtimes together. Without an engine there is no runtime and there is no JavaScript at all. However the engine alone is not enough. In order to work properly, we also need access to the web APIs, and we talked about web APIs before, remember? So that's everything related to the DOM or timers or even the console.log that we use all the time.

So essentially web APIs are functionalities provided to the engine, but which are actually not part of the JavaScript language itself. JavaScript simply gets access to these APIs through the global window object. But it still makes sense that the web APIs are also part of the runtime, because again a runtime is just like a box that contains all the JavaScript related stuff that we need. Next a typical JavaScript runtime also includes a so called callback queue.

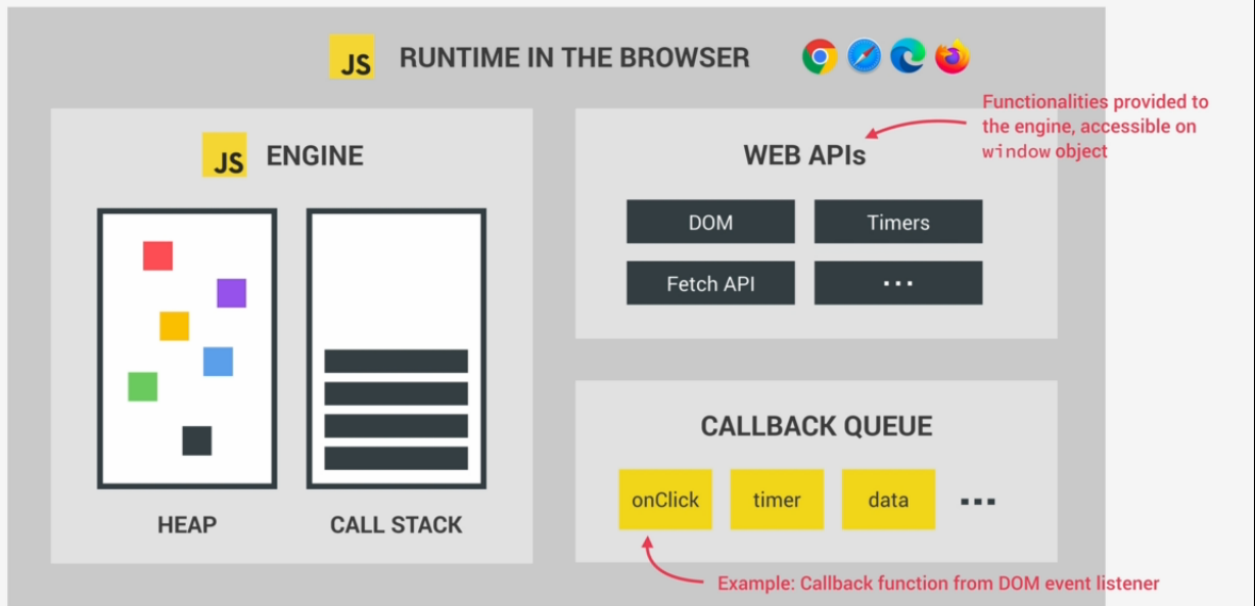
This is a data structure that contains all the callback functions that are ready to be executed. For example we attach event handler functions to DOM elements like a button to react to certain events, right? And these event handler functions are also called callback functions okay. So as the event happens, for example a click, the callback function will be called.

And here is how that actually works behind the scenes. So the first thing that actually happens after the event is that the callback function is put into the callback queue. Then when the stack is empty the callback function is passed to the stack so that it can be executed. And this happens by something called the event loop.

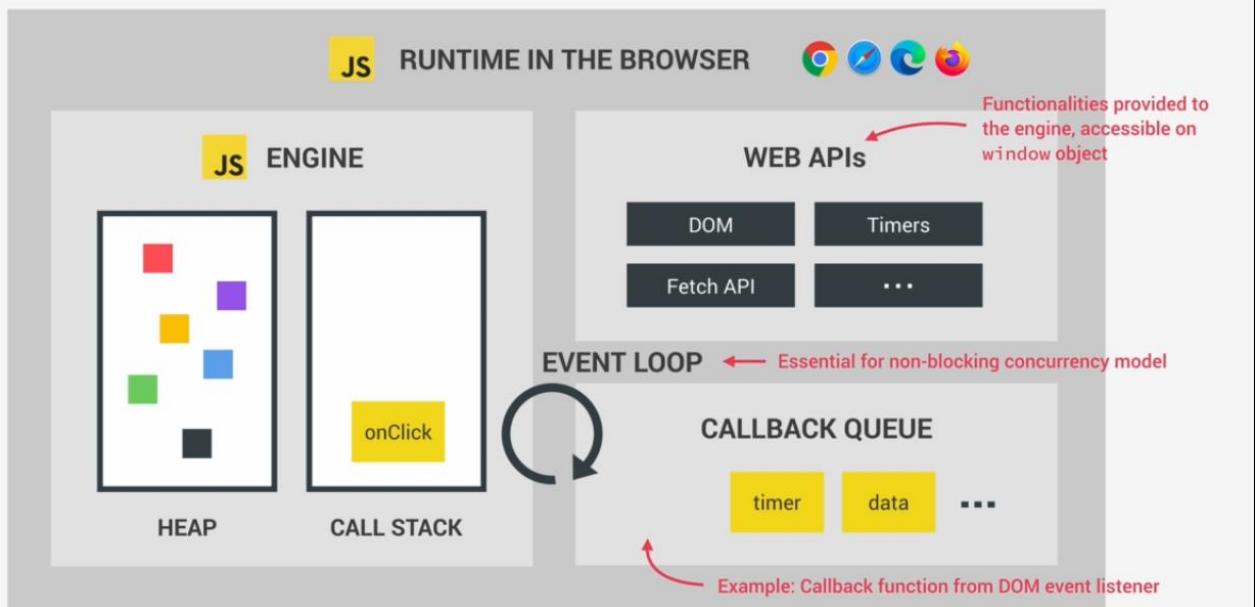
So basically the event loop takes callback functions from the callback queue and puts them in the call stack so that they can be executed. And remember how I said in the last lecture that the event loop is how JavaScript's nonblocking concurrency model is implemented? Well, here is an overview of how that works.

Now we will go over why this makes JavaScript nonblocking in a special lecture about the event loop later in the course, because this is really a fundamental piece of JavaScript development that every developer needs to understand deeply. Alright, so as they already said the focus in this course is on JavaScript in the browser and that's why we analyzed the browser JavaScript runtime.

## THE BIGGER PICTURE: JAVASCRIPT RUNTIME

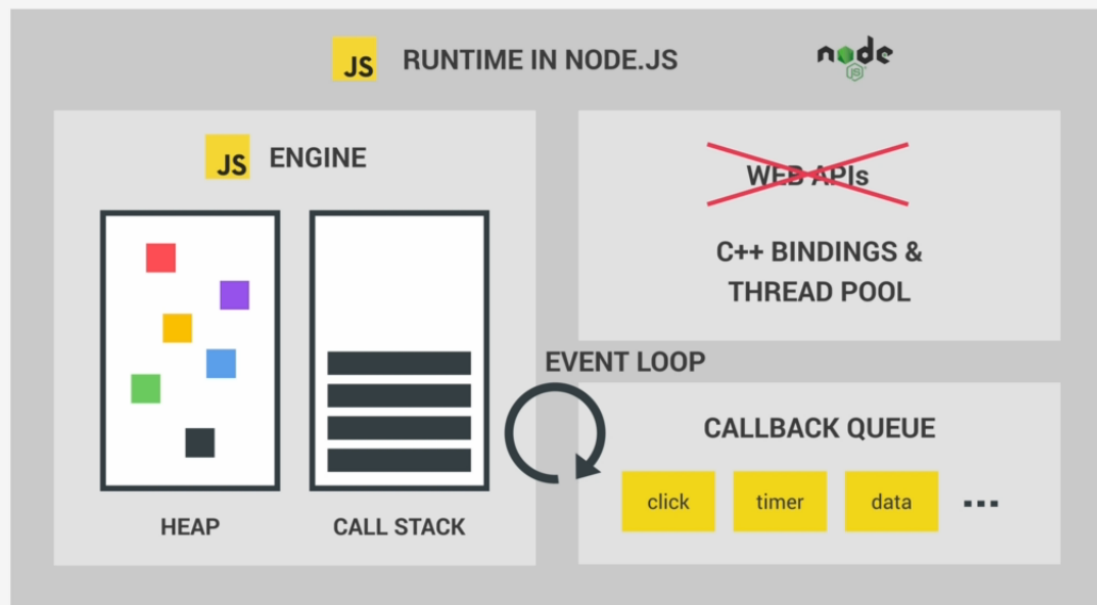


## THE BIGGER PICTURE: JAVASCRIPT RUNTIME



However, it's also important to remember that JavaScript can exist outside of browsers, for example, in Node.js. And so here is what the node JS JavaScript runtime looks like. It's pretty similar, but since we don't have a browser of course, we can't have the web APIs because it's the browser who provides these. Instead we have multiple C++ bindings and a so called thread pool. Now details don't matter here at all. I just want you to know that different JavaScript runtimes do exist.

## THE BIGGER PICTURE: JAVASCRIPT RUNTIME



### ❖ 8. How JavaScript Works Behind the Scenes → 5. Execution Contexts and The Call Stack

How is JavaScript code executed? We already know that it happens in a call stack in the engine, but let's dig a bit deeper now.

And let's start

by supposing that our code was just finished compiling.

Just in the way that we learned in the last lecture.

So the code is now ready to be executed.

What happens then, is that a so-called global execution context is created for the top-level code.

And top-level code is basically code that is not inside any function.

So again, in the beginning

only the code that is outside of functions will be executed.

And this makes sense, right?

Functions should only be executed when they are called.

And actually we saw this happening already

in our pig game project. So there we had an init function,

which initialized our entire project but in order to actually initialize the game

the first time that the page loaded, we needed to call that function immediately

in our top-level code. And so that's what I mean here. But anyway, we can also see

what top-level code is in this example here. So this name variable declaration

is clearly top-level code right? And therefore it will be executed

in the global execution context. Next, we have two functions, one expression,

and one declaration. So these will also be declared, so that they can be called later.

But the code inside the functions, will only be executed when the functions are called. Okay, so we know that a global execution context is created for top-level code. But now what exactly is an execution context? Well, an execution context is an abstract concept. But I define it basically as an environment in which a piece of JavaScript is executed. It's like a box that stores all the necessary information for some code to be executed. Such as local variables or arguments passed into a function. So, JavaScript code always runs inside an execution context. And to make this a bit more intuitive let's imagine you order a pizza at a takeaway. So usually that pizza comes in a box right? And it might also come with some other stuff that is necessary for you to eat a pizza such as cutlery or a receipt, so that you can actually pay for the pizza before eating it. So, in this analogy, the pizza is the JavaScript code to be executed, and the box is of course the execution context for our pizza. And that's because eating the pizza happens inside the box which is then the environment for eating pizza. The box also contains cutlery and the receipt, which are necessary to eat a pizza or in other words, to execute the code, okay? I hope that made sense, and to made the concept of execution context a little bit more clear. Now, in any JavaScript project, no matter how large it is, there is only ever one global execution context. It's always there as the default context, and it's where top-level code will execute. And speaking of execute, now that we have an environment where the top-level code can be executed, it finally is executed. And there is not a lot to say about the execution itself. It's just the computer CPU processing the machine code that it received. Okay, and once this first code, so the top-level of code is finished, functions finally start to execute as well. And here is how that works. For each and every function call, and you execution context will be created containing all the information that is necessary to run exactly that function. And the same goes for methods, of course, because they're simply functions attached to objects remember? Anyway, all these execution contexts together, make up the call stack that I mentioned before. But more on that in a second. Now, when all functions are done executing, the engine will basically keep waiting for callback functions to arrive so that it can execute these. For example, a callback function associated with a click event. And remember, that it's the event loop who provides these new callback functions as we learned in the last lecture. All right.





Finally, each context also gets a special variable called the `this` keyword. And once more, there is a special lecture just about the `this` keyword later in the section.

Okay. Now, the content of the execution context, so variable environment, scope chain and `this` keyword is generated in a so-called creation phase. Which happens right before execution. And now just one final but very important detail that we need to keep in mind, is that execution contexts belonging to arrow functions, do not get their own arguments keyword, nor do they get the `this` keyword, okay? So, basically arrow functions don't have the arguments object and the `this` keyword. Instead, they can use the arguments object, and the `this` keyword from their closest regular function parent. And this is an extremely important detail to remember about arrow functions and we will come back to it later. So these are the things that are necessary to run each function as well as the code in the top-level.

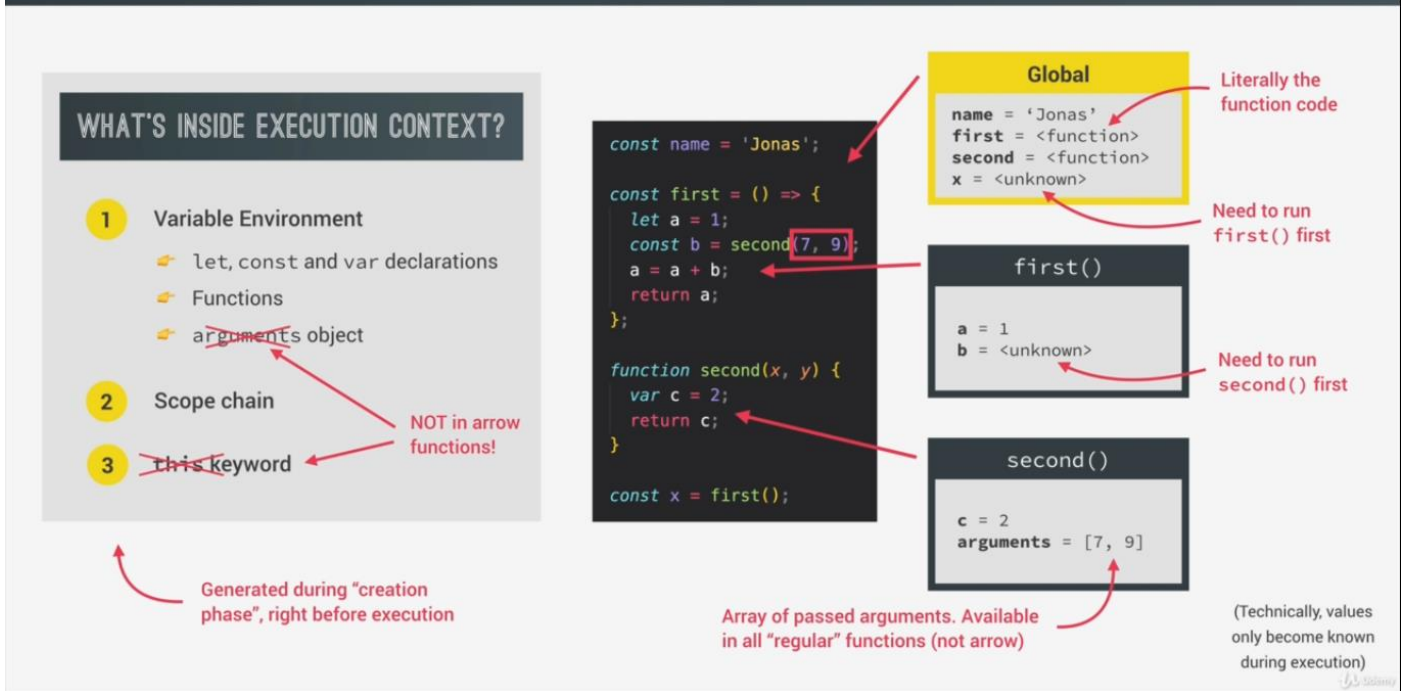
Now behind the scenes, it's actually even more complex but I think we're fine like this, aren't we? And now let's actually try to simulate the creation phase for this code example here. So, as you hopefully know, by now, we will get one global execution context and one for each function. So one for the first function, and one for the second function. In the global context, we have the name variable declaration, the first and second function declarations, as well as the `X` variable declaration.

For the functions, the variable environment will literally contain all the code of a particular function. Now the value of `X` is marked as unknown here, because this value is the result of the first function that we didn't run yet. But we will simulate this in the next slide. Now, technically none of these values actually become known during the creation phase, but only in the execution phase. So this is not 100% accurate here, but it's just to illustrate how these execution contexts work. Okay? So just keep that in mind. Anyway, now in the first function, we have the `a` variable set to 1 and the `b` variable which once again requires a function call in order to become known.

Finally, the variable environment of the second function, contains the `C` variable set to 2, and since this is a irregular function, so not an arrow function, it also has the arguments object. And this object is an array, which contains all the arguments that were passed into the function when it was called.

In this case, as you can see, that's 7 and 9. Quite simple right? Well, it's simple because this is an extremely small amount of code. But now imagine there are hundreds of execution contexts for hundreds of functions. How will the engine keep track of the order in which functions we're called? And how will it know where it currently is in the execution? Well, that's where the call stack finally comes in.

# EXECUTION CONTEXT IN DETAIL



And remember that the call stack, together with the memory heap, makes up the JavaScript engine itself.

But what actually is the call stack?

Well, it's basically a place where execution contexts get stacked on top of each other, in order to keep track of where we are in the programs execution.

So the execution context that is on top of the stack, is the one that is currently running.

And when it's finished running, it will be removed from the stack, and execution will go back to the previous execution context.

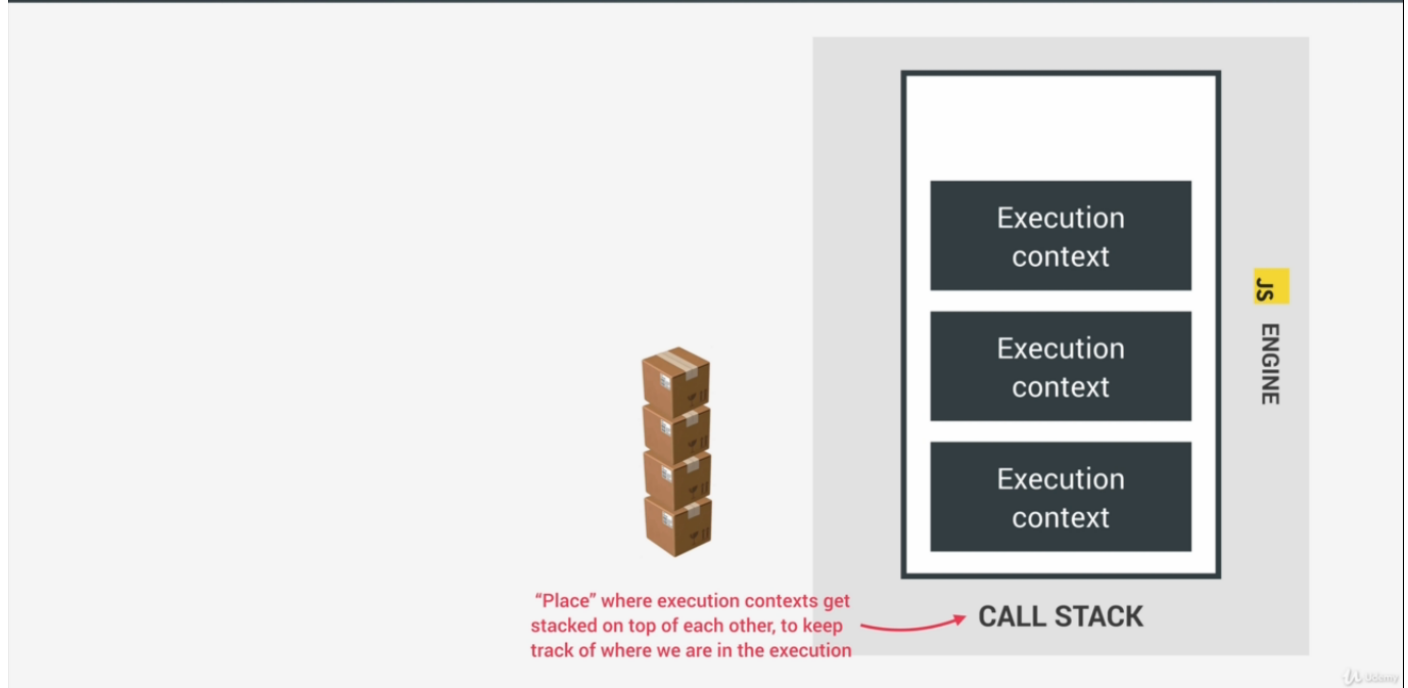
And using the analogy from before, it is as if you bought pizzas with some friends.

Each friend has a pizza box, and then you put the boxes on top of each other, forming a stack, in order to keep track, which pizza belongs to each friend.

Now, all this sounds a bit abstract, doesn't it?

And so to demonstrate how the call stack works,

# THE CALL STACK



So, once the code is compiled, top-level code will start execution.

And then as we learned in the beginning of the lecture, a global execution context will be created for the top-level of code, right? So this is where all the code outside of any function will be executed.

And what happens with this execution context?

That's right, it will be put in the call stack. And since this context is now at the top of the stack, it is the one where the code is currently being executed.

So, let's continue now with this execution. So here, there is a simple variable declaration. And then the first and the second functions are declared.

So nothing fancy, but that's just how normal top-level code gets executed.

But then, in the last line is where things start to get interesting.

Here, we declare the X variable, with the value that is gonna be returned from calling the first function. And so let's actually call that function.

Now what happens immediately when a function is called? Well, it gets its own execution context so that it can run the code that's inside its body. Perfect.

And what happens to the context? Well, again it is put in the call stack, on top of the current context, and so it's now the new current execution context.

Great.

# THE CALL STACK

Compiled code starts execution

```
const name = 'Jonas';  
  
const first = () => {  
  let a = 1;  
  const b = second(7, 9);  
  a = a + b;  
  return a;  
};  
  
function second(x, y) {  
  var c = 2;  
  return c;  
}  
  
const x = first();
```

Global

JS  
ENGINE

"Place" where execution contexts get stacked on top of each other, to keep track of where we are in the execution

CALL STACK

So, let's continue. So we have yet another simple variable declaration here, and this variable will of course be defined in the variable environment of the current execution context, and not in the global context, right? Then right in the next line, we have another function call. So, let's call that function and move there. And as you guessed a new execution context was created right away for this second function. And once more, it is pushed onto the call stack and becomes the new act of context. Now what's important to note here is that the execution of the first function has now been paused, okay? So again, we are running the second function now and in the meantime, no other function is being executed. The first function stopped at this point where the second function was called and will only continue as soon as this second function returns. And it has to work this way because remember, JavaScript has only one thread of execution. And so it can only do one thing at a time.

# THE CALL STACK

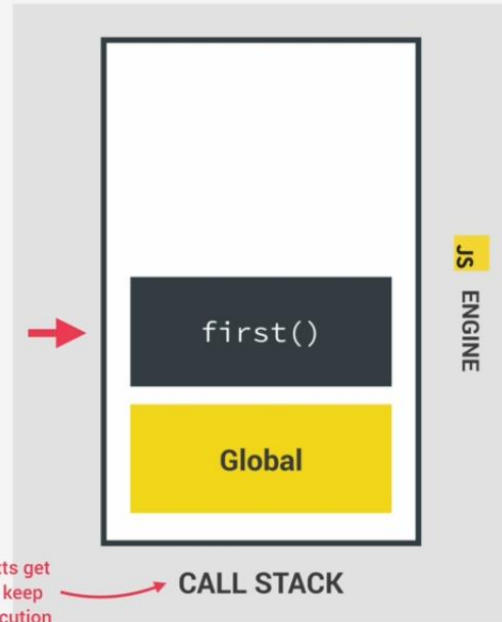
➡ Compiled code starts execution

```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second(7, 9);
  a = a + b;
  return a;
};

function second(x, y) {
  var c = 2;
  return c;
}

const x = first();
```



# THE CALL STACK

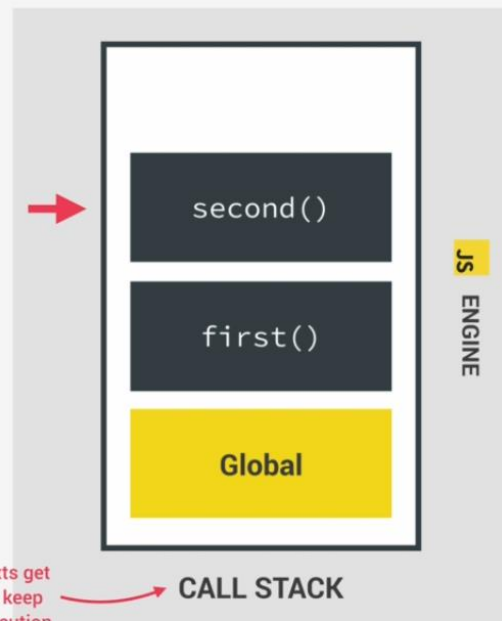
➡ Compiled code starts execution

```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second(7, 9);
  a = a + b;
  return a;
};

function second(x, y) {
  var c = 2;
  return c;
}

const x = first();
```

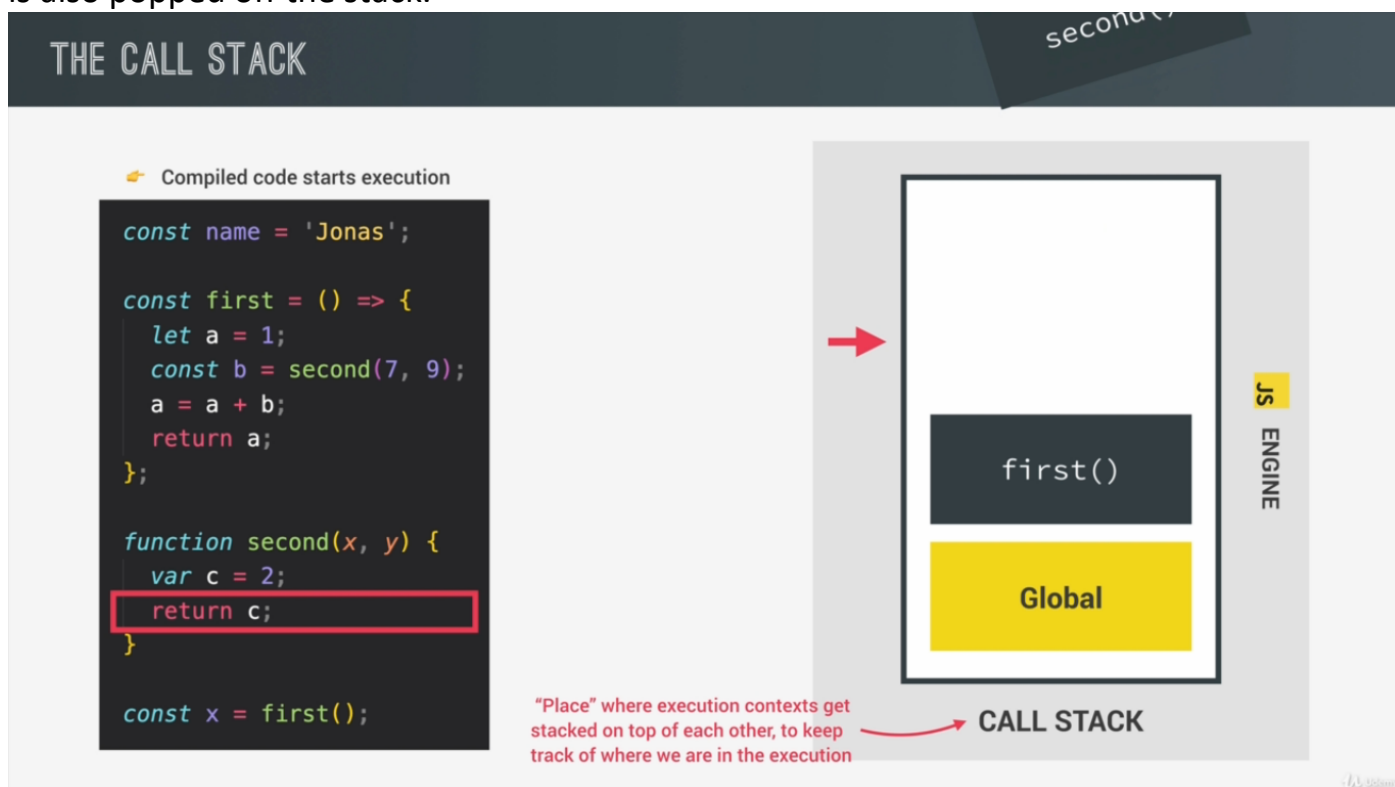


Okay? Never forget that. Now, moving to the next line, we have a return statement meaning that the function will finish its execution.

So, what does that mean for the call stack?

Well, it basically means that the function's execution context, will be popped off the stack and disappear from the computer's memory. At least that's what you need to know for now because actually the popped off execution context might keep living in memory.

But more about that later in the course. Anyway, what happens next, is that the previous execution context, will now be back to being the active execution context again. And so let's also go back to where we were before in the code. And I hope that by now, you start to see how the call stack really keeps track of the order of execution here. Without the call stack, how would the engine know which function was being executed before? It wouldn't know where to go back to, right? And that's the beauty of the call stack. It makes this process almost effortless. So I like to use the analogy of the call stack being like a map for the JavaScript engine. Because the call stack ensures that the order of execution never gets lost. Just like a map does, at least if you use it correctly. All right. So, we returned from the second function, or back in the first function where we have this calculation, and then finally this first function also returns. And so here the same as before happens. So the current execution context gets popped off the stack, and the previous context is now the current context where code is executed. In this case, we're back to the global execution context and the line of code where the first function was first called. So here, the return value is finally assigned to X and the execution is finished. Now the program will now actually stay in this state for forever until it is eventually really finished. And that only happens like when we close the browser tab, or the browser window. Only when the program is really finished like this, is when the global execution context is also popped off the stack.



# THE CALL STACK

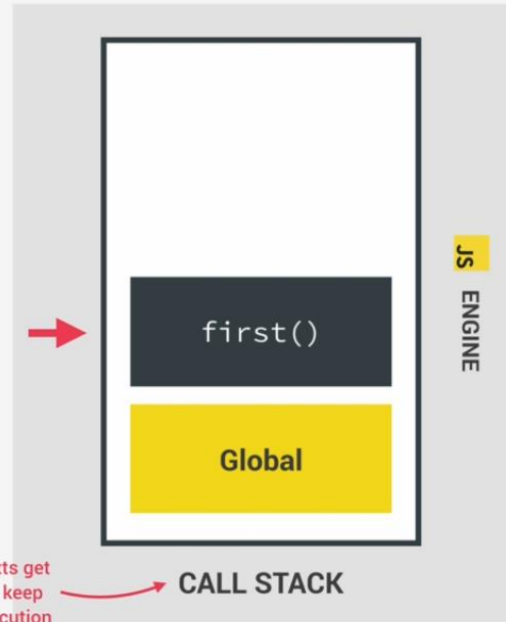
➡ Compiled code starts execution

```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second(7, 9);
  a = a + b;
  return a;
};

function second(x, y) {
  var c = 2;
  return c;
}

const x = first();
```



"Place" where execution contexts get stacked on top of each other, to keep track of where we are in the execution

Udemy

# THE CALL STACK

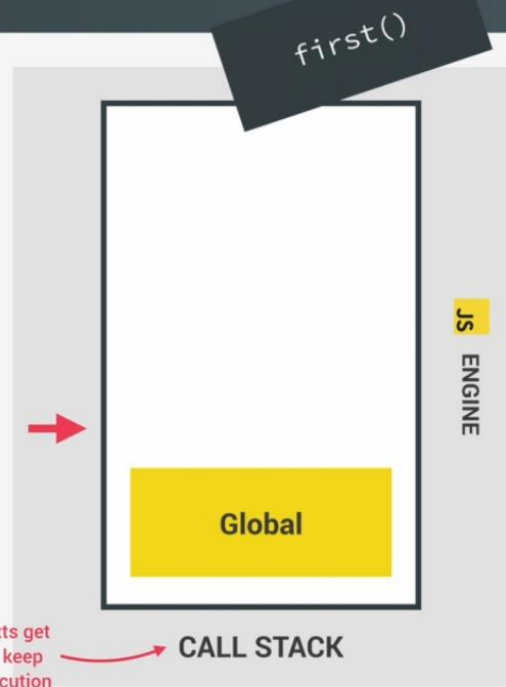
➡ Compiled code starts execution

```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second(7, 9);
  a = a + b;
  return a;
};

function second(x, y) {
  var c = 2;
  return c;
}

const x = first();
```



"Place" where execution contexts get stacked on top of each other, to keep track of where we are in the execution

Udemy

# THE CALL STACK

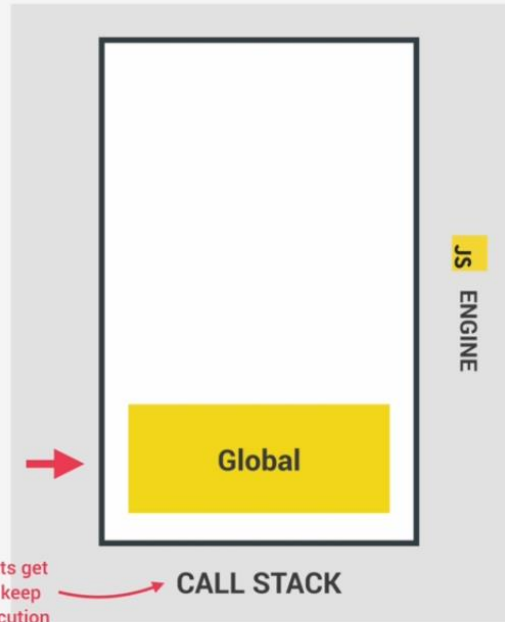
➡ Compiled code starts execution

```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second(7, 9);
  a = a + b;
  return a;
};

function second(x, y) {
  var c = 2;
  return c;
}

const x = first();
```



"Place" where execution contexts get stacked on top of each other, to keep track of where we are in the execution

# THE CALL STACK

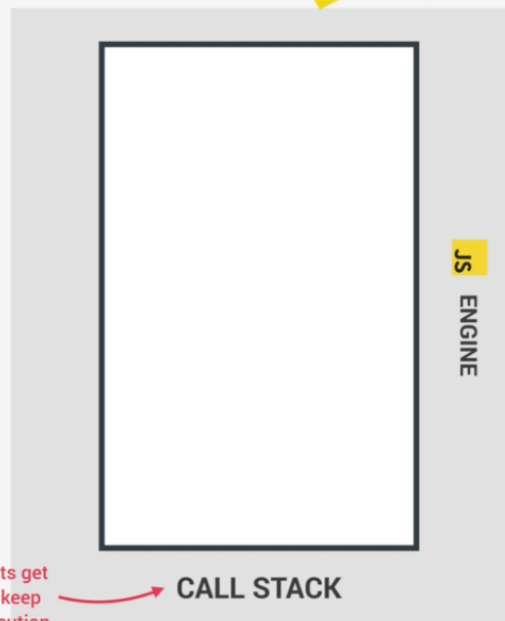
➡ Compiled code starts execution

```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second(7, 9);
  a = a + b;
  return a;
};

function second(x, y) {
  var c = 2;
  return c;
}

const x = first();
```



"Place" where execution contexts get stacked on top of each other, to keep track of where we are in the execution



## ❖ 8. How JavaScript Works Behind the Scenes → 6. Scope and The Scope Chain

a scope chain and a this keyword. So in this lecture, let's learn what scope and a scope chain are,

why they are so important and how they work.

And let's start by understanding

what scoping actually means,

and learn about some related concepts as well.

So scoping controls how our program's variables are organized and accessed by the JavaScript engine.

So basically scoping asks the question, where do variables live?

Or where can we access a certain variable and where not?

Now in JavaScript,

we have something called lexical scoping.

And lexical scoping means that the way variables are organized and accessed

is entirely controlled by the placement of functions and of blocks in the programs code.

For example, a function that is written inside another function has access to the variables of the parent function, okay?

So again, variable scoping is influenced by where exactly we write our functions and code blocks.

Okay, and now about scope itself. Scope is the space or environment in which a certain variable is declared, simple as that. And in the case of functions,

that's essentially the variable environment which is stored in the functions execution context. So if now you're asking yourself, what's the difference between scope

and variable environment? Then the answer is that for the case of functions,

it's basically the same. Now in JavaScript, we have the global scope,

function scope, and block scope. And we will talk about these in a second.

But first, let's also define what the scope of a variable is. So the scope of a variable is basically the entire region of our code, where a certain variable can be accessed. Now, some people use the word scope for all of this, but I like to define all these concepts that we have here in a clear way, because actually subtle differences.

For example, if you take a close look at it, scope is not the same as scope of a variable.

And so you should know about the subtle differences, right?

## SCOPING AND SCOPE IN JAVASCRIPT: CONCEPTS

### EXECUTION CONTEXT

- Variable environment
- Scope chain
- this keyword

### SCOPE CONCEPTS

- 👉 **Scoping:** How our program's variables are **organized** and **accessed**. "Where do variables live?" or "Where can we access a certain variable, and where not?";
- 👉 **Lexical scoping:** Scoping is controlled by **placement** of functions and blocks in the code;
- 👉 **Scope:** Space or environment in which a certain variable is **declared** (*variable environment in case of functions*). There is **global** scope, **function** scope, and **block** scope;
- 👉 **Scope of a variable:** Region of our code where a certain variable can be **accessed**.

let's now talk about the three different types of scope in JavaScript.

So that's global scope, function scope and block scope. And remember, scope is the place in our code where variables are declared. And when I say variables, the exact same thing is true for functions as well. Because in the end, functions are just values that are stored in variables.

**first, the global scope:** is once more for top level code. So this is for variables that are declared outside of any function or block. These variables will be accessible everywhere in our program, in all functions and all blocks. So really, everywhere.

**function scope:**

Next, each and every function creates a scope. And the variables declared inside that function scope are only accessible inside that function. This is also called a local scope opposed to the global scope. So local variables live in the function so to say. And outside of the function, the variables are then not accessible at all.

Again, this is technically the same as the functions variable environment, but we still need to give it the name of scope in this context, because blocks also creates scopes. Anyway, in this example here, the now variable is 2037 inside the cog H function. And therefore, we can use it in the function to do calculations. But outside of the function, as we try to log it to the console, we get a reference error. So JavaScript is trying to find the now variable in this global scope, so outside of the function, but it cannot find it. And so there is gonna be an error. And if you remember, or pick game project from the previous section, there is also the reason why we had to declare a couple of variables outside of the init function, remember that? So we had some variables declared in the init function, and then that gave us an error because other functions were trying to access these variables. But of course they were in the function scope. And so they were locally scoped, and so we couldn't access them outside of that function where they were declared. And here, it actually does not matter what kind of function we're using. So function declarations, function expressions and arrow functions all create their own scope.

**Block scope:** Now traditionally, only functions used to create scopes in JavaScript. But starting in ES6, blocks also creates scopes now. And with blocks, we mean everything that is between curly braces, such as the block of an if statement or a for loop. So just like with functions, variables declared inside a block are only accessible inside that block and not outside of it. Now, the big difference is that block scopes only apply to variables declared with let or const, okay? So again, only let and const variables are restricted to the block in which they were created. That's why we say that let and const variables are block scoped. So if I declared a variable using var in this block, then that variable would actually still be accessible outside of the block, and would be scoped to the current function or to the global scope. And so we say that var is function scoped. So in ES5 and before, we only had global scope and function scope. And that's why ES5 variables declared with var, only care about functions, but not about blocks. They simply ignore them. Finally, also starting in ES6, all functions are now also block scoped, at least in strict mode, which you should always be using anyway. And just like with let and const variables, this means that functions declared inside a block are only accessible inside that block, okay? And we will see examples of all that in the next video, when we're gonna go back to coding. So to recap, let and const variables as well as functions are block scoped. And if you already know other programming languages, block scoping is probably more in line with what you already know. Function scopes are weird for some beginners in the JavaScript world. And that's why block scopes were introduced in ES6. But now to understand all this a little bit better, let's actually look at a more real and detailed example and also learn about the scope chain.

# THE 3 TYPES OF SCOPE

## GLOBAL SCOPE

```
const me = 'Jonas';  
const job = 'teacher';  
const year = 1989;
```

- ✦ Outside of **any** function or block
- ✦ Variables declared in global scope are accessible **everywhere**

## FUNCTION SCOPE

```
function calcAge(birthYear) {  
  const now = 2037;  
  const age = now - birthYear;  
  return age;  
}  
  
console.log(now); // ReferenceError
```

- ✦ Variables are accessible only **inside function, NOT** outside
- ✦ Also called local scope

## BLOCK SCOPE (ES6)

```
if (year >= 1981 && year <= 1996) {  
  const millennial = true;  
  const food = 'Avocado toast';  
} ← Example: if block, for loop block, etc.  
  
console.log(millennial); // ReferenceError
```

- ✦ Variables are accessible only **inside block** (block scoped)
- ⚠ **HOWEVER**, this only applies to **let** and **const** variables!
- ✦ Functions are **also block scoped** (only in strict mode)

And here we have some code with different functions and blocks, and we're gonna take a look at the scopes that are in this code as well as build the scope chain. And of course, we start with the global scope. As you can see, the myName variable is the only variable declaration that we have in the global scope. Now, technically, the first function also counts as a variable that is present in the global scope, but I want to keep it simple here. And so I will only consider variable declarations and no functions, all right? Just keep in mind that whatever I'm explaining here for variables also works the same for functions. Anyway, inside the global scope, we have a scope for the first function because each function creates its own scope, remember? And what's in the scope? Well, it's to age variable that's declared right at the top of the function. Next inside the first scope, let's now consider the second function, which will also create its own scope containing the job variable set to teacher. So as you see, we have a nested structure of scopes with one scope inside the other. But now comes the actually interesting part.

Because here in the second function, we have this line of code where we need the myName variable and the age variable, which were both not declared inside the current scope. But we really need these variables here, because otherwise we can't create this string here, right? So how can this be fixed? How will the JavaScript engine know the values of these variables? Well, the secret is that every scope always has access to all the variables from all its outer scopes. So from all its parent scopes. In our example, this means that the second scope

can access the age variable from the scope of the first function. Of course, this also means that the first scope can access variables that are in the global scope, because that is the parent scope. As a consequence of this, the second scope will then also be able to access the myName variable from the global scope, because it has access to the variables from the first scope. And by the way, all this also applies to function arguments. But in this example, we just don't have any. And this is essentially how the scope chain works. In other words, if one scope needs to use a certain variable, but cannot find it in the current scope, it will look up in the scope chain and see if it can find a variable in one of the parent scopes. If it can, it will then use that variable. And if it can't, then there will be an error. And this process is called variable lookup. Now it's important to note that these variables are not copied from one scope to another, okay? Instead, scopes simply look up in the scope chain until they find a variable that they need and then they use it. What's also extremely important to note is that this does not work the other way around. A certain scope will never, ever have access to the variables of an inner scope. In this example, the first scope, for example, will never get access to the job variable that is stored in the second scope, okay? So again, one scope can only look up in a scope chain, but it cannot look down basically. So only parent scope can be used, but no child scopes. Anyway, with all this in place now, this line of code can be executed and print to the console. Jonas is a 30 year old teacher, even though the myName and age variables were not defined in the current scope. All the engine did was to get them from the scope chain. And as you might be noticing, we have actually already done this before in our own code. We just didn't really understand what was going on and how it all worked. But now we do know how it works. Amazing, right?

## THE SCOPE CHAIN

```

const myName = 'Jonas';

function first() {
  const age = 30;

  if (age >= 30) { // true
    const decade = 3;
    var millenial = true;
  }

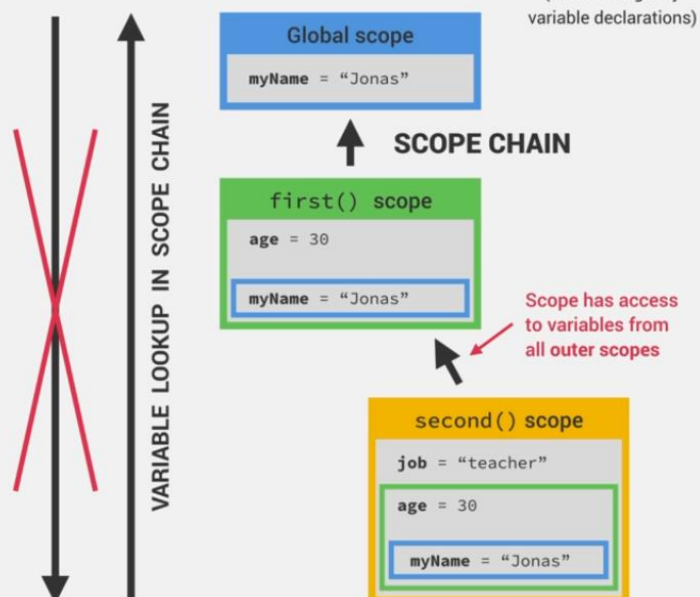
  function second() {
    const job = 'teacher';
    console.log(`myName is a $age -old ${job}`);
    // Jonas is a 30-old teacher
  }

  second();
}

first();

```

Variables not in current scope



Anyway, we still have one more scope left here, and that's the one created by this block here. Remember that starting with ES6, not only functions create scopes, but also blocks. However, these scopes only work for the ES6 variable types. So for let and const variables. That's why the only variable that's in the scope is the decade variable. The millennial variable isn't declared with const or let, and therefore it is not scoped to just this block. Instead, the millennial variable is actually part of the first function scope. So again, for a variable declared with var, block scopes don't apply at all. They are functions scoped, not block scoped. Let and const on the other hand are in fact blocks scoped, okay? This is one of the fundamental things that you need to keep in mind about let, const and var, and about scoping in general. So if you're taking notes and I hope you are taking lots of notes, then this must definitely be in there. Now about a scope chain, if the millennial variable is in the first function scope, then of course the second function scope also has access to it, even if it doesn't really need that variable. Also the scope chain does of course, apply to block scopes as well. And therefore in or if block scope, we get access to all the variables from all its outer scopes. So from the first function scope, and of course from the global scope. That's why I said in the last slide that variables in a global scope are accessible from everywhere. They are, because they are always at the top of the scope chain. In fact, we call variables in the global scope, global variables, very creative, right? But we actually use this term a lot in JavaScript. Now it's important to understand that our purple blocks scope does not get access to any variables from the yellow second function scope. And the same, the other way around. And why is that? Well it's because of lexical scoping as we learned in the last slide. So the way that we can access variables depends on where the scope is placed, so where it is written in the code. In this case, none of these two scopes is written inside of one another. They're both child scopes of the first function. We could even say that they are a sibling scopes. And so by the rules of lexical scoping, they cannot have access to each others variables, simply because one is not written inside the other one. We can also say that the scope chain only works upwards, not sideways.

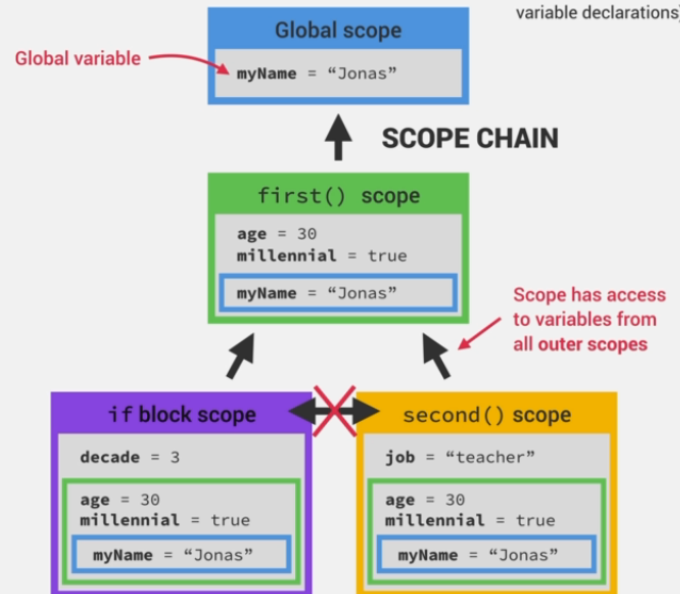
# THE SCOPE CHAIN

(Considering only variable declarations)

```
const myName = 'Jonas';

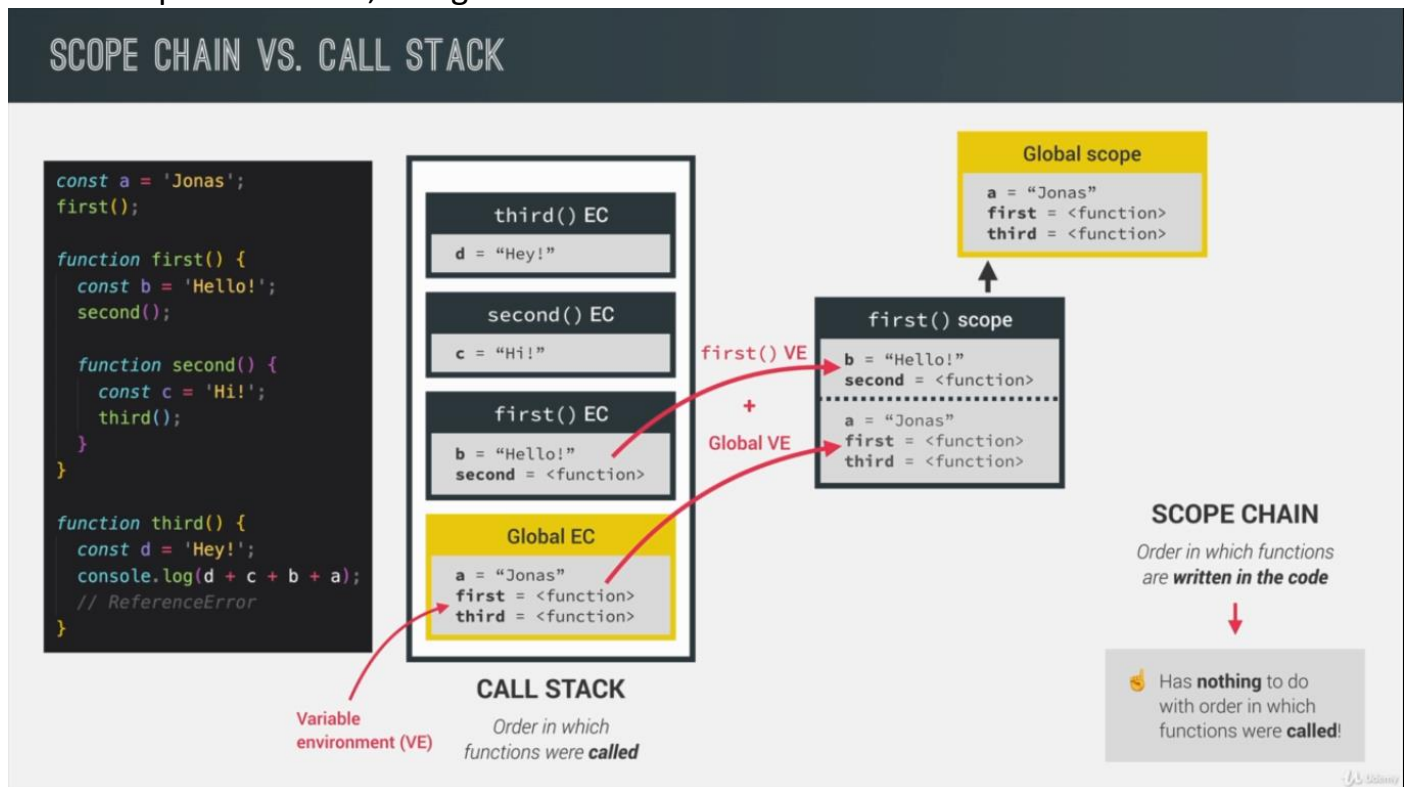
function first() {
  const age = 30;
  if (age >= 30) { // true
    const decade = 3;
    var millennial = true;
  }
  function second() {
    const job = 'teacher';
    console.log(`myName is a $age-old ${job}`);
    // Jonas is a 30-old teacher
  }
  second();
}

first();
```



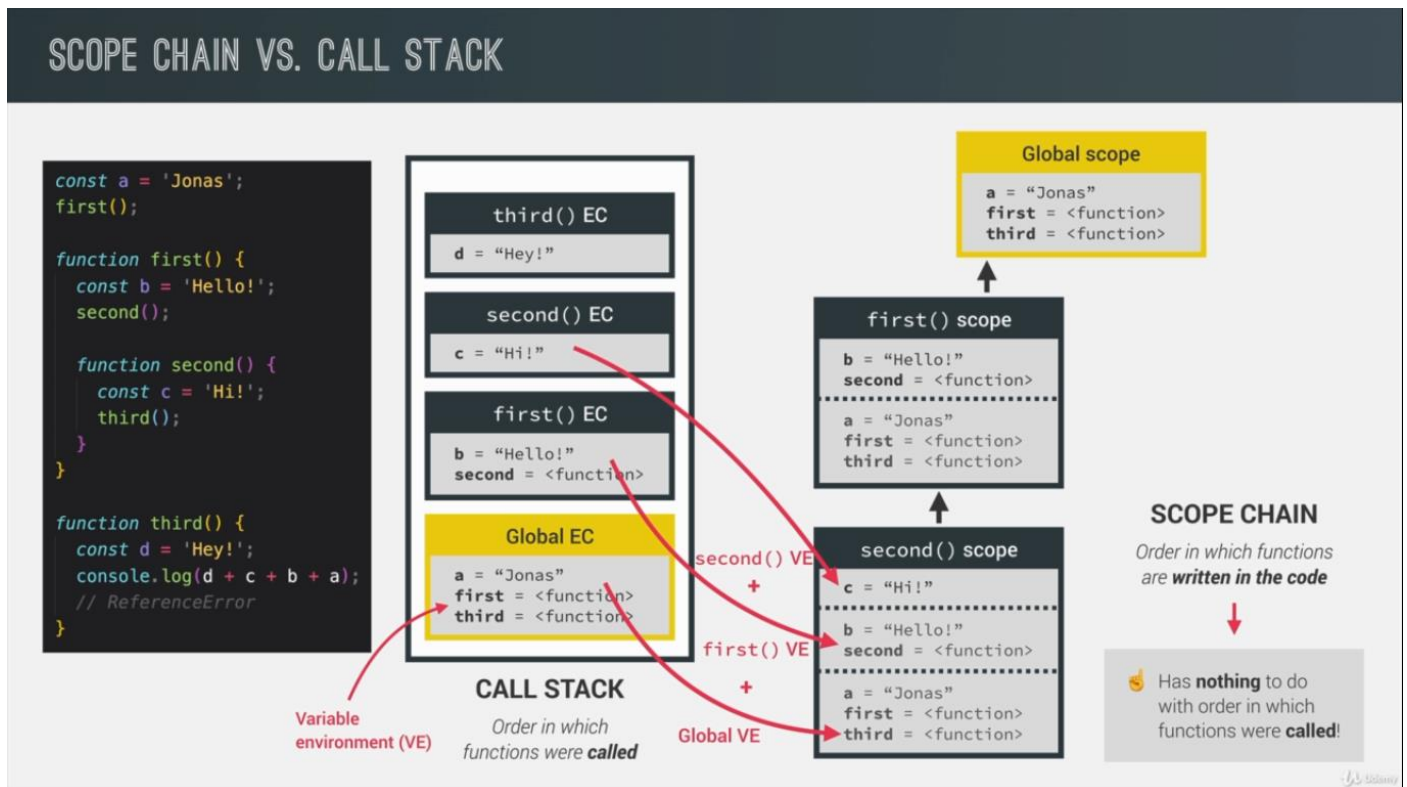
But for now, though, there is one more thing that we need to talk about, which is the difference between the scope chain and to call stack. I get a lot of questions about this all the time. And so I decided to talk about how the call stack, execution context, variable environments and scope are all related to one another. So before we move on to the next video. And once more, let's look at some more code here. So we have three functions called first, second and third, in order to make this easier to understand. We start by calling the first function, which then calls the second function, which in turn calls the third function. So from what we learned before, the call stack for this example will look like this, right? One execution context for each function in the exact order in which they were called. They also included the variable environment of each execution context. For now, all this has nothing to do with scopes or the scope chain, all right? All I'm doing is creating one execution context for each function call and filling it with the variables of that function. And you can pause the video here for a moment to understand the content of each variable environment. Okay, and now that you did that and we have all these variable environments in place, we can actually start building the scope chain. As always, we're gonna start with the global scope. And the variables available in the global scope are exactly the ones stored in the variable environment of the global execution context. And given everything we've learned so far,

that makes sense, right? And note that in this example, I am actually including functions in each scope unlike we did in the previous slide. Now in the global scope, we also call the first function, which is the reason why we have an execution context for it in the call stack. And this function of course, also gets its own scope, which contains all the variables that are declared inside of the function. And once again, this is exactly the same as the variable environment of the functions execution context. However, that's not all because now we already know about the scope chain. So the first scope also gets access to all the variables from its parent scope, thanks to the scope chain. Now, as we already know, the scope chain is all about the order in which functions are written in the code. But what's really important to note here is that the scope chain has nothing to do with the order in which functions were called. Or in other words, the scope chain has nothing to do with the order of the execution contexts in the call stack. The scope chain does get the variable environments from the execution context as shown by the red arrows here, but that's it. The order of function calls is not relevant to the scope chain at all, all right?





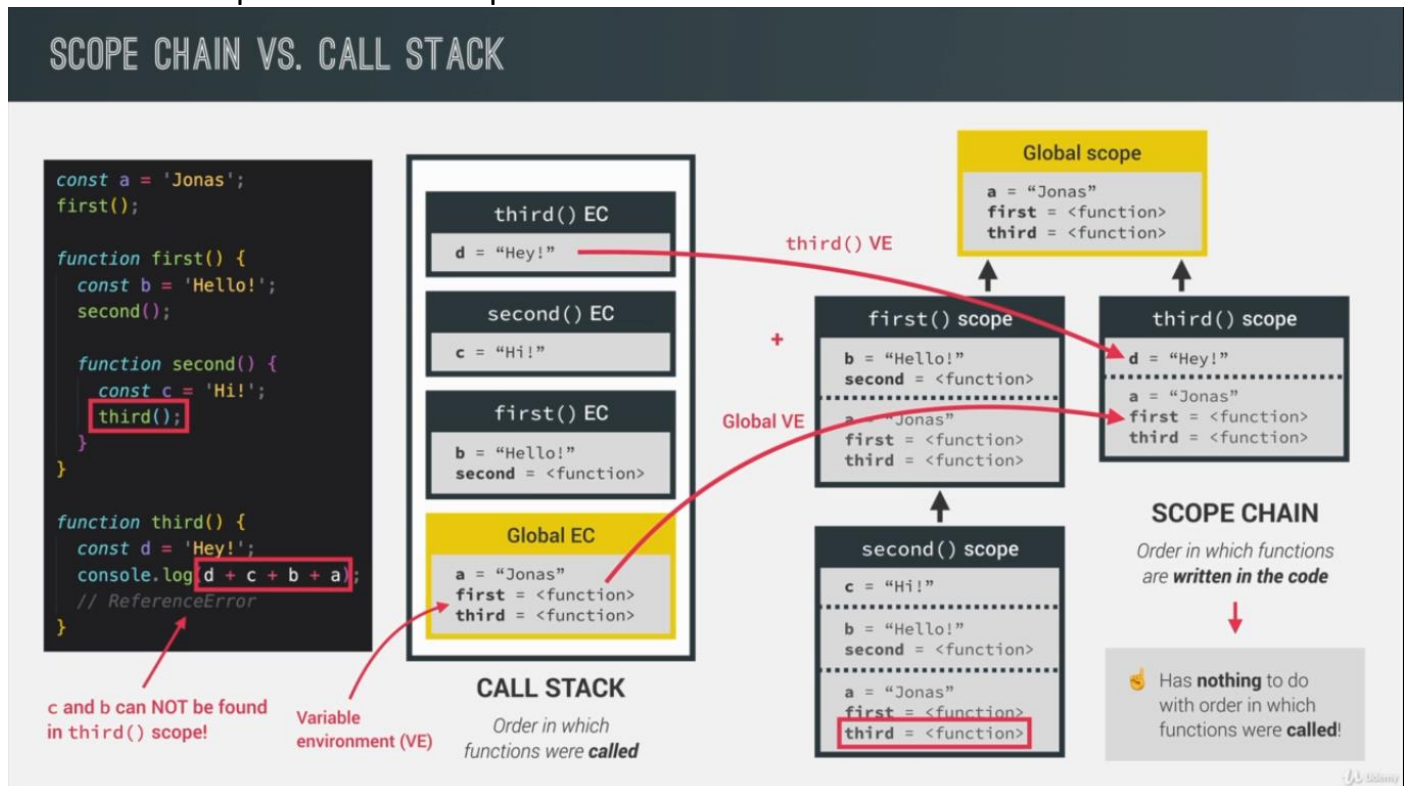
Now, moving on to the second function now, once again, its scope is equal to its variable environment. Also it's lexically written within the first function. And so of course, it will have access to all its parent scopes as well. So we can say that the scope chain in a certain scope is equal to adding together all the variable environments of all the parent scopes. And so this is our scope, and the scope chain are built in the JavaScript engine behind the scenes.



Now in the second function, we try to call the third function. But why does that work? Well, it works because the third function is in the scope chain of the second function scope as we can see here in our scope chain diagram. It's a function in the global scope or a global function, and therefore it's accessible everywhere. Of course, this will create a new scope along with the scope chain as we already know. Great, so what happens in this third function? Well, we're trying to act as variables B, C, D and A here. D is no problem because it's right there in the third function scope. So that one is easy. Then variable C is not in a local scope and so JavaScript needs to do a variable lookup. So it looks up in a scope chain looking for variable C, but it's not there. And of course it isn't, because C is defined in the second function, and there is just no way in which the third function can access variables

defined in the second function. And that is true, even though it was the second function who called the third. And so here is even more proof that the order in which functions are called does not affect the scope chain at all.

And so here as a result, we get the reference error because both C and B cannot be found in the third scope nor in the scope chain.



## SUMMARY 🤖

- 👉 Scoping asks the question “Where do variables live?” or “Where can we access a certain variable, and where not?”;
- 👉 There are 3 types of scope in JavaScript: the global scope, scopes defined by functions, and scopes defined by blocks;
- 👉 Only `let` and `const` variables are block-scoped. Variables declared with `var` end up in the closest function scope;
- 👉 In JavaScript, we have lexical scoping, so the rules of where we can access variables are based on exactly where in the code functions and blocks are written;
- 👉 Every scope always has access to all the variables from all its outer scopes. This is the scope chain!
- 👉 When a variable is not in the current scope, the engine looks up in the scope chain until it finds the variable it’s looking for. This is called variable lookup;
- 👉 The scope chain is a one-way street: a scope will never, ever have access to the variables of an inner scope;
- 👉 The scope chain in a certain scope is equal to adding together all the variable environments of the all parent scopes;
- 👉 The scope chain has nothing to do with the order in which functions were called. It does not affect the scope chain at all!



### ❖ 8. How JavaScript Works Behind the Scenes → 7. Scoping in Practice Just review the video

### ❖ 8. How JavaScript Works Behind the Scenes → 8. Variable Environment Hoisting and The TDZ

is a very misunderstood concept in JavaScript and that is hoisting. So we learned that an execution context always contains three parts. A variable environment, the scope chain in the current context, and the global object. We already learned about the scope chain and so now it's time to take a closer look at the variable environment and in particular at how variables are actually created in JavaScript. So in JavaScript we have a mechanism called hoisting. And hoisting basically make some types of variables accessible, or let's say usable in the code before they are actually declared in the code. Now, many people simply define hoisting by saying that variables are magically lifted or moved to the top of their scope for example, to the top of a function. And that is actually what hoisting looks like on the surface. But behind the scenes that's in fact not what happens. Instead, behind the scenes the code is basically scanned for variable declarations before it is executed. So this happens during the so-called creation phase of the execution context that we talked about before. Then for each variable that is found in the code, a new property is created in a variable environment object. And that's how hoisting really works. Now, hoisting does not work the same for all variable types. And so let's analyze the way hoisting works for function declarations, variables defined with var variables defined with let or const and function expressions, and also arrow functions. So function declarations are actually hoisted and the initial value in the variable environment is set to the actual function. So in practice, what this means is that we can use function declarations before they are actually declared in the code, again, because they are stored in the variable environment object, even before the code starts executing. And I actually told you before that function declarations work this way, but now you understand why. It is because of hoisting. And just to make this table a bit more complete so that it serves as a nice overview for you, I also show here that function declarations are block scoped as we learned before. Just keep in mind that this is only true for strict mode. So if you're using a sloppy mode, which you shouldn't, then functions are functioned sculpt. Next, variables declared with var are also hoisted, but hoisting works in a different way here. So unlike functions, when we try to access a var variable before it's declared in a code, we don't get the declared value but we get undefined. And this is a really weird behavior for beginners. You might expect that you simply get an error when using a variable before declaring it or to get the actual value. But not to get undefined because getting undefined is just really weird and it's not really useful either, right? And actually this behavior is a common source of bugs in JavaScript. So this is one of the main reasons why in modern JavaScript we almost never use var. Now on the other hand, let and const variables are not hoisted. I mean, technically they are actually hoisted

but their value is basically set to an initialized. So there is no value to work with at all. And so in practice, it is as if hoisting was not happening at all.

Instead, we say that these variables are placed in a so-called Temporal Dead Zone or TDZ which makes it so that we can't access the variables between the beginning of the scope and to place where the variables are declared.

So as a consequence, if we attempt to use a let or const variable before it's declared, we get an error. Also keep in mind that let and const are block scoped. So they exist only in the block in which they were created.

And all these factors together is basically the reason why let and const were first introduced into the language, and why we use them now instead of var in modern JavaScript. Okay. But now what about function expressions and arrow functions? How does hoisting work for this? Well, it depends if they were created using var or const or let. Because keep in mind that these functions are simply variables. And so they behave the exact same way as variables in regard to hoisting. This means that a function expression or arrow function created with var is hoisted to undefined. But if created with let or const, it's not usable before it's declared in a code because of the Temporal Dead Zone so again, just like normal variables, right?

And this is actually the reason why I told you earlier that we cannot use function expressions before we write them in the code, unlike function declarations.

## HOISTING IN JAVASCRIPT

👉 **Hoisting:** Makes some types of variables accessible/usable in the code before they are actually declared. "Variables lifted to the top of their scope".

↓ **BEHIND THE SCENES**

Before execution, code is scanned for variable declarations, and for each variable, a new property is created in the **variable environment object**.

### EXECUTION CONTEXT

- 👉 Variable environment
- ✅ Scope chain
- 👉 this keyword

	HOISTED? 👉	INITIAL VALUE 👉	SCOPE 👉
function declarations	✅ YES	Actual function	Block <span style="color: red;">↖ In strict mode. Otherwise: function!</span>
var variables	✅ YES	undefined	Function
let and const variables	❌ NO	<uninitialized>, TDZ	Block
function expressions and arrows	👉 Depends if using var or let/const		Temporal Dead Zone

*Technically, yes. But not in practice* (points to 'NO' for let and const)

let's take a more detailed look at this mysterious Temporal Dead Zone. So in this example code we're gonna look at the job variable. It is a const so it's scoped only to this if block and it's gonna be accessible starting from the line where it's defined.

Why? Well, because there is this Temporal Dead Zone for the job variable. It's basically the region of the scope in which the variable is defined, but can't be used in any way. So it is as if the variable didn't even exist.

Now, if we still tried to access the variable while in the TDZ like we actually do in the first line of this if block, then we get a reference error telling us that we can't access job before initialization. So exactly as we learned in the last slide, right? However, if we tried to access a variable that was actually never even created, like in the last line here where we want to log x, then we get a different error message saying that x is not defined at all. What this means is that job is in fact in the Temporal Dead Zone where it is still initialized, but the engine knows that it will eventually be initialized because it already read the code before and set the job variable in the variable environment to uninitialized.

Then when the execution reaches the line where the variable is declared, it is removed from the Temporal Dead Zone and it's then safe to use.

So to recap, basically each and every let and const variable get their own Temporal Dead Zone that starts at the beginning of the scope until the line where it is defined. And the variable is only safe to use after the TDZ, so the Temporal Dead Zone. Alright, now what is actually the need for JavaScript to have a Temporal Dead Zone? Well, the main reason that the TDZ was introduced in ES6 is that the behavior I described before makes it way easier to avoid and catch errors. Because using a variable that is set to undefined before it's actually declared can cause serious bugs which might be hard to find. And I will show you a small example in the next lecture. So accessing variables before declaration is bad practice and should be avoided. And the best way to avoid it is by simply getting an error when we attempt to do so. And that's exactly what a Temporal Dead Zone does. A second and smaller reason why the TDZ exists is to make const variables actually work the way they are supposed to. So as you know, we can't reassign const variables. So it will not be possible to set them to undefined first and then assign their real value later. Const should never be reassigned. And so it's only assigned when execution

actually reaches the declaration. And that makes it impossible to use the variable before. Okay? Makes sense? Now, if hoisting creates so many problems, why does it exist in the first place? I get this question all the time. And so let's quickly talk about that here. So the creator of JavaScript basically implemented hoisting so that we can use function declarations before we use them. Because this is essential for some programming techniques, such as mutual recursion. Some people also think that it makes code a lot more readable. Now, the fact that it also works for var declarations is because that was the only way hoisting could be implemented at the time. So the hoisting of var variables is basically just a byproduct of hoisting functions. And it probably seemed like a good idea to simply set variables to undefined, which in hindsight is not really that great. But we need to remember that JavaScript was never intended to become the huge programming language that it is today. Also, we can't remove this feature from the language now. And so we just use let and const to work around this.

## TEMPORAL DEAD ZONE, LET AND CONST

```
const myName = 'Jonas';

if (myName === 'Jonas') {
  console.log(`Jonas is a ${job}`);
  const age = 2037 - 1989;
  console.log(age);
  const job = 'teacher';
  console.log(x);
}
```

TEMPORAL DEAD ZONE FOR `job` VARIABLE

👉 Different kinds of error messages:

ReferenceError: Cannot access 'job' before initialization

ReferenceError: x is not defined

### WHY HOISTING?

- 👉 Using functions before actual declaration;
- 👉 var hoisting is just a byproduct.

### WHY TDZ?

- 👉 Makes it easier to avoid and catch errors: accessing variables before declaration is bad practice and should be avoided;
- 👉 Makes const variables actually work

❖ 8. How JavaScript Works Behind the Scenes → 9. Hoisting and TDZ in Practice



❖ 8. How JavaScript Works Behind the Scenes → 10. The this Keyword

the `this` keyword or `this` variable is basically a special variable that is created for every execution context and therefore any function. And that sounds very abstract but we will see what that means in a second.

For now, what's very important to understand is that the value of the `this` keyword is not static.

So it's not always the same. It depends on how the function is actually called. And its value is only assigned when the function is actually called. So it's very different from a normal value,

In this regard, if we set, for example, `X` to five, then `X` will always just be five. But the `this` keyword again,

depends on the way in which a function is called. But what does that actually mean? In fact, we learned before that it's one of the three components of any execution context, along with the variable environment and scope chain that we already learned about before. Now, in general terms, the `this` keyword, will always take the value of the owner of the function in which, the `this` keyword is used. We also say, that it points to the owner of that function. Well, let's analyze four different ways in which functions can be called. And the first way to call a function is as a method. So as a function attached to an object. So when we call a method, the `this` keyword inside that method will simply point to the object on which the method is called, or in other words, it points to the object that is calling the method.

And so let's illustrate this with a simple example. So the method here is the `calcAge` method, again, because it's a function attached to the `Jonas` object.

in the last line here, we then call the method and as you see inside the method, we used the `this` keyword. Now, according to what we just learned, what should be the value of the `this` keyword here?

And that's right, it should be `Jonas`, because that is the object that is calling the method down there

in the last line. Isn't it? And so then, on `Jonas`, we can of course access all the properties that it has.

And so, `this.year` will become 1989, because that's `Jonas.year` as well. So in this case, writing `'Jonas.year'`

would have the exact same effect as `'this.year'`. But doing it this way is a way better solution,

for many reasons that we will get into throughout the course. And we will play around with this example a bit more in the next video. But anyway, another way we call functions



is by simply calling them as normal functions. So not as a method and so not attached to any object.

In this case, the `this` keyword, will simply be undefined. However, that is only valid for strict mode.

So if you're not in strict mode, `this` will actually point to the global object, which in case of the browser

is the window object. And that can be very problematic and so, this is yet another very good reason

to always use strict mode. Next, we have arrow functions and while arrow functions

are not exactly a way of calling functions. It's an important kind of function that we need to consider,

because, remember, arrow functions do not get their own 'this keyword'. Instead, if you use 'the this variable' in an arrow function, it will simply be the `this` keyword of the surrounding function.

So of the parent function and in technical terms, this is called the 'lexical this keyword,' because it simply gets picked up from the outer lexical scope of the arrow function. So never ever forget this very important property of arrow functions. Believe me that actually I run into one or two bugs because of this behavior. So it's really important not to forget that arrow functions do not get their own 'this keyword.' Okay. And finally, if a function is called as an event listener, then the `this` keyword will always point to the DOM element that the handler function is attached to. Very straightforward right?

Now, the `this` keyword is usually a big source of confusion for beginners. But if you know these rules,

then it shall become a lot simpler and to make it even simpler. It's also important to know what the `this` keyword is not. So `this` will never point to the function in which we are using it. Also, the `this` keyword will never point to the variable environment of the function. And these are two pretty common misconceptions and so that's why I'm addressing them here. Okay? So again, the rules that I showed you here is all that you need to know. All right. And now just for the sake of completion, there are actually other ways in which we can call a function, for example, using the `new` keyword or the `call` apply

and `bind` methods, but we don't know yet what any of these are. And so I will talk about how the `this` keyword works with them, when the time comes. Anyway, that's it for this lecture. So let's now use `this` in practice to make it crystal clear.

# HOW THE THIS KEYWORD WORKS

- 👉 **this keyword/variable:** Special variable that is created for every execution context (every function). Takes the value of (points to) the "owner" of the function in which the `this` keyword is used.
- 👉 `this` is **NOT** static. It depends on **how** the function is called, and its value is only assigned when the function is **actually called**.

## EXECUTION CONTEXT

- ✅ Variable environment
- ✅ Scope chain
- 👉 `this` keyword

Method 👉 `this = <Object that is calling the method>`

Simple function call 👉 `this = undefined` In strict mode! Otherwise: window (in the browser)

Arrow functions 👉 `this = <this of surrounding function (lexical this)>` Don't get own this

Event listener 👉 `this = <DOM element that the handler is attached to>`

`new, call, apply, bind` 👉 `<Later in the course...>`

👉 `this` does **NOT** point to the function itself, and also **NOT** the its variable environment!

## Method example:

```
const jonas = {
  name: 'Jonas',
  year: 1989,
  calcAge: function() {
    return 2037 - this.year
  }
};
jonas.calcAge(); // 48
```

`calcAge` is method      `jonas`      `1989`

Way better than using `jonas.year!`

❖ **8. How JavaScript Works Behind the Scenes → 11. The this Keyword in Practice**

of how the `this` keyword is defined in action. And let's start by taking a look at the `this` keyword outside of any function whatsoever. So just outside here in the global scope. And so we get that the `this` keyword in the global scope is simply the window object. So that's the global object that we just saw before. Remember, and now let's do the same but inside of just a regular function call so we're going to use our old friend called `H` here. So let's look to the console, just the usual result so that we're actually doing something here. But then what we're really interested in is taking a look at `this`, the `this` keyword. So let's now call this function. And so let's see. And now the result of taking a look so at the logging, the `this` keyword is `undefined`. Okay. And so what that means is that insight is just regular function. Call here. The `this` keyword will be `undefined` and that's because we are in strict mode. Remember that in sloppy mode, it would be also the global object. So in this case, the window object but you already know that you always should use strict mode. And so then in a regular function call like this `D` the `this` keyword will simply point to `undefined` and with a regular function call `I` simply mean a call of `function` without the function being attached to any object. So without having an owner

1:

```
'use strict';
```

```
const calcAge = function (birthYear) {
```

```
console.log(2037 - birthYear); // 46
console.log(this); // undefined
};
calcAge(1991);
```

2:

```
'use strict';
const calcAgeArrow = birthYear => {
  console.log(2037 - birthYear); // 57
  console.log(this); // window
};
calcAgeArrow(1980);
```

Why is the this keyword undefined in function #1, but window in function #2 ?

Well, it is because the arrow function does not get its own this keyword. So instead the arrow function simply uses the lexical this keyword, which means that it uses the this keyword of its parent function or of its parents scope.

in this case, what is the lexical, this keyword? So what is the this keyword in the parent's scope of this function? Well, it is window because window is the this keyword here in the global scope.

**let's try to use the this keyword inside of a method.**

```
'use strict';

const jonas = {
  year: 1991,
  calcAge: function () {
    console.log(this); // jonas object
  },
};
jonas.calcAge();
```

on just to one final example here in this lecture. So I keep saying that the `disc` keyword will point

to the object that is calling the method, right? And that means that the `disc` keyword will not simply point at the object in which we wrote the method. So here we wrote the `caulk H` method inside

of the `Jonas` object. And so we might think that `disc` is the reason why the `disc` keyword points

to `Jonas`, but that is not true. The reason that the `disc` keyword will point to `Jonas` in this case is

because `Jonah's` was the object calling `debt` method and that's a subtle, but very important difference.

And let me know, show you why that is. So let's create a new object which I'm going to call a `Matilda`

and let's create a first name. We're actually, we don't need a name. Let's just say, year 2017. And that's it. That's all we need in this object. Now, remember that a function is just a value right? And therefore we can do this. We can say `Matilda` thought `caulk eight` should be equal

to `Jonas` that `caulk H` so basically here we simply copy the `caulk H` method from `Jonah's` to `Matilda`.

But now when we take a look at `Matilda` here then we see that the `caulk H` function is also in here.

Alright. And of course it's still also in `Jonah's`, but now we copied it from one object

to the other, and this is called to method borrowing. So we basically borrowed here the method

from one object to the other. And so we don't have to write it in a duplicate way.

So now let's say `Matilda` dot `caulk H` and what do you think the `disc` keyword will look like now?

And let's get rid of some of these before, so that they don't clutter or output here. And you can of course take them back if you want to see them in your own coat. But now let me save this and see what the result of this `cog age` will be.

So the `age` of `Matilda` gave us the correct result

of 20, which is exactly 37 minus 27. And so this means that in this method call here to this keyword does not actually point to Matilda. And we can also see that here. So here, as we looked to this keyword we see that it's actually the Matilda object because of this birth here. And so this proves exactly what I was just telling you before which is the fact that the `disc` keyword always points to the object that is calling the method. And so here we are calling the method on Matilda right? And so therefore the `disc` keyword will point to Matilda, which was the object, which called the method.

So even though the method is written here inside of the Jonas object the `disc` keyword will still point to Matilda. If it is Matilda, who calls the method. Okay. And so that's really important to understand.

And it's the reason why I said in the last video that the `disc` keyword is really dynamic.

It's not static and it depends on how the function is called.

```
'use strict';

const jonas = {
  year: 1991,
  calcAge: function () {
    console.log(this);
    console.log(2037 - this.year);
  },
};
jonas.calcAge(); // jonas object ---- 46

const matilda = {
  year: 2017,
};

matilda.calcAge = jonas.calcAge;
matilda.calcAge(); // matilda object ----- 20
```

Uncaught TypeError: Cannot read properties of undefined (reading 'year')

```
const f = jonas.calcAge;
f(); //Uncaught TypeError: Cannot read properties of undefined (reading 'year')
```

On Paper ... to 9.3

## ❖ 9. Data Structures, Modern Operators and Strings → 3. Destructuring arrays

Destructuring is an ES6 feature

+ A way of unpacking values from an array or an object into separate variables  
+ is to break a complex data structure down into a smaller data structure like a variable

```
const arr = [2,3,4];  
const [a,b,c] = arr
```

\*whenever see “[ ]” left side of equal sign “=”, this is **destructuring**

**\*to skip items we can simply leave hole “,”**

```
const restaurant = {  
  name: 'Classico Italiano',  
  categories: ['Italian', 'Pizzeria', 'Vegetarian', 'Organic']  
}
```

```
let [main, , secondary] = restaurant.categories;  
console.log(main, secondary);
```

\*another usage of destructuring is Switching variables

```
// Switching variables  
// const temp = main;  
// main = secondary;  
// secondary = temp;
```

```
[main, secondary] = [secondary, main];
```

\*another usage of destructuring is Receive multiple return values from a function

```
const restaurant = {  
  starterMenu: ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad'],  
  mainMenu: ['Pizza', 'Pasta', 'Risotto'],
```

```
// ES6 enhanced object literals  
openingHours,
```

```

order(starterIndex, mainIndex) {
  return [this.starterMenu[starterIndex], this.mainMenu[mainIndex]];
},

// Receive 2 return values from a function
//const [starter, mainCourse] = restaurant.order(2, 0);

// Nested destructuring
const nested = [2, 4, [5, 6]];
// const [i, , j] = nested;
const [i, , [j, k]] = nested;
console.log(i, j, k);

// Default values
const [p = 1, q = 1, r = 1] = [8, 9];
console.log(p, q, r);

```

❖ 9. Data Structures, Modern Operators and Strings → 4. Destructuring Objects

```

// sample object:
const restaurant = {
  name: 'Classico Italiano',
  location: 'Via Angelo Tavanti 23, Firenze, Italy',
  categories: ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'],
  starterMenu: ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad'],
  mainMenu: ['Pizza', 'Pasta', 'Risotto'],

  // ES6 enhanced object literals
  openingHours,

  order(starterIndex, mainIndex) {
    return [this.starterMenu[starterIndex], this.mainMenu[mainIndex]];
  },

  orderDelivery({ starterIndex = 1, mainIndex = 0, time = '20:00', address }) {
    console.log(
      `Order received! ${this.starterMenu[starterIndex]} and ${this.mainMenu[mainIndex]}
      will be delivered to ${address} at ${time}`
    );
  },

  orderPasta(ing1, ing2, ing3) {

```

```

console.log(
  `Here is your delicious pasta with ${ing1}, ${ing2} and ${ing3}`
);
},

orderPizza(mainIngredient, ...otherIngredients) {
  console.log(mainIngredient);
  console.log(otherIngredients);
},
};

```

### // Destructuring Objects

```

restaurant.orderDelivery({
  time: '22:30',
  address: 'Via del Sole, 21',
  mainIndex: 2,
  starterIndex: 2,
});

```

```

restaurant.orderDelivery({
  address: 'Via del Sole, 21',
  starterIndex: 1,
});

```

```

const { name, openingHours, categories } = restaurant;
console.log(name, openingHours, categories);

```

```

const {
  name: restaurantName,
  openingHours: hours,
  categories: tags,
} = restaurant;
console.log(restaurantName, hours, tags);

```

### // Default values

```

const { menu = [], starterMenu: starters = [] } = restaurant;
console.log(menu, starters);

```

### // Mutating variables

```

let a = 111;
let b = 999;
const obj = { a: 23, b: 7, c: 14 };

```



```
{ a, b } = obj);  
console.log(a, b);
```

```
// Nested objects
```

```
const {  
  fri: { open: o, close: c },  
} = openingHours;  
console.log(o, c);
```

## ❖ // 5. Data structures, Modern Operators and Strings → The Spread Operator (...)

Used to basically expand an array into all its elements.

Spread Operator is an ES6 feature

```
// The Spread Operator (...)
```

```
const arr = [7, 8, 9];  
const badNewArr = [1, 2, arr[0], arr[1], arr[2]];  
console.log(badNewArr);
```

```
const newArr = [1, 2, ...arr];  
console.log(newArr);
```

```
console.log(...newArr);  
console.log(1, 2, 7, 8, 9);  
const newMenu = [...restaurant.mainMenu, 'Gnocci'];  
console.log(newMenu);
```

the spread operator is actually a bit similar to destructuring, because it also helps us get elements out of arrays. Now, the big difference is that the spread operator takes all the elements from the array and it also doesn't create new variables. And as a consequence, we can only use it in places where we would otherwise write values separated by commas.

```
// Copy array
```

```
const mainMenuCopy = [...restaurant.mainMenu];
```

```
// Join 2 arrays
```

```
const menu = [...restaurant.starterMenu, ...restaurant.mainMenu];  
console.log(menu);
```

the spread operator works on arrays, but that's not entirely true, because actually, the spread operator works on all so-called iterables. Now what is an iterable? Well, there are different iterables in JavaScript. Iterables are things like all arrays, strings, maps, or sets, but not objects.

we can still only use the spread operator when building an array, or when we pass values into a function.

```
// Iterables: arrays, strings, maps, sets. NOT objects
const str = 'Jonas';
const letters = [...str, '', 'S.'];
console.log(letters);
console.log(...str);
// console.log(`${...str} Schmedtmann`); // Error – cant use spread in template literal
```

```
// Real-world example
const ingredients = [
  // prompt("Let's make pasta! Ingredient 1?"),
  // prompt('Ingredient 2?'),
  // prompt('Ingredient 3'),
];
console.log(ingredients);
```

```
restaurant.orderPasta(ingredients[0], ingredients[1], ingredients[2]); // old solution
restaurant.orderPasta(...ingredients); // with spread operator
```

since ES 2018, the spread operator actually also works on objects, even though objects are not iterables.

```
// Objects
const newRestaurant = { foundedIn: 1998, ...restaurant, founder: 'Guiseppe' };
console.log(newRestaurant);
```

```
const restaurantCopy = { ...restaurant };
restaurantCopy.name = 'Ristorante Roma';
console.log(restaurantCopy.name);
console.log(restaurant.name);
```

## ❖ // 6. Data structures, Modern Operators and Strings → Rest Pattern and Parameters

- and the rest pattern looks exactly like the spread operator.  
has the same syntax with the three dots but it actually does the opposite of the spread operator.  
we used the spread operator to build new arrays or to pass multiple values into a function. in both cases, we use the spread operator to expand an array into individual elements.

Now, the rest pattern uses the exact same syntax however, to collect multiple elements and condense them into an array. so that's really the opposite of spread

The spread operator is to unpack an array while rest is to pack elements into an array

- the rest element must be the last element
- for the same reason there can only ever be one rest in any destructuring assignment

```
// Rest Pattern and Parameters
```

```
// 1) Destructuring
```

```
// SPREAD, because on RIGHT side of =
```

```
const arr = [1, 2, ...[3, 4]];
```

```
// REST, because on LEFT side of =
```

```
const [a, b, ...others] = [1, 2, 3, 4, 5];
```

```
console.log(a, b, others);
```

```
const [pizza, , risotto, ...otherFood] = [
```

```
  ...restaurant.mainMenu,
```

```
  ...restaurant.starterMenu,
```

```
];
```

```
console.log(pizza, risotto, otherFood);
```

```
// Objects
```

```
const { sat, ...weekdays } = restaurant.openingHours;
```

```
console.log(weekdays);
```

how can we do that:

```
add(2, 3);
```

```
add(5, 3, 7, 2);
```

```
add(8, 2, 5, 3, 2, 1, 4);
```

it is actually called rest parameters:

```
// 2) Functions
```

```
const add = function (...numbers) {
```

```
  let sum = 0;
```

```
  for (let i = 0; i < numbers.length; i++) sum += numbers[i];
```

```
  console.log(sum);
```

```
};
```

```
add(2, 3);
add(5, 3, 7, 2);
add(8, 2, 5, 3, 2, 1, 4);
```

// pass simply multiple parameters to a function :

```
const x = [23, 5, 7];
add(...x);
```

So the spread operator is used where we would otherwise write values, separated by a comma.

On the other hand the rest pattern is basically used where we would otherwise write variable names separated by commas.

So, again the rest pattern can be used where we would write variable names, separated by commas and not values separated by commas.

So it's a subtle distinction, but this is how you know when and where to use spread and rest.

## ❖ // 7. Data structures, Modern Operators and Strings → Short Circuiting (&& and)

**short circuiting: returning the first truthy value**

in the case of the OR operator short circuiting means that if the first value is a truthy value, it will immediately return that first value.

```
// Short Circuiting (&& and ||)
//console.log('---- OR ----');
// Use ANY data type, return ANY data type, short-circuiting
console.log(3 || 'Jonas'); // 3
console.log("" || 'Jonas'); // jonas
console.log(true || 0); // true
console.log(undefined || null); // null
console.log(undefined || 0 || "" || 'Hello' || 23 || null); // hello
```

```
restaurant.numGuests = 0;
const guests1 = restaurant.numGuests ? restaurant.numGuests : 10;
console.log(guests1);
```

```
const guests2 = restaurant.numGuests || 10;
console.log(guests2);
```

```
//console.log('---- AND ----');
console.log(0 && 'Jonas'); // 0
console.log(7 && 'Jonas'); // jonas
console.log('Hello' && 23 && null && 'jonas'); // Practical example
```

```
// Practical example
```

```
if (restaurant.orderPizza) {
  restaurant.orderPizza('mushrooms', 'spinach');
}
```

```
restaurant.orderPizza && restaurant.orderPizza('mushrooms', 'spinach');
```

## ❖ // 8. Data structures, Modern Operators and Strings → Nullish Coalescing

introduced in ES2020

nullish coalescing operator works with the idea or with the concept of nullish values instead of falsy values.

**nullish values are null and undefined.**

It does not include a zero or the empty string.

```
// The Nullish Coalescing Operator
```

```
restaurant.numGuests = 0;
const guests = restaurant.numGuests || 10;
console.log(guests); // 10
```

```
// Nullish: null and undefined (NOT 0 or "")
```

```
const guestCorrect = restaurant.numGuests ?? 10;
console.log(guestCorrect); // 0
```

## ❖ // Data structures, Modern Operators and Strings → 10. Looping Arrays The for-of Loop

for-of loop (ES6)

- we can still use the continue and break keywords

```
// The for-of Loop
```

```
const menu = [...restaurant.starterMenu, ...restaurant.mainMenu];
```

```
for (const item of menu) console.log(item); // print each "item" of "menu" array
```

if we need index of elements: `array.entries()`

```
for (const item of menu.entries()) {  
  console.log(`${item[0] + 1}: ${item[1]}`); // [0,"pasta"] [1,"pizza"].....  
}
```

So this works great here, but we are actually at this point smarter than doing it like this And that's because if item is now an array, we can de-structure it. We don't have to manually take element zero and element one, that is the old school way.

```
for (const [i, el] of menu.entries()) {  
  console.log(`${i + 1}: ${el}`);  
}
```

## ❖ Data structures, Modern Operators and Strings → 11. Enhanced Object Literals

ES6 introduced three ways, which make it easier to write object literals

```
const openingHours = {  
  [weekdays[3]]: {  
    open: 12,  
    close: 22,  
  },  
  [weekdays[4]]: {  
    open: 11,  
    close: 23,  
  },  
  [weekdays[5]]: {  
    open: 0, // Open 24 hours  
    close: 24,  
  },  
};
```

```
const restaurant = {  
  name: 'Classico Italiano',  
  location: 'Via Angelo Tavanti 23, Firenze, Italy',
```

```
// ES6 enhanced object literals (instead of openingHours: openingHours)  
openingHours,
```

```
order(starterIndex, mainIndex) {
```

```

    return [this.starterMenu[starterIndex], this.mainMenu[mainIndex]];
  },
};

```

second enhancement to object literals is about writing methods.

in ES6 we no longer have to create a property, and then set it to a function expression,

before ES6:

```

const restaurant = {
  name: 'Classico Italiano',
  order: function(starterIndex, mainIndex) {
    return [this.starterMenu[starterIndex], this.mainMenu[mainIndex]];
  },
}

```

After ES6:

```

const restaurant = {
  name: 'Classico Italiano',
  order(starterIndex, mainIndex) {
    return [this.starterMenu[starterIndex], this.mainMenu[mainIndex]];
  },
}

```

the third enhancement is that we can now actually compute property names instead of having to write them out manually and literally:

```

const weekdays = ['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun'];
const openingHours = {
  [weekdays[3]]: {
    open: 12,
    close: 22,
  },
  [weekdays[4]]: { // we can use js on property name side
    open: 11,
    close: 23,
  },
  [`day-${3+2}`]: { // we can calculate on property name side
    open: 0, // Open 24 hours
    close: 24,
  },
}

```

```
};
```

## ❖ Data structures, Modern Operators and Strings → 12. Optional Chaining (.) Is a ES2020

if a certain property does not exist, then undefined is returned immediately.

```
// Optional Chaining
```

```
if (restaurant.openingHours && restaurant.openingHours.mon)  
  console.log(restaurant.openingHours.mon.open);
```

```
// console.log(restaurant.openingHours.mon.open);
```

```
// WITH optional chaining
```

```
console.log(restaurant.openingHours.mon?.open);  
console.log(restaurant.openingHours?.mon?.open); // multiple optional chaining
```

If left side of “?” is exist? Then read right side (a property exists if it's not null and not undefined), if not, then immediately undefined will be returned.

If we don't use optional chaining >> uncaught error for not exists  
But in optional chaining (use of “?”) if not exist > undefined

```
restaurant.openingHours.mon?.open
```

```
// optional chaining usage in Methods
```

```
console.log(restaurant.order?.(0, 1) ?? 'Method does not exist');  
console.log(restaurant.orderRisotto?.(0, 1) ?? 'Method does not exist');
```

```
// optional chaining usage in Arrays
```

```
const users = [{ name: 'Jonas', email: 'hello@jonas.io' }];  
// const users = [];
```

```
console.log(users[0]?.name ?? 'User array empty');
```

```
if (users.length > 0) console.log(users[0].name);  
else console.log('user array empty');
```

## ❖ Data structures, Modern Operators and Strings → 13. Looping Objects Object Keys, Values, and Entries

the for of loop > loop over a arrays, which remember is an Iterable, but we can also loop over objects, which are not Iterable, but in an indirect way.

we have different options here, depending on what exactly we want to loop over.



So do we want to loop over the objects, property names over the values or both together.

```
// Property NAMES
for (const day of Object.keys(openingHours)) { // Object.keys return a array
  console.log(day);
}

// Property VALUES
const values = Object.values(openingHours);
console.log(values);

// Entire object
const entries = Object.entries(openingHours);
console.log(entries);

// [key, value]
for (const [day, { open, close }] of entries) { // using Destructuring [key, value] (nested
  // Destructuring because value side is also an object)
  console.log(`On ${day} we open at ${open} and close at ${close}`);
}
```

## ❖ Data structures, Modern Operators and Strings → 15. Sets

ES6 built-in data structures :

Sets , maps (In the past, js only had objects and array)

**set** is basically just a collection of unique values. So that means that a set can never have any duplicates

```
// Sets
const ordersSet = new Set([ // we need to pass in an iterable (array,string ...)
  'Pasta',
  'Pizza',
  'Pizza',
  'Risotto',
  'Pasta',
  'Pizza',
]);
console.log(ordersSet); // {"Pasta","Pizza","Risotto"} (duplicates are gone)
```

set can hold mixed data types. there are no key value pairs, it's just a bunch of values grouped together, sets are also iterables,

set is still very different from an array its elements are unique. And second, because the order of elements in the set is irrelevant.

```
console.log(ordersSet.has('Pizza'));
console.log(ordersSet.size); // like array.lenght
console.log(ordersSet.has('Bread')); // like include method
ordersSet.add('Garlic Bread'); // add new element (will be ignored if exist )
ordersSet.delete('Risotto'); // delete element
ordersSet.clear(); // delete all of the elements
```

no need and no have index of elements because elements are unique  
the main use case of sets is actually to remove duplicate values of arrays.

// Example

```
const staff = ['Waiter', 'Chef', 'Waiter', 'Manager', 'Chef', 'Waiter'];
const staffUnique = [new Set(staff)];
console.log(staffUnique); // {"Waiter","Chef","Manager"} (isnt array)
```

```
const staff = ['Waiter', 'Chef', 'Waiter', 'Manager', 'Chef', 'Waiter'];
const staffUnique = [...new Set(staff)];
console.log(staffUnique); // ["Waiter","Chef","Manager"] (using spread > output is array )
```

we wanted to just know how many unique positions there are:

```
console.log(
  new Set(['Waiter', 'Chef', 'Waiter', 'Manager', 'Chef', 'Waiter']).size
);
```

## ❖ Data structures, Modern Operators and Strings → 16. Maps Fundamentals

ES6 built-in data structures

big difference between objects and maps is that in maps, the keys can have any type  
in objects, the keys are basically always strings.

// Maps: Fundamentals

```
const restaurant = new Map();
rest.set('name', 'Classico Italiano'); //add new element to data structure (and return
updated map)
rest.set(1, 'Firenze, Italy');
console.log(rest.set(2, 'Lisbon, Portugal')); // {"name => "Classico Italiano",1 => "Firenze,
Italy"}
```

```
// we can use chain for set
restaurant
.set('categories', ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'])
.set('open', 11)
.set('close', 23)
```

```
// read data from map
console.log(restaurant.get('name')); // Classico Italiano
console.log(restaurant.get(true)); // We are open :D
```

```
// some methods
console.log(rest.has('categories'));
rest.delete(2);
rest.clear();
```

- ❖ **Data structures, Modern Operators and Strings → 17 . Maps Iteration**  
there is actually another way (after set) of populating a new map

```
const question = new Map([ // this is exactly the same array structure that is returned from
calling Object.entries(). (array of arrays, first key, sec value)
  ['question', 'What is the best programming language in the world?'],
  [1, 'C'],
  [2, 'Java'],
  [3, 'JavaScript'],
  ['correct', 3],
  [true, 'Correct 🍷'],
  [false, 'Try again!'],
]);
console.log(question);
```

```
// Convert object to map
console.log(Object.entries(openingHours));
const hoursMap = new Map(Object.entries(openingHours));
console.log(hoursMap);
```

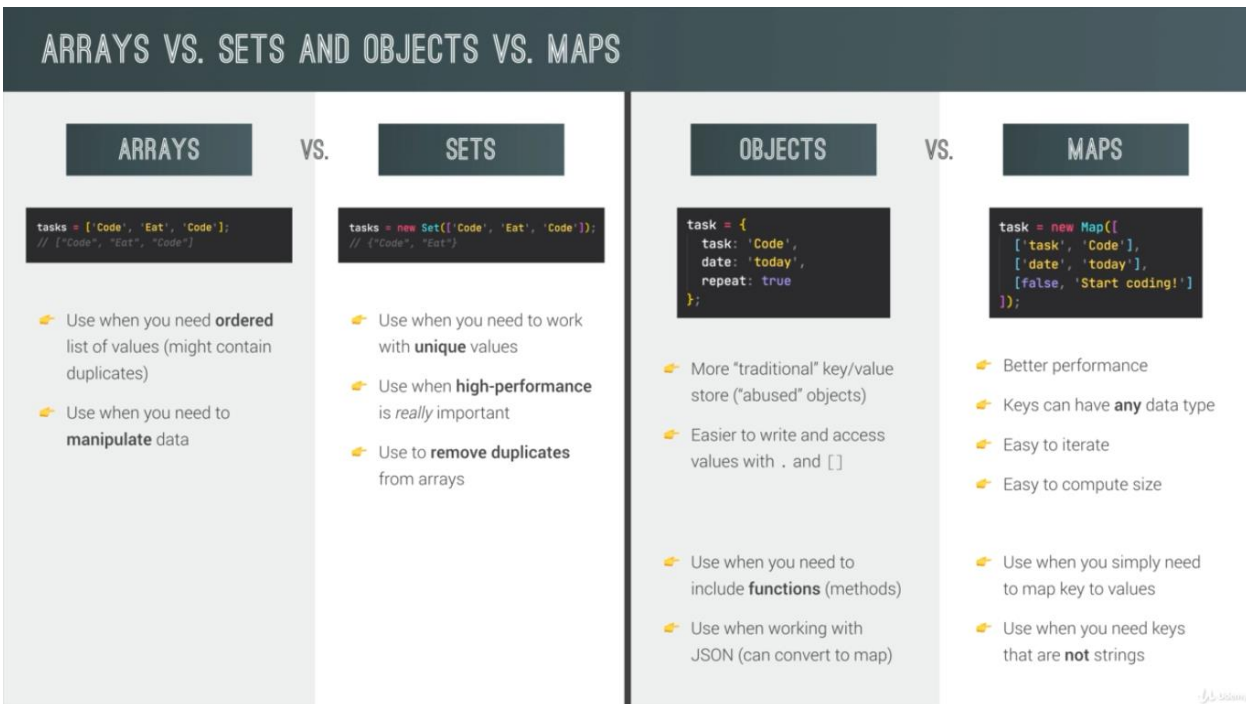
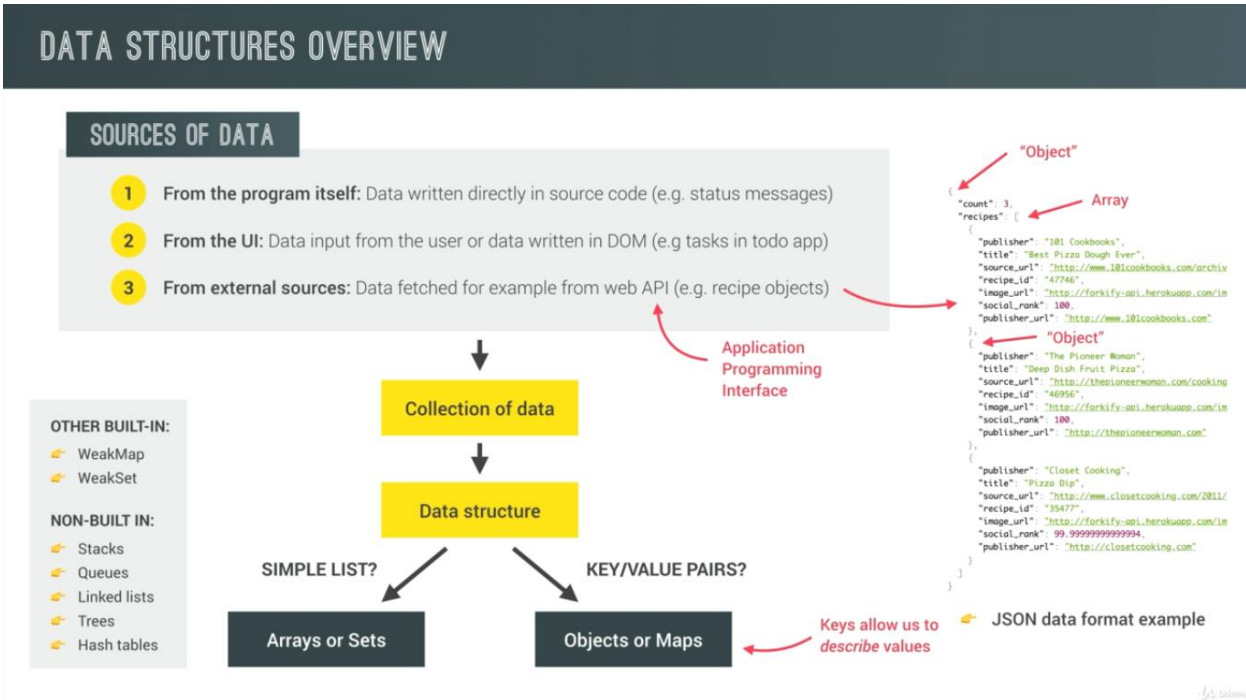
iteration is possible on maps because as we already

```
// iteration
for (const [key, value] of question) {
  console.log(`Answer ${key}: ${value}`);
}
```

```
}
```

```
// Convert map to array  
console.log([...question]);  
console.log([...question.keys()]);  
console.log([...question.values()]);
```

## ❖ Data structures, Modern Operators and Strings → 18. Summary Which Data Structure to Use



arrays versus sets, we already know that we should use them for simple lists of values when we do not need to describe the values. Now you should use arrays whenever

you need to store values in order and when these values might contain duplicates. Also you should always use arrays when you need to manipulate data because there are a ton of useful array methods.

Now sets on the other hand should only be used when you are working with unique values, you can also use sets when high performance is really important because operations like searching for an item or deleting an item from a set can be to 10 times faster in sets than in arrays. one great use case for sets is to remove duplicate values from an array

## ❖ Data structures, Modern Operators and Strings → 20. Working With Strings - Part 1

```
const airline = 'TAP Air Portugal';
const plane = 'A320';
```

```
console.log(plane[0]); // "A"
console.log(plane[1]); // "3"
console.log('B737'[0]); // "B"
```

```
console.log(airline.length); // "16"
console.log('B737'.length); // "4"
```

```
console.log(airline.indexOf('r')); // 6
console.log(airline.lastIndexOf('r')); // 10
console.log(airline.indexOf('Portugal')); // "8"
console.log(airline.indexOf('portugal')); // "-1" case sensitive (-1 when not exist)
```

```
console.log(airline.slice(4)); // "Air Portugal"
console.log(airline.slice(4, 7)); // "Air"
```

```
console.log(airline.slice(0, airline.indexOf(' '))); // "TAP"
console.log(airline.slice(airline.lastIndexOf(' ') + 1)); // "Portugal"
```

```
console.log(airline.slice(-2)); // "al"
console.log(airline.slice(1, -1)); // "AP Air Portuga"
```

\*\* Whenever we call a method on a string JavaScript will automatically behind the scenes convert that string primitive to a string object with the same content.

whenever we call a method on a string. And then when the operation is done the object is converted back to a regular string primitive.

## ❖ Data structures, Modern Operators and Strings → 21. Working With Strings - Part 2

// Working With Strings - Part 2

```

const airline = 'TAP Air Portugal';

console.log(airline.toLowerCase()); // tap air portugal
console.log(airline.toUpperCase()); // TAP AIR PORTUGAL

// Fix capitalization in name
const passenger = 'jOnAS';
const passengerLower = passenger.toLowerCase();
const passengerCorrect =
  passengerLower[0].toUpperCase() + passengerLower.slice(1);
console.log(passengerCorrect); // Jonas

// Comparing emails
const loginEmail = ' Hello@Jonas.io \n';
const normalizedEmail = loginEmail.toLowerCase().trim();
console.log(normalizedEmail); //hello@jonas.io

// replacing
const priceGB = '288,97£';
const priceUS = priceGB.replace('£', '$').replace(',', '.');
console.log(priceUS); //288.97$

const announcement =
  'All passengers come to boarding door 23. Boarding door 23!';

console.log(announcement.replace('door', 'gate'));
// console.log(announcement.replaceAll('door', 'gate')); // not work recently (so use next
line)
console.log(announcement.replace(/door/g, 'gate')); //All passengers come to boarding
gate 23. Boarding gate 23! (instead of previous line)

// Booleans
const plane = 'Airbus A320neo';
console.log(plane.includes('A320')); //true
console.log(plane.includes('Boeing')); //false
console.log(plane.startsWith('Airb')); //true

// example
if (plane.startsWith('Airbus') && plane.endsWith('neo')) {
  console.log('Part of the NEW ARirbus family');
}

```

❖ Data structures, Modern Operators and Strings → 22. Working With Strings – Part 3

```

// Split and join
console.log('a+very+nice+string'.split('+')); //["a", "very", "nice", "string"]
console.log('Jonas Schmedtmann'.split(' ')); //["Jonas", "Schmedtmann" ]

// Padding
const message = 'Go to gate 23!';
console.log(message.padStart(20, '+').padEnd(30, '+')); //+++++Go to gate 23!+++++
console.log('Jonas'.padStart(20, '+').padEnd(30, '+')); //+++++Jonas+++++

// Repeat
const message2 = 'Bad waether... All Departues Delayed... ';
console.log(message2.repeat(5));

```

[MDN String methods \(listed side of the web page\)](#)

### ❖ A Closer Look at Functions → 3. Default Parameters

```

// Default Parameters
const bookings = [];

const createBooking = function (flightNum, numPassengers = 1, price = numPassengers * 5)
{
    //// ES5 default value solution
    // numPassengers = numPassengers || 1;
    // price = price || 199;

    booking = {
        flightNum,
        numPassengers,
        price
    }
    console.log(booking)
    bookings.push(booking)
}

createBooking('lb3575')

```

we can't skip a default value. Solution for this case is send undefined value:

```
createBooking('LH123', undefined, 1000);
```

#### ❖ A Closer Look at Functions → 4. How Passing Arguments Works Value vs. Reference

kind of a review of primitive types and reference types.

```
// How Passing Arguments Works: Values vs. Reference
```

```
const flight = 'LH234';
```

```
const jonas = {  
  name: 'Jonas Schmedtmann',  
  passport: 24739479284,  
};
```

```
const checkIn = function (flightNum, passenger) {
```

```
  flightNum = 'LH999';
```

```
  passenger.name = 'Mr. ' + passenger.name; //effect directly on "jonas " object because  
the refrece is same
```

```
  if (passenger.passport === 24739479284) {
```

```
    alert('Checked in');
```

```
  } else {
```

```
    alert('Wrong passport!');
```

```
  }
```

```
};
```

```
checkIn(flight, jonas); // "flightNum" contains a copy of "flight" not the original one but  
when we manipulating te "passenger.name" it is exactly the same as manipulating the  
Jonas object
```

in programming, there are two terms that are used all the time when dealing with functions, which is passing by value, and passing by reference, **So JavaScript does not have passing by reference**, only passing by value, even though it looks like it's passing by reference.

#### ❖ A Closer Look at Functions → 5. First-Class and Higher-Order Functions

Some people think that first-class functions and higher order functions are the same thing. But actually they mean different things. So, first class functions is just a feature that a programming language either has or does not have. All it means is that all functions are values. There are no first class functions in practice, okay? It's just a concept. There are however higher order functions in practice, which are possible because the language supports first class functions.



# FIRST-CLASS VS. HIGHER-ORDER FUNCTIONS

## FIRST-CLASS FUNCTIONS

- JavaScript treats functions as **first-class citizens**
- This means that functions are **simply values**
- Functions are just another **"type" of object**

Store functions in variables or properties:

```
const add = (a, b) => a + b;  
  
const counter = {  
  value: 23,  
  inc: function() { this.value++; }  
};
```

Pass functions as arguments to OTHER functions:

```
const greet = () => console.log('Hey Jonas');  
btnClose.addEventListener('click', greet);
```

Return functions FROM functions

Call methods on functions:

```
counter.inc.bind(someOtherObject);
```

## HIGHER-ORDER FUNCTIONS

- A function that **receives** another function as an argument, that **returns** a new function, or **both**
- This is only possible because of first-class functions

1 Function that receives another function

```
const greet = () => console.log('Hey Jonas');  
btnClose.addEventListener('click', greet);
```

Higher-order function

Callback function

2 Function that returns new function

```
function count() {  
  let counter = 0;  
  return function() {  
    counter++;  
  };  
};
```

Higher-order function

Returned function

## ❖ A Closer Look at Functions → 6. Functions Accepting Callback Functions

we're calling the transformer function here and into that function we are passing the callback function and remember that we call these functions that we pass. So this one (\*\*), the callback functions. And that's because we do not call them ourselves. But instead we call JavaScript to basically tell them later. calling them later happens right here.(^^)

// Functions Accepting Callback Functions

```
const upperFirstWord = function (str) {  
  const [first, ...others] = str.split(' ');  
  return [first.toUpperCase(), ...others].join(' ');  
};
```

// Higher-order function

```
const transformer = function (str, fn) {  
  console.log(`Original string: ${str}`); //Original string: JavaScript is the best!  
  console.log(`Transformed string: ${fn(str)}`); // ^^ //Transformed string: JAVASCRIPT is the best!  
  console.log(`Transformed by: ${fn.name}`); //Transformed by: upperFirstWord  
};  
transformer('JavaScript is the best!', upperFirstWord); // **
```

this is exactly the same idea that we already talked about, using the add vent listener function.

```
// JS uses callbacks all the time
const high5 = function () {
  console.log('👏');
};
document.body.addEventListener('click', high5);
['Jonas', 'Martha', 'Adam'].forEach(high5); // call the "high5" function for each of array
elements - "forEach" is higher order and "high5" is call back
```

Why our callback functions so much used in JavaScript and why are they so helpful?

- it makes it easy to split up or code into more reusable and interconnected parts.
- the fact that callback functions allow us to create abstraction.
  - In “transformer “ function we created a new level of obstruction and by doing this or main transformer function, here is really only concerned with transforming the input string itself. But no matter how that transforming itself actually works.

For example, the add event listener function on its own would have no idea of what to do whenever the click event happens here. to tell the add event listener function exactly what to do.

## ❖ A Closer Look at Functions → 7. Functions Returning Functions

The opposite of previous session

```
// Functions Returning Functions
const greet = function (greeting) {
  return function (name) {
    console.log(` ${greeting} ${name}`);
  };
};
const greeterHey = greet('Hey'); // store the returned function on "greeterHey"
greeterHey('Jonas'); // "Hey Jonas"
greeterHey('Steven'); // "Hey Steven"
```

## ❖ A Closer Look at Functions → 8. The call and apply Methods

In this lecture we're gonna go back to the this keyword and learn how we can set the this keyword manually and also why we would want to do that.

“Call” allows us to manually and explicitly set the this keyword of any function that we want to call.

```
// The call and apply Methods
const lufthansa = {
  airline: 'Lufthansa',
```

```

iataCode: 'LH',
bookings: [],
// book: function() {}
book(flightNum, name) {
  console.log(
    `${name} booked a seat on ${this.airline} flight ${this.iataCode}${flightNum}`
  );
  this.bookings.push({ flight: `${this.iataCode}${flightNum}`, name });
},
};

```

```

lufthansa.book(239, 'Jonas Schmedtmann');
lufthansa.book(635, 'John Smith');

```

```

const eurowings = {
  airline: 'Eurowings',
  iataCode: 'EW',
  bookings: [],
};

```

```
const book = lufthansa.book;
```

```

// Does NOT work
// book(23, 'Sarah Williams');

```

```

// Call method
book.call(eurowings, 23, 'Sarah Williams');//now dose work
console.log(eurowings);

```

```

const swiss = {
  airline: 'Swiss Air Lines',
  iataCode: 'LX',
  bookings: [],
};
book.call(swiss, 583, 'Mary Cooper');

```

```

// Apply method
const flightData = [583, 'George Cooper'];
book.apply(swiss, flightData);
console.log(swiss);

```

```
book.call(swiss, ...flightData);
```

## ❖ A Closer Look at Functions → 9. The bind Method

## ❖ A Closer Look at Functions → 11. Immediately Invoked Function Expressions (IIFE)

sometimes in JavaScript, we need a function that is only executed once. And then never again.

```
(function () {  
  console.log('once print');  
  const isPrivate=11; // dont access out of function, data is encapsulated.  
})();
```

```
(() => console.log('also once again'))()
```

Well, we already know that functions create scopes, And what's important here is that one scope does not have access to variables from an inner scope in modern JavaScript. Immediately Invoked Function Expressions are not that used anymore. Because if all we want is to create a new scope for data privacy. All we need to do, is to just create a block like this “\*\*”

```
{ // **  
  const isPrivate = 23;  
  var notPrivate = 46;  
} // **  
// console.log(isPrivate);  
console.log(notPrivate);
```

## A Closer Look at Functions → 12. Closures

## A Closer Look at Functions → 13. More Closure Examples

## ❖ Working With Arrays → 3. Simple Array Methods

```
let arr = ['a', 'b', 'c', 'd', 'e'];  
// SLICE --> does not mutate the original arr, array, it returns a new array  
console.log(arr.slice(2));//[ 'a', 'b', 'c']  
console.log(arr.slice(2, 4)); //['c', 'd'] the end parameter here is not included in the  
output (length of the output array right here will be the end parameter minus the  
beginning one.)  
console.log(arr.slice(-2));//start from the end  
console.log(arr.slice(-1));//[ 'e'] simply get the last element of any array.  
console.log(arr.slice()); // ['a', 'b', 'c', 'd', 'e'] shallow copy of array  
console.log([...arr]);// ['a', 'b', 'c', 'd', 'e'] shallow copy of array  
  
// SPLICE --> does actually change the original array So it mutates that array.  
// console.log(arr.splice(2));  
arr.splice(-1);
```

```
console.log(arr);
arr.splice(1, 2);
console.log(arr);
//one pretty common use case of splice is to simply remove the last element of an
array.
```

// REVERSE → does actually mutate the original array

```
arr = ['a', 'b', 'c', 'd', 'e'];
const arr2 = ['j', 'i', 'h', 'g', 'f'];
console.log(arr2.reverse());
console.log(arr2);
```

// CONCAT

```
const letters = arr.concat(arr2);
console.log(letters);
console.log([...arr, ...arr2]); // does not mutate any of the involved arrays
```

// JOIN

```
console.log(letters.join(' - '));
```

#### ❖ Working With Arrays → 4. Looping Arrays forEach

So forEach is technically a higher order function which requires a callback function in order to tell it what to do

```
// Looping Arrays: forEach
const movements = [200, 450, -400, 3000, -650, -130, 70, 1300];

// for (const movement of movements) {
for (const [i, movement] of movements.entries()) {
  if (movement > 0) {
    console.log(`Movement ${i + 1}: You deposited ${movement}`);
  } else {
    console.log(`Movement ${i + 1}: You withdrew ${Math.abs(movement)}`);
  }
}

console.log('---- FOREACH ----');
movements.forEach(function (mov, i, arr) {
  if (mov > 0) {
    console.log(`Movement ${i + 1}: You deposited ${mov}`);
  } else {
    console.log(`Movement ${i + 1}: You withdrew ${Math.abs(mov)}`);
  }
})
```

```
});  
// 0: function(200)  
// 1: function(450)  
// 2: function(400)  
// ...
```

when should you use the for of loop one fundamental difference between the two of them is that you cannot break out of a forEach loop

So the continue and break statements do not work in a forEach loop at all

### ❖ Working With Arrays → 5. forEach With Maps and Sets

forEach is also available on maps and sets.

forEach it will call function with three arguments. the first one will be the current value, the current value in the current iteration, the second one is the key, and the third one is the entire map that is being looped over.

```
// forEach With Maps and Sets  
// Map  
const currencies = new Map([  
  ['USD', 'United States dollar'],  
  ['EUR', 'Euro'],  
  ['GBP', 'Pound sterling'],  
]);  
currencies.forEach(function (value, key, map) {  
  console.log(`${key}: ${value}`);  
});  
// Set  
const currenciesUnique = new Set(['USD', 'GBP', 'USD', 'EUR', 'EUR']);  
console.log(currenciesUnique);  
currenciesUnique.forEach(function (value, _, map) {  
  console.log(`${value}: ${value}`);  
});  
// Output (same for map and set):  
// USD : United States dollar  
// EUR: Euro  
// GBP: Pound sterling
```

### ❖ Working With Arrays → 7. Creating DOM Elements

Adding something to DOM

```
element.insertAdjacentHTML(position, text);
```

```
<!-- beforebegin -->
```

```
<p>
```

```
<!-- afterbegin -->
```

```
foo
```

```
<!-- beforeend -->
```

```
</p>
```

```
<!-- afterend -->
```

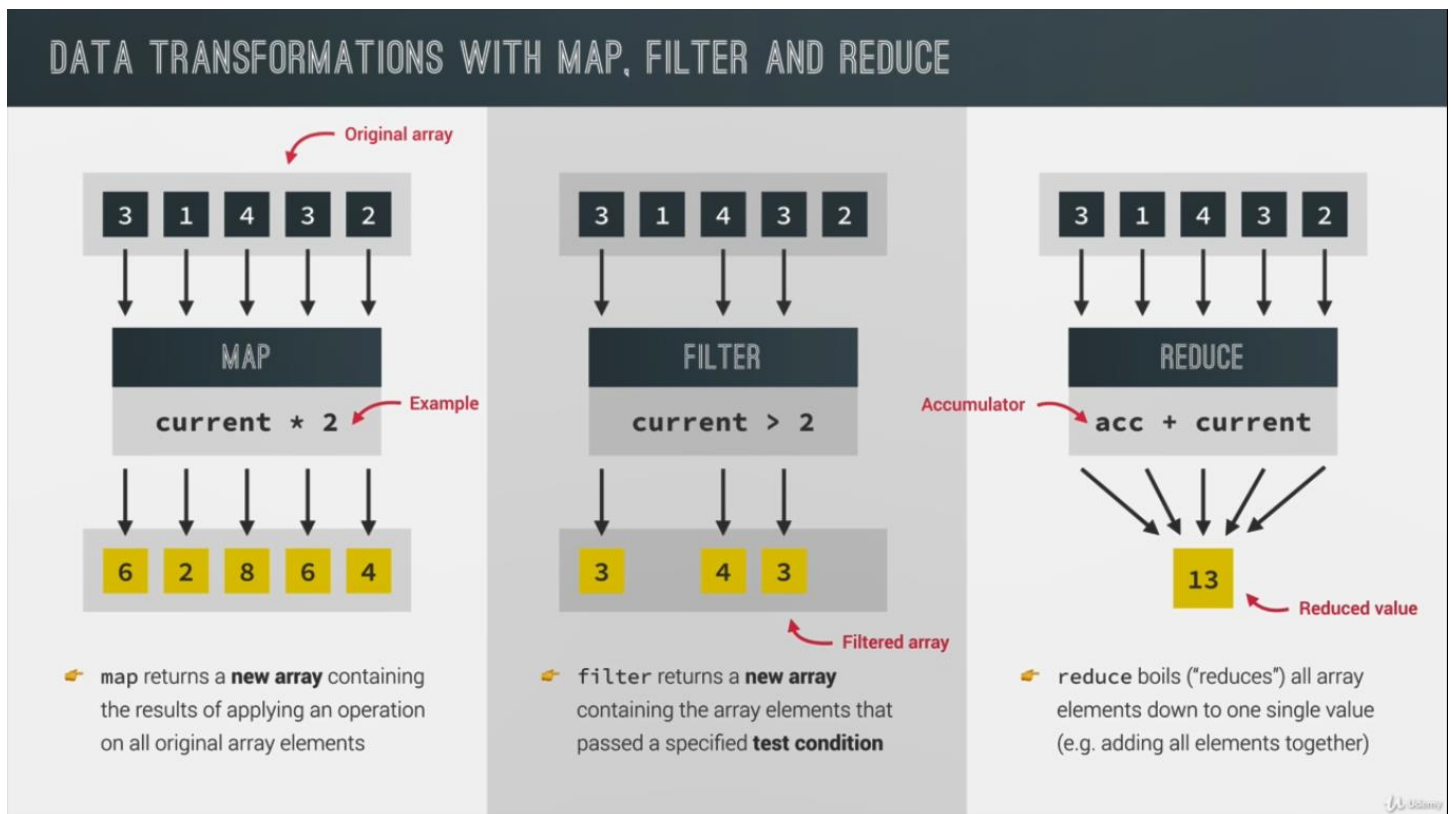
**.innerHTML** is a little bit similar to **.textContent** So remember that now the difference is that text content simply returns the text itself while HTML returns everything, including the HTML.

### ❖ Working With Arrays → 9. Data Transformations map, filter, reduce

In js there are three big and important array methods that we use all the time to perform data transformations. So basically, these are methods that we use to create new arrays based on transforming data from other arrays. the tools I'm talking about are; map, filter and reduce.

- **Map:** First the map method is yet another method that we can use to loop over arrays. So, map is actually similar to the forEach method that we studied before but with the difference that map creates a brand new array based on the original array. So essentially the map method takes an array, loops over that array and in each alteration, it applies a callback function that we specify on our code o the current array element.
- **Filter:** next up we have the filter method, which as the name says, is used to filter for elements in the original array which satisfy a certain condition. So all the elements that pass the test that we specified will make it into a new filtered array. Or in other words elements for which the condition is true will be included in a new array that the filter method returns.
- **Reduce:** And finally there is also the reduce method which we use to boil down all the elements of the original array into one single value.

Now we also said that this whole process has now reduced the original array to one single value which in this example is the sum of all the elements but it can of course be many other operations. Now it's this value that then actually gets



returned from the reduce method in the end.

### ❖ Working With Arrays → 10. The map Method

```
const movements = [200, 450, -400, 3000, -650, -130, 70, 1300];
const eurToUsd = 1.1;

const movementToUsd = movements.map(function (mov) {
  return mov * eurToUsd;
})
console.log(movementToUsd)
// *****
// with arrow functon:
const movementToUSDArrow = movements.map(mov => mov * eurToUsd);
console.log(movementToUSDArrow)
// *****
// with for of:
const movementToUSDfor = []
for (let mov of movements) {
  movementToUSDfor.push(mov * eurToUsd);
}
console.log(movementToUSDfor)
```



\*\*use function (arrow is better), because functional programming.

### ❖ Working With Arrays → 12. The filter Method

```
const movements = [200, 450, -400, 3000, -650, -130, 70, 1300];  
// The filter Method  
const deposits = movements.filter(function (mov) {  
  return mov > 0;  
});  
console.log(deposits); // [200, 450, 3000, 70, 1300]  
  
const depositsFor = [];  
for (const mov of movements) if (mov > 0) depositsFor.push(mov);  
console.log(depositsFor); // [200, 450, 3000, 70, 1300]  
  
const withdrawals = movements.filter(mov => mov < 0);  
console.log(withdrawals); // [-400, -650, -130]
```

### ❖ Working With Arrays → 13. The reduce Method

So the reduce function also gets a callback function, but this one is a little bit different from the other ones, like the one in map or for each. So in these other callbacks, the first parameter is always the current element of the array. Let's call it current. The second one is the current index and the third one is the entire array. But here in the callback function of the reduce method, the first parameter is actually something called the accumulator.

```
// accumulator -> SNOWBALL  
const balance = movements.reduce(function (acc, cur, i, arr) {  
  console.log(`Iteration ${i}: ${acc}`);  
  return acc + cur;  
}, 0);  
  
// using arrow fuction  
const balance = movements.reduce((acc, cur) => acc + cur, 0);  
console.log(balance);  
  
// using for instead of reduce  
let balance2 = 0;  
for (const mov of movements) balance2 += mov;  
console.log(balance2);  
  
// Maximum value  
const max = movements.reduce((acc, mov) => {
```

```
if (acc > mov) return acc;
else return mov;
}, movements[0]);
console.log(max);
```

### ❖ Working With Arrays → 15. The Magic of Chaining Methods

```
const totalDepositsUSD = movements
  .filter(mov => mov > 0)
  .map(mov => mov * eurToUsd)
  .reduce((acc, mov) => acc + mov, 0);
console.log(totalDepositsUSD);
```

### ❖ Working With Arrays → 17. The find Method

Find method: So as the name says, we can use the Find method to retrieve one element of an array based on a condition.

unlike the Filter method, the Find method will actually not return a new array but it will only return the first element in the array that satisfies this condition.

the Find method is a bit similar to the Filter method, but there are two fundamental differences. First Filter returns all the elements that match the condition while the Find method only returns the first one and second and even more important, the Filter method returns a new array while Find only returns the element itself

```
const account1 = {
  owner: 'Jonas Schmedtmann',
  movements: [200, 450, -400, 3000, -650, -130, 70, 1300],
  interestRate: 1.2, // %
  pin: 1111,
};

const account2 = {
  owner: 'Jessica Davis',
  movements: [5000, 3400, -150, -790, -3210, -1000, 8500, -30],
  interestRate: 1.5,
  pin: 2222,
};
```

```
const accounts = [account1, account2];
```

```
const account = accounts.find(acc => acc.owner === 'Jessica Davis');  
console.log(account); // Object { owner: "Jessica Davis", movements: (8) [...], interestRate:  
1.5, pin: 2222 }
```

## ❖ Working With Arrays → 20. The findIndex Method

the the findIndex method works almost the same way as find. But as the name says, findIndex returns the index of the found element and not the element itself.

```
btnClose.addEventListener('click', function (e) {  
  e.preventDefault();  
  
  if (  
    inputCloseUsername.value === currentAccount.username &&  
    Number(inputClosePin.value) === currentAccount.pin  
  ) {  
    const index = accounts.findIndex(  
      acc => acc.username === currentAccount.username  
    );  
    console.log(index);  
    // .indexOf(23)  
  
    // Delete account  
    accounts.splice(index, 1);  
  
    // Hide UI  
    containerApp.style.opacity = 0;  
  }  
  
  inputCloseUsername.value = inputClosePin.value = "";  
});
```

both the find and findIndex methods get access to also the current index, and the current entire array. So as always, besides the current element, these other two values are also available. But in practice, I never found these useful. And second, the both the find and findIndex methods were added to JavaScript in ES6. And so they will not work in like super

old browsers. But don't worry, there is going to be a lecture a little bit later on how to support all of these old browsers.

## ❖ Working With Arrays → 21. some and every

We can use the includes method to test if an array includes a certain value. this is essentially testing for equality but what if we wanted to test for a condition instead? And so that's where this some method comes into play.

```
// some and every
const movements = [200, 450, -400, 3000, -650, -130, 70, 1300];

// EQUALITY
console.log(movements.includes(-130));

// SOME: CONDITION
console.log(movements.some(mov => mov === -130));

const anyDeposits = movements.some(mov => mov > 0);
console.log(anyDeposits);
```

the every method is pretty similar to the some method but as you might guess, the difference between them is that every only returns true if all of the elements in the array satisfy the condition that we pass in. So in other words, if every element passes the test in our callback function, only then the every method returns true (“any” ask is all of the values under condition??)

```
// EVERY
console.log(movements.every(mov => mov > 0));
console.log(account4.movements.every(mov => mov > 0));
```

when we want to define callback function separate and pass as argument:  
this help to DRY (don't repeat yourself)

```
// Separate callback
const deposit = mov => mov > 0;
console.log(movements.some(deposit));
console.log(movements.every(deposit));
console.log(movements.filter(deposit));
```

## ❖ Working With Arrays → 22. flat and flatMap

flat and also flat map were introduced in **ES2019**.

```
// flat and flatMap
const arr = [[1, 2, 3], [4, 5, 6], 7, 8];
console.log(arr.flat()); // [1,2,3,4,5,6,7,8]

const arrDeep = [[[1, 2], 3], [4, [5, 6]], 7, 8];
console.log(arrDeep.flat())//[[1,2],3,4,[5, 6],7,8]
// to solve this nesting matter we use level argument => .flat(2)
console.log(arrDeep.flat(2)); // [1,2,3,4,5,6,7,8]
```

using a map first and then flattening the result, and it's a pretty common operation.

```
// flat
const overalBalance = accounts
  .map(acc => acc.movements)
  .flat()
  .reduce((acc, mov) => acc + mov, 0);
console.log(overalBalance);
```

there is another method that was also introduced at the same time, which is flat map. And so flat map essentially combines, a map and a flat method, into just one method, which is better for performance.

```
// flatMap
const overalBalance2 = accounts
  .flatMap(acc => acc.movements)
  .reduce((acc, mov) => acc + mov, 0);
console.log(overalBalance2);
```

\* flatmap here only goes one level deep and we cannot change it. So if you do need to go deeper than just one level, you still need to use the flat method.

## ❖ Working With Arrays → 23. Sorting Arrays

```
// Strings
const owners = ['Jonas', 'Zach', 'Adam', 'Martha'];
console.log(owners.sort());
console.log(owners);
```

```
// Numbers
```

```

console.log(movements);

// return < 0, A, B (keep order)
// return > 0, B, A (switch order)

// Ascending
// movements.sort((a, b) => {
//   if (a > b) return 1;
//   if (a < b) return -1;
// });
movements.sort((a, b) => a - b);
console.log(movements);

// Descending
// movements.sort((a, b) => {
//   if (a > b) return -1;
//   if (a < b) return 1;
// });
movements.sort((a, b) => b - a);
console.log(movements);

```

### ❖ Working With Arrays → 24. More Ways of Creating and Filling Arrays

Array.fill()

Array.from()

```

const z = Array.from({ length: 7 }, (_, i) => i + 1);
console.log(z); // [1,2,3,4,5,6,7]

```

### ❖ Working With Arrays → 25. Summary Which Array Method to Use

So do I want to mutate the original array, or do I want a new array? Do I maybe want an array index, or do I want to retrieve an entire array element? Or do I want to know, if an array includes, a certain element,

or maybe I just want to get a new string, to transform the array to a new value,

or simply to loop over the array? Well, these are a lot of different scenarios, and by asking exactly these questions,

we can now categorize the methods, and then easily choose between them.

So this is kind of a framework, that I developed, and so let me go briefly, through all of these different scenarios with you.

So these are the methods that mutate the original array.

So if we want to add an element, to the original array, we can use push or unshift.

So we already talked about all of these methods, so of course I will not explain them again, this is just a reference for you to keep.

Now anyway, if we want to remove, from the original, then we use either, pop, shift or splice. And finally, we also talked about,

three other methods that mutate the original array, which is the reverse, sort and fill methods. Okay.

So keep in mind, that you have to be careful, because they do change the underlying array, that we're working with.

And that many times is not what we want. So if instead, we want a new array, then we can use one of these methods.

So if we want to calculate one array, from the original, then we use map methods. And this one we use all the time, as it loops over the original array, and creates a new one, based on that. We can also create new arrays, by filtering for a condition, or by taking a slice of the original array. We can also concatenate two arrays, and create a new array based on that, and finally, we can flatten the original array, using flat and flatMap.

Next, sometimes we need an array index. And for that, we have the indexOf and findIndex methods. Now the difference between them, is that findIndex can basically search, for an element in the array, based on a test condition, that we provide, in the callback function.

But besides that, they are pretty similar, and both give us the index, of a certain element in an array. Now, if we actually need the array element itself, then we use, the find method. And so this one again, is based on a test condition, specified in a callback function. Next up, sometimes it's very important to know, whether an array includes, a certain element or not. And for that we have, includes, some and every. So with includes, we can simply test, if an array contains a single value, while on the other hand, with some and every, we can specify a condition, based on a callback function.

So some is gonna return, if at least one of the elements in the array, satisfies the condition, and every, only returns true, if all of the elements, satisfy the condition. So these three methods, are all return Boolean values, which is very helpful, in a condition. For example, in an if/else statement. Okay. So that's where you many times will use, one of these methods.

Next up, sometimes we want to transform an array, into a string. And then we use the join method. Or we might want to reduce the entire array, to just one value.

And as we already know, for that, we use the reduce method, just as we studied in this section. So in the reduce method, we use an accumulator, which has like a snowball, to boil down, an entire array, to just one single value. And that value can be of any type.

So it can be a number, string, Boolean, or even an array or an object. And so in fact, we could, replace some of these methods that we have here, with a reduce. ow, if all we want to do, is to just loop over an array, without producing any new value, we use the, forEach method.

So remember, this one does not create a new array, and in fact, it doesn't create any new value. All we do in the `forEach` method is to, basically just do some work, but not produce a new value. All right? And that's it. That's an overview of all the 23 methods, that we can use on arrays. And it might be a good idea, to somehow keep this overview handy. For example, print it out on a paper, or just store it somewhere, on your computer. Anyway, now all there's left to do in the section, is to complete the final coding challenge. So let's do that right in the next video.

## WHICH ARRAY METHOD TO USE? 🤔

"I WANT...:"

To mutate original array	A new array	An array index	Know if array includes	To transform to value
<p>➤ Add to original:</p> <ul style="list-style-type: none"> <li><code>.push</code> (end)</li> <li><code>.unshift</code> (start)</li> </ul> <p>➤ Remove from original:</p> <ul style="list-style-type: none"> <li><code>.pop</code> (end)</li> <li><code>.shift</code> (start)</li> <li><code>.splice</code> (any)</li> </ul> <p>➤ Others:</p> <ul style="list-style-type: none"> <li><code>.reverse</code></li> <li><code>.sort</code></li> <li><code>.fill</code></li> </ul>	<p>➤ Computed from original:</p> <ul style="list-style-type: none"> <li><code>.map</code> (loop)</li> </ul> <p>➤ Filtered using condition:</p> <ul style="list-style-type: none"> <li><code>.filter</code></li> </ul> <p>➤ Portion of original:</p> <ul style="list-style-type: none"> <li><code>.slice</code></li> </ul> <p>➤ Adding original to other:</p> <ul style="list-style-type: none"> <li><code>.concat</code></li> </ul> <p>➤ Flattening the original:</p> <ul style="list-style-type: none"> <li><code>.flat</code></li> <li><code>.flatMap</code></li> </ul>	<p>➤ Based on value:</p> <ul style="list-style-type: none"> <li><code>.indexOf</code></li> </ul> <p>➤ Based on test condition:</p> <ul style="list-style-type: none"> <li><code>.findIndex</code></li> </ul> <p><b>An array element</b></p> <p>➤ Based on test condition:</p> <ul style="list-style-type: none"> <li><code>.find</code></li> </ul>	<p>➤ Based on value:</p> <ul style="list-style-type: none"> <li><code>.includes</code></li> </ul> <p>➤ Based on test condition:</p> <ul style="list-style-type: none"> <li><code>.some</code></li> <li><code>.every</code></li> </ul> <p><b>A new string</b></p> <p>➤ Based on separator string:</p> <ul style="list-style-type: none"> <li><code>.join</code></li> </ul>	<p>➤ Based on accumulator:</p> <ul style="list-style-type: none"> <li><code>.reduce</code></li> </ul> <p><i>(Boil down array to single value of any type: number, string, boolean, or even new array or object)</i></p> <p><b>To just loop array</b></p> <p>➤ Based on callback:</p> <ul style="list-style-type: none"> <li><code>.forEach</code></li> </ul> <p><i>(Does not create a new array, just loops over it)</i></p>

## ❖ 12. Numbers, Dates, Intl and Timers → 3. Converting and Checking Numbers

in JavaScript, all numbers are presented internally as floating point numbers. So basically, always as decimals, no matter if we actually write them as integers or as decimals.

`23 === 23.0`

```
// Conversion
console.log(Number('23')); // 23
console.log(+ '23'); // 23

// Parsing
console.log(Number.parseInt('30px', 10)); // 30
```



```

console.log(Number.parseInt('e23', 10)); // NaN

console.log(Number.parseInt(' 2.5rem ')); // 2
console.log(Number.parseFloat(' 2.5rem ')); // 2.5

// console.log(parseFloat(' 2.5rem '));

```

### Don't use this in practice

```

// Check if value is NaN
console.log(Number.isNaN(20)); // false
console.log(Number.isNaN('20')); // false
console.log(Number.isNaN(+ '20X')); // true
console.log(Number.isNaN(23 / 0)); // false

```

this is your go-to whenever you need to check if something is a number or not.

```

// Checking if value is number
console.log(Number.isFinite(20)); // true
console.log(Number.isFinite('20')); // false
console.log(Number.isFinite(+ '20X')); // false
console.log(Number.isFinite(23 / 0)); // false

```

```

console.log(Number.isInteger(23)); // true
console.log(Number.isInteger(23.0)); // true
console.log(Number.isInteger(23 / 0)); // false

```

## ❖ 12. Numbers, Dates, Intl and Timers → 4. Math and Rounding

```

console.log(Math.sqrt(25)); //5
console.log(25 ** (1 / 2)); //5
console.log(8 ** (1 / 3)); //2

console.log(Math.max(5, 18, 23, 11, 2)); // 23
console.log(Math.max(5, 18, '23', 11, 2)); // 23
console.log(Math.max(5, 18, '23px', 11, 2)); // NaN
console.log(Math.min(5, 18, 23, 11, 2)); // 2
console.log(Math.PI * Number.parseFloat('10px') ** 2); //314.1592653589793
console.log(Math.trunc(Math.random() * 6) + 1); //2
const randomInt = (min, max) =>
  Math.floor(Math.random() * (max - min) + 1) + min;

```

```
// 0...1 -> 0...(max - min) -> min...max
// console.log(randomInt(10, 20));
```

```
// Rounding integers
```

```
console.log(Math.round(23.3)); //23
console.log(Math.round(23.9)); //24
```

```
console.log(Math.ceil(23.3)); //24
console.log(Math.ceil(23.9)); //24
```

```
console.log(Math.floor(23.3)); //23
console.log(Math.floor('23.9')); // 23
```

```
console.log(Math.trunc(23.3)); // 23
```

```
console.log(Math.trunc(-23.3)); // -23
console.log(Math.floor(-23.3)); // -24
```

```
// Rounding decimals
```

```
console.log((2.7).toFixed(0)); // "3"
console.log((2.7).toFixed(3)); // "2.700"
console.log((2.345).toFixed(2)); // "2.35"
console.log(+ (2.345).toFixed(2)); //2.35
```

## ❖ 12. Numbers, Dates, Intl and Timers → 5. The Remainder Operator

```
// The Remainder Operator
```

```
console.log(5 % 2); // 1
console.log(5 / 2); // 5 = 2 * 2 + 1 => output is 2.5
```

```
console.log(8 % 3); // 2
console.log(8 / 3); // 8 = 2 * 3 + 2 => output is 2.6666666666666665
```

```
console.log(6 % 2); // 0
console.log(6 / 2); // 3
```

```
console.log(7 % 2); //1
console.log(7 / 2); //3.5
```

```
const isEven = n => n % 2 === 0;
console.log(isEven(8)); //true
console.log(isEven(23)); //false
console.log(isEven(514)); //true
```

```

//creating kind of zebra background
// to convere node list to real array we used "...
labelBalance.addEventListener('click', function () {
  [...document.querySelectorAll('.movements__row')].forEach(function (row, i) {
    // 0, 2, 4, 6
    if (i % 2 === 0) row.style.backgroundColor = 'orangered';
    // 0, 3, 6, 9
    if (i % 3 === 0) row.style.backgroundColor = 'blue';
  })
});

```

## ❖ 12. Numbers, Dates, Intl and Timers → 6. Working with BigInt

one of the primitive data types, introduced in year 2020

```

// Working with BigInt
console.log(2 ** 53 - 1); //9007199254740991
console.log(Number.MAX_SAFE_INTEGER); //9007199254740991

```

this n here basically transforms a regular number, into a BigInt number.

```

const huge = 20289830237283728378237n;
const num = 23;
console.log(huge * num); //error: cannot mix BigInt and other types. to fix use=> huge *
BigInt(num)

```

there are two exceptions to this which are the comparison operators and the plus operator when working with strings.

```

console.log(Math.sqrt(16n)); // error - Math doesnt work on big int
// Exceptions
console.log(20n > 15); //true
console.log(20n === 20); //false the types are different
console.log(20n == '20'); //true

const huge = 20289830237283728378237n;
console.log(huge + ' is REALLY big!!!'); // "20289830237283728378237 is REALLY big!!!" (as
string)

// Divisions
console.log(11n / 3n); // 3n
console.log(10 / 3); // 3.3333333333333335
console.log(10n / 3n); // 3n
console.log(12n / 3n); // 4n

```

```
console.log(9n / 3n); // 3n
```

## ❖ 12. Numbers, Dates, Intl and Timers → 7. Creating Dates

create a date, and there are exactly four ways of creating dates in JavaScript. I mean, they all use the new date constructor function, but they can accept different parameters.

```
// Create a date
```

```
const now = new Date();
```

```
console.log([now]); // Sat Jan 08 2022 09:54:31 GMT+0330 (Iran Standard Time)
```

```
console.log(new Date('Aug 02 2020 18:05:41')); // Sun Aug 02 2020 18:05:41 GMT+0430  
(Iran Daylight Time)
```

```
console.log(new Date('December 24, 2015')); // Thu Dec 24 2015 00:00:00 GMT+0330 (Iran  
Standard Time)
```

```
console.log(new Date(account1.movementsDates[0])); // Tue Nov 19 2019 01:01:17  
GMT+0330 (Iran Standard Time)
```

```
console.log(new Date(2037, 10, 19, 15, 23, 5)); // Thu Nov 19 2037 15:23:05 GMT+0330  
(Iran Standard Time)
```

```
console.log(new Date(2037, 10, 31)); // Tue Dec 01 2037 00:00:00 GMT+0330 (Iran Standard  
Time)
```

```
console.log(new Date(0)); // Thu Jan 01 1970 03:30:00 GMT+0330 (Iran Standard Time) -----  
(zero milliseconds after that initial Unix time)
```

```
console.log(new Date(3 * 24 * 60 * 60 * 1000)); // Sun Jan 04 1970 03:30:00 GMT+0330  
(Iran Standard Time)
```

```
3 * 24 * 60 * 60 * 1000 // 259200000 (this is the timestamp of the day number three)
```

```
// Working with dates
```

```
const future = new Date(2037, 10, 19, 15, 23);
```

```
console.log(future); // Thu Nov 19 2037 15:23:00 GMT+0330 (Iran Standard Time)
```

```
console.log(future.getFullYear()); // 2037 (also getYear is exist but dont use )
```

```
console.log(future.getMonth()); // 10
```

```
console.log(future.getDate()); // 19
```

```
console.log(future.getDay()); // 4
```

```
console.log(future.getHours()); // 15
```

```
console.log(future.getMinutes()); // 23
```

```
console.log(future.getSeconds()); // 0
```

```
console.log(future.toISOString()); // 2037-11-19T11:53:00.000Z (this is the ISO string, which  
follows some kind of international standard)
```

```
console.log(future.getTime()); // 2142244380000 (timestamp)
```

```
console.log(new Date(2142256980000)); //Thu Nov 19 2037 18:53:00 GMT+0330 (Iran Standard Time) ----- (using timestamp to create date)
```

if you want simply the current timestamp for this exact moment, then you don't even need to create a new date. All we have to do is to call `date.now`

```
console.log(Date.now()); //1641624144293
```

Finally, there are also the set versions of all of `getMonth`, `getDay`,... methods.

```
future.setDate(2040);
console.log(future); // Mon Nov 19 2040 15:23:00 GMT+0330 (Iran Standard Time)
```

## ❖ 12. Numbers, Dates, Intl and Timers → 8. Adding Dates to Bankist App

```
const now = new Date();
const day = `${now.getDate()}.padStart(2,0)`.padStart(2,0); // use .padStart to add 0 if we have 1 digit
const month = `${now.getMonth()+1}.padStart(2,0)`.padStart(2,0); // add +1 because getMonth is 0 base
const year = now.getFullYear();
labelDate.textContent = `${day}/${month}/${year} , ${hour}:${min}`; // 30/01/2022
```

## ❖ 12. Numbers, Dates, Intl and Timers → 9. Operations With Dates

```
// Operations With Dates
const future = new Date(2037, 10, 19, 15, 23);
console.log(+future) // convert to timestamps(milliseconds) with "+" or Number()
console.log(Number(future));

// calculatind
const calcDaysPassed = (date1,date2) => Math.abs(date2 - date1) / (1000 * 60 * 60 * 24)
const days1 = calcDaysPassed(new Date(2037, 3, 24),new Date(2037, 3, 14));
console.log(days1);
// Math.abs is for removing "-",
// (1000 * 60 * 60 * 24) is for convert milliseconds to days
```

## ❖ 12. Numbers, Dates, Intl and Timers → 10. Internationalizing Dates (Intl)

```
// Internationalization API
const now = new Date();
console.log(new Intl.DateTimeFormat('fa-IR').format(now)) // ۱۳۹۷/۱۱/۱۹ . .
```

```
// "fa-IR" is ISO language code
```

we can actually take it to the next level and add some options to also customize this a little bit. For example, you see that right now, it only displays the date, but not any time. And so we can change that by providing an options object

```
// Internationalization API
const now = new Date();
const options = {
  hour:'numeric',
  minute:'numeric',
  day:'numeric',
  month:'numeric',
  // month:'2-digit',
  month:'long',
  year:'numeric',
  // year:'2-digit',
  weekday:'long',
  // weekday:'short',
  // weekday:'narrow',
}
const locale = navigator.language; // get the local from user browser and show in right ISO language code
console.log(locale)// en-GB - depends on user locale
console.log(new Intl.DateTimeFormat('en-GB',locale, options).format(now))
```

## ❖ 12. Numbers, Dates, Intl and Timers → 11. Internationalizing Numbers (Intl)

```
// Internationalizing Numbers (Intl)
const num = 3884764.23;

const options = {
  style: 'currency',
  unit: 'celsius',
  currency: 'EUR',
  // useGrouping: false, // print whitout separator
};

console.log('US: ', new Intl.NumberFormat('en-US',
options).format(num));//US:   €3,884,764.23
console.log('Germany: ', new Intl.NumberFormat('de-DE',
options).format(num));//Germany: 3.884.764,23 €
```

```

console.log('Syria: ', new Intl.NumberFormat('ar-SY',
options).format(num)); // Syria: ٣,٨٨٤,٧٦٤,٢٣ €
console.log(
  navigator.language,
  new Intl.NumberFormat(navigator.language, options).format(num)
); // en-US €3,884,764.23

```

❖ **12. Numbers, Dates, Intl and Timers** → **12. Timers** `setTimeout` and `setInterval` in JavaScript we have two kinds of timers. First, the **set timeout** timer runs **just once**, after a defined time, while the **set interval** timer **keeps running basically forever**, until we stop it.

```

setTimeout(()=>console.log('im setTimeout'),3000)

```

if we now needed to pass some arguments into this function here It's not that simple Because we are not calling this function ourselves. So we are not using the parenthesis like this “()” .

The solution is: basically, all the arguments here that we pass after the delay will be arguments to the function.

```

// setTimeout
const ingredients = ['olives', 'spinach'];
const pizzaTimer = setTimeout(
  (ing1, ing2) => console.log(`Here is your pizza with ${ing1} and ${ing2} 🍕`),
  3000,
  ...ingredients
);

```

we can actually cancel the timer, at least until the delay has actually passed.

```

if (ingredients.includes('spinach')) clearTimeout(pizzaTimer);

```

```

// setInterval
setInterval(function () {
  const now = new Date();
  console.log(now);
}, 1000);

```

```

const options={

```

```

hour:'numeric',
minute:'numeric',
second:'numeric',
}

setInterval(() => {
  const now = new Date();
  const time = Intl.DateTimeFormat('fa-IR',options).format(now)
  console.log(
    time
  )
}, 1000);

// ۱۴:۳۷:۵۰
// ۱۴:۳۷:۵۱
// ۱۴:۳۷:۵۲
// .
// .
// .

```

Creating countdown timer and logout user after that:

```

const startLogOutTimer = function () {
  const tick = function () {
    // make minute and second
    const min = String(Math.trunc(time / 60)).padStart(2, 0);
    const sec = String(time % 60).padStart(2, 0);

    // In each call, print the remaining time to UI
    labelTimer.textContent = `${min}:${sec}`;

    // When 0 seconds, stop timer and log out user
    if (time === 0) {
      clearInterval(timer);
      labelWelcome.textContent = 'Log in to get started';
      containerApp.style.opacity = 0;
    }

    // Decrease 1s
    time--;
  };
};

```



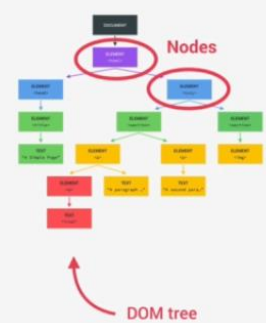
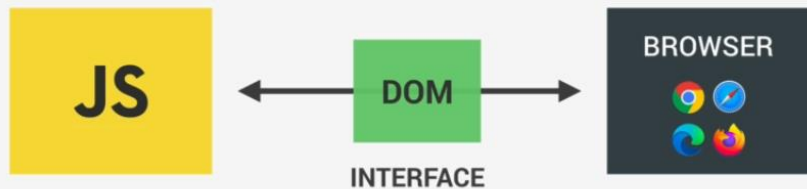
```
// Set time to 5 minutes
let time = 120;

// Call the timer every second
tick(); // wrapped in a function for starting immediately
const timer = setInterval(tick, 1000);

return timer;
};
```

### ❖ 13. Advanced DOM and Events → 4. How the DOM Really Works

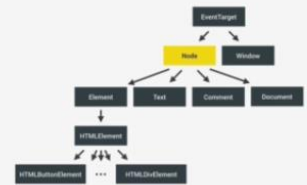
#### REVIEW: WHAT IS THE DOM?



- Allows us to make JavaScript interact with the browser;
- We can write JavaScript to create, modify and delete HTML elements; set styles, classes and attributes; and listen and respond to events;
- DOM tree is generated from an HTML document, which we can then interact with;
- DOM is a very complex API that contains lots of methods and properties to interact with the DOM tree

Application Programming Interface

```
.querySelector() / .addEventListener() / .createElement() /
.innerHTML / .textContent / .children / etc ...
```



This diagram is not a representation of any HTML document. This is just a way that different types of nodes are represented behind the scenes in the DOM API.

But anyway, now comes the really important part, because what makes all of this work is something called **inheritance**. So what is inheritance?

Well **inheritance** means that all the child types will also get access to the methods and properties of all their parent node types.

from the element type, like innerHTML, or classList or all these other methods and properties.

And besides that it will also get access to everything from the node type because that is also its parent type.

So we can think of this as though the HTML button element for example, is also an element and also a node.

For now, what I want you to understand is that a DOM API is broken up into these different types of nodes.

I also want you to understand that each of these types of nodes has access to different properties and methods and that some of them even inherit more properties and methods from their ancestors in this organization.

Now we didn't talk yet about the documents node type.

So document, which we use all the time in DOM manipulation is in fact just another type of node so it contains important methods, such as `querySelector`, `createElement` and `getElementById`. And note how `querySelector` is available on both the document and element types.

So keep this in mind because it will be important later on.

All right, and now there is just one final missing piece here because the DOM API actually needs a way of allowing all the node types to listen to events.

And remember we usually listen for events by calling the `addEventListener` method on an element or the document. So why does that actually work?

Well its because there is a special node type called `EventTarget` which is a parent of both the node type and also the window node type.

And so with this, thanks to inheritance, we can call `addEventListener` on every single type of node in the DOM API because all elements

as well as document and window, and even text and comment will inherit this method and therefore we will be able to use `addEventListener` on all of them just as if it was their own method. Now just to be clear, we do never

manually create an `EventTarget` object.

This is just an abstract type that we do not use in practice.

This all really happens behind the scenes to make all the functionality work as we expect it to work. So in a nutshell this is how the DOM API works and is structured behind the scenes. There are still some simplifications here but this is all that really matters.

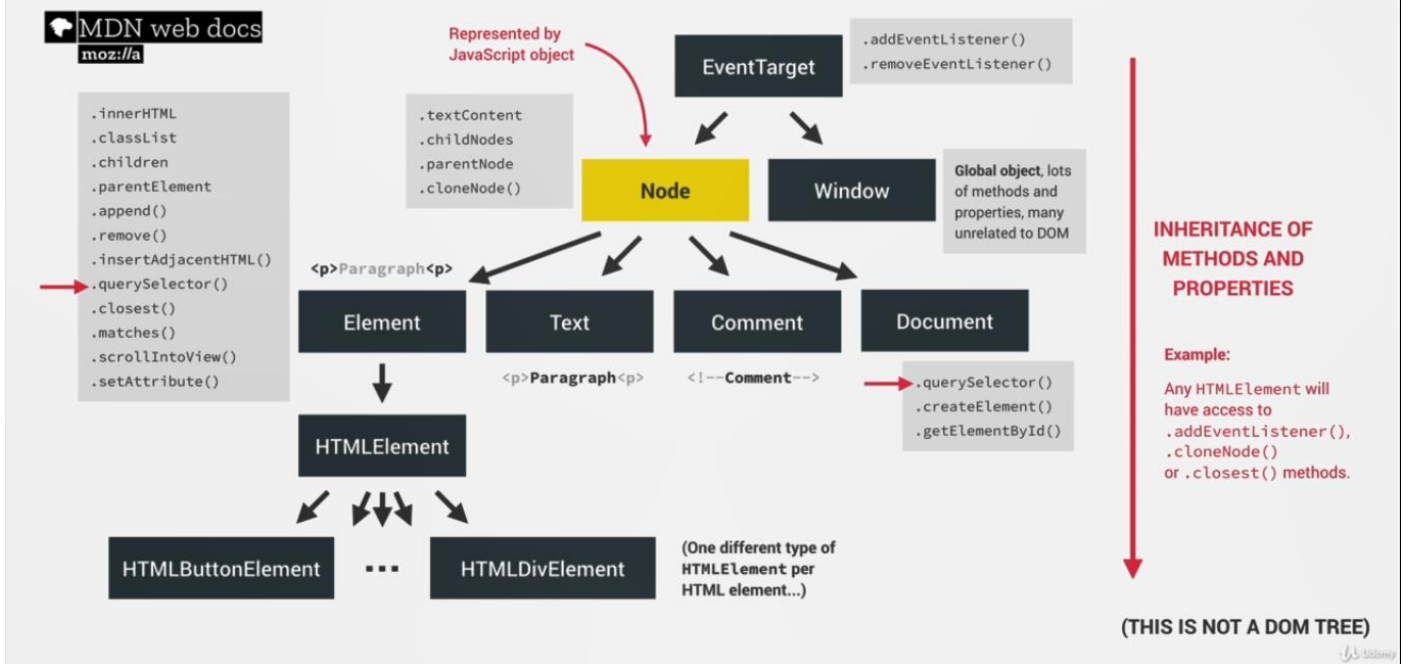
And I really wish that I could have had this diagram when I learnt JavaScript for the first time. Because I really think this helps structuring all this information in your mind. Now if you want to go even deeper than this there is still tons of material

that you can check out in the MDN documentation and if you ask me it's all really fascinating. But again, all that you need to know is really in this lecture.

It took me a lot of hours to put this one together but I think it was well worth it

and I hope you think the same. But anyway lets now move on to the practical part Of this section

# HOW THE DOM API IS ORGANIZED BEHIND THE SCENES



## ❖ 13. Advanced DOM and Events → 5. Selecting, Creating, and Deleting Elements

\*\*getElementsByTagName() and getElementsByClassName return an HTML collection is actually a so-called live collection. And that means that if the DOM changes then this collection is also immediately updated automatically.

// Selecting elements

```
console.log(document.documentElement);
console.log(document.head);
console.log(document.body);
```

```
const header = document.querySelector('.header');
const allSections = document.querySelectorAll('.section');
console.log(allSections); //node list
```

```
document.getElementById('section--1');
const allButtons = document.getElementsByTagName('button');
console.log(allButtons); //returns an HTML collection. that's different from a node list
console.log(document.getElementsByClassName('btn')); //returns an HTML collection.
that's different from a node list
```

// Creating and inserting elements

```
const message = document.createElement('div');
message.classList.add('cookie-message');
// message.textContent = 'We use cookies for improved functionality and analytics.';
```

```
message.innerHTML = 'We use cookies for improved functionality and analytics. <button  
class="btn btn--close-cookie">Got it!</button>';  
header.prepend(message); // prepending basically adds the element as the first child of  
this element  
header.append(message); // adds the element as the last child of this element
```

*// was actually only inserted once, now that's because this element here so message is now  
indeed a live element living in the DOM. And so therefore it cannot be at multiple places at  
the same time.*

if we actually wanted to insert multiple copies of the same element?

Well, in that case we actually would have to first copy the first element. So let's  
comment out this part here and say `header.append` and then instead of  
appending the message directly we first clone it. So that's `cloneNode`, and then  
we need to pass in `true` or we can pass in `true` which simply means that all the  
child elements will also be copied.

```
header.prepend(message);  
header.append(message.cloneNode(true));
```

```
header.before(message); // insert as a sibling before  
header.after(message); // insert as a sibling after
```

we don't have to select the message element again because we already have it in  
memory. So we already have it stored in a variable.

```
// Creating and inserting elements  
const message = document.createElement('div');  
message.classList.add('cookie-message');  
message.innerHTML = 'We use cookies for improved functionality and analytics. <button  
class="btn btn--close-cookie">Got it!</button>';  
header.append(message);
```

```
// Delete elements  
document
```

```
.querySelector('.btn--close-cookie')
.addEventListener('click', function () {
  // message.remove();
  message.parentElement.removeChild(message);
});
```

- ❖ **13. Advanced DOM and Events → 6. Styles, Attributes and Classes**  
**getComputedStyle(element).cssProperty** -> get all computed styles for an element even from browser

```
// Styles
message.style.backgroundColor = '#37383d';
message.style.width = '120%';

console.log(getComputedStyle(message).color); // all computed styles even from browser

message.style.height =
  Number.parseFloat(getComputedStyle(message).height, 10) + 30 + 'px';
// Number.parseFloat take just numeric part from string (ex: 45.11px =>45.11)
```

let's also work with CSS custom properties, which we usually just call CSS variables, so what I mean here is these variables that we have here, so all of these, they are called custom properties, but again, they are more like variables.

#### Define:

```
:root {
  --color-primary: #5ec576;
  --color-secondary: #ffcb03;
  --color-tertiary: #ff585f;
  --gradient-primary: linear-gradient(to top left, #39b385, #9be15d);
}
```

#### Usage:

```
.section__description {
  font-size: 1.8rem;
  color: var(--color-primary);
  margin-bottom: 1rem;
}
```

they're defined in the document root and so in JavaScript that is equivalent to the document, so basically the document element. we want to change our primary color with `setProperty()`

```
document.documentElement.style.setProperty('--color-primary', 'orangered'); //replace a property(color/with/backgroundcolor ...) with js
```

```
// Attributes
const logo = document.querySelector('.nav__logo');
console.log(logo.alt); //Bankist logo
console.log(logo.className); // nav__logo

logo.alt = 'Beautiful minimalist logo'; // set alt of logo to 'Beautiful minimalist logo'

// Non-standard
console.log(logo.designer); // undefined => "desiner" insnt a standard attr and created
custom and cant reach this way
console.log(logo.getAttribute('designer')); // Jonas
logo.setAttribute('company', 'Bankist');

console.log(logo.src); // http://127.0.0.1:5500/final/img/logo.png (.src return absolute
path)
console.log(logo.getAttribute('src')); //img/logo.png (getAttribute return relative path)

const link = document.querySelector('.nav__link--btn');
console.log(link.href); // http://127.0.0.1:5500/final/index.html# (.href return absolute
path)
console.log(link.getAttribute('href')); //(getAttribute return relative path)
```

**data-attr:** for these special attributes, they are always stored in the dataset object and indeed then down here we have that 3.0 so we use actually data attributes quite a lot when we work with the UI and especially

HTML:

```

```

JS:

```
console.log(logo.designer); // undefined – designer not a standard attr
```

```
console.log(logo.getAttribute('designer')); // Jonas
```

```
// Data attributes
```

```
console.log(logo.dataset.versionNumber);
```

```
logo.setAttribute('company', 'SONY'); // also you can use setAttribute to set attr
```

```
logo.src // return absolute URL (also for href attr)
```

```
logo.segetAttribute('src'); // return relative URL (also for href attr)
```

```
// Classes
```

```
logo.classList.add('c', 'j');
```

```
logo.classList.remove('c', 'j');
```

```
logo.classList.toggle('c');
```

```
logo.classList.contains('c'); // not includes
```

```
// Don't use override all the existing classes
```

```
logo.className = 'jonas';
```

### ❖ 13. Advanced DOM and Events → 7. Implementing Smooth Scrolling

returns a **DOMRect** object providing information about the size of an element and its position relative to the **viewport**.

```
const mySection = document.querySelector('#section--1');
```

```
console.log(mySection.getBoundingClientRect());
```

Output:

```
DOMRect {x: 0, y: 591, width: 916, height: 1652.6875, top: 591, ...}
```

1. bottom: 2243.6875

2. height: 1652.6875
3. left: 0
4. right: 916
5. top: 591
6. width: 916
7. x: 0
8. y: 591
9. [[Prototype]]: DOMRect

```
console.log(window.pageXOffset, window.pageYOffset); //Y-> distance from top of visible part of site page TO top of site page
```

```
console.log(
  document.documentElement.clientHeight, // viewport height
  document.documentElement.clientWidth // viewport width
);
```

```
btnScrollTo.addEventListener('click', function (e) {
  const s1coords = section1.getBoundingClientRect();

  //Scrolling;way1:
  window.scrollTo(
    s1coords.left + window.pageXOffset, // elem.left is relative to viewport (when you scroll the page,value is change to), so we add window.pageXOffset to calculate the scrolled part of page
    s1coords.top + window.pageYOffset
  );

  //way2:
  window.scrollTo({
    left: s1coords.left + window.pageXOffset,
    top: s1coords.top + window.pageYOffset,
    behavior: 'smooth',
  });

  //way3: (just supported in modern browsers)
  section1.scrollIntoView({ behavior: 'smooth' });
});
```



## ❖ 13. Advanced DOM and Events → 8. Types of Events and Event Handlers

an event is basically a signal that is generated by a certain DOM node and a signal means that something has happened, for example, a click somewhere or the mouse moving, or the user triggering the full screen mode and really anything of importance, that happens on our webpage, generates an event.

```
// Types of Events and Event Handlers
const h1 = document.querySelector('h1');

// way1:
h1.addEventListener('mouseenter', function (e) {
  alert('addEventListener: Great!');
});

// way2:
h1.onmouseenter = function (e) {
  alert('onmouseenter: Great!');
};

// way3:
const alertH1 = function (e) {
  alert('addEventListener: Great!');
};

h1.addEventListener('mouseenter', alertH1);
```

there are two reasons why `addEventListener` (way1) is better.

- ❖ it allows us to add multiple event listeners to the same event. So, we could do this here again and simply change the function. But if we did the same with property (way2), then the second function would basically simply override the first one.
1. The more important is that we can actually remove an event handler in case we don't need it anymore.

*\*we can use `removeEventListener` to run an eventListener **just once***

```
const alertH1 = function (e) {
  alert('addEventListener: Great!');
```

```
h1.removeEventListener('mouseenter', alertH1) // removing an eventlistener  
};  
h1.addEventListener('mouseenter', alertH1);
```

```
setTimeout(() => h1.removeEventListener('mouseenter', alertH1), 3000); // removing  
eventListener after a time
```

Another way to handle events : using property in html (don't use this old way)

```
<h1 onclick="alert('HTML alert')">
```

### ❖ 13. Advanced DOM and Events → 9. Event Propagation Bubbling and Capturing

JavaScript events have a very important property. They have a so-called capturing phase and a bubbling phase. So what does that mean? Well, let's find out. So here we have a very simple HTML document along with a dumb tree, but only for the anchor element that's represented in red here. So here we can see exactly all the parent elements of that red anchor element. And that's because we're gonna simulate what exactly happens with an event when someone clicks on that link. So maybe pause the video for a minute, and analyze this structure here. But anyway, let's now say that a click happens on the link. And as we already know, the dumb then generates a click event right away. However, this event is actually not generated at the target element. So at the element, where the event happened,

in this case, the click on the anchor element. Instead, the event is actually generated at the root of the document, so at the very top of the dumb tree. And from there, the so-called capturing phase happens, where the event then travels all the way down from the document route to the target element. And as the event travels down the tree, it will pass through every single parent element of the target element. So in our example, here, the HTML element, the body element, the section, then the paragraph, until it finally reaches its target. As soon as the event reaches the target, the target phase begins, where events can be handled right at the target. And as we already know, we do that with event listeners, such as this one. So event listeners wait for a certain event to happen on a certain element, and as soon as the event occurs, it runs the attached callback function.

In this example, it will simply create this alert window, all right? And again, this happens in the target phase. All right, now, after reaching the target, the event then actually travels all the way up to the document route again, in the so-called bubbling phase. So we say that events bubble up from the target to the document route. And just like in the capturing phase, the event passes through all its parent elements, and really just the parents, so not

through any sibling elements. So as an event travels down and up the tree, they pass through all the parent elements, but not through any sibling element. But now you might be wondering why is this so important?

Why are we learning about all this detail? Well, it is indeed very important because basically, it's as if the event also happened in each of the parent elements. So again, as the event bubbles through a parent element, it's as if the event had happened right in that very element. What this means is that if we attach the same event listener, also for example, to the section element, then we would get the exact same alert window for the section element as well. So we would have handled the exact same event twice, once at its target, and once at one of its parent elements. And this behavior will allow us to implement really powerful patterns, as we will see throughout the rest of the section.

So this really is very, very important to understand. Now by default, events can only be handled in the target, and in the bubbling phase. However, we can set up event listeners in a way that they listen to events in the capturing phase instead. Also, actually not all types of events that do have a capturing and bubbling phase. Some of them are created right on the target element, and so we can only handle them there. But really, most of the events do capture and bubble such as I described it here in this lecture. We can also say that events propagate, which is really what capturing and bubbling is.

It's events propagating from one place to another. All right, so I hope that all of this made sense, and so let's now actually see this in action in the next video.

### BUBBLING AND CAPTURING

The diagram illustrates the three phases of event propagation:

- 1 CAPTURING PHASE:** An event starts at the target element and travels up through its parents. In the example, it goes from the `<a>` element to the `<p>` element, then to the `<section>` element, then to the `<body>` element, and finally to the `<html>` element.
- 2 TARGET PHASE:** The event is handled at the target element (`<a>`).
- 3 BUBBLING PHASE:** The event travels back down through the DOM tree from the target element back up to the root. In the example, it goes from `<a>` to `<p>`, then to `<section>`, then to `<body>`, and finally to `<html>`.

Code snippets for the bubbling phase:

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

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

❖ **13. Advanced DOM and Events → 10. Event Propagation in Practice**  
**Review the video**

### ❖ 13. Advanced DOM and Events → 10. Event Propagation in Practice

at event listener here, it's only listening for events in the bubbling phase, but not in the capturing phase.

So that is the default behavior of the add event listener method, and the reason for that is that the capturing phase is usually irrelevant for us. It's just not that useful. Now, on the other hand, the bubbling phase can be very useful for something called event delegation.

if we really do want to catch events during the capturing phase, we can define a third parameter in the addEventListener function.

For example, we can set the third parameter to true or false. And so in this case where this used capture parameter is set to true, the event handler will no longer listen to bubbling events, but instead, to capturing events.

```
// Event Propagation in Practice
const randomInt = (min, max) =>
  Math.floor(Math.random() * (max - min + 1) + min);
const randomColor = () =>
  `rgb(${randomInt(0, 255)},${randomInt(0, 255)},${randomInt(0, 255)})`;

document.querySelector('.nav__link').addEventListener('click', function (e) {
  this.style.backgroundColor = randomColor();
  console.log('LINK', e.target, e.currentTarget);
  console.log(e.currentTarget === this); //true

  // Stop propagation
  e.stopPropagation(); //in general not a good idea
});

document.querySelector('.nav__links').addEventListener('click', function (e) {
  this.style.backgroundColor = randomColor();
  console.log('CONTAINER', e.target, e.currentTarget);
});

document.querySelector('.nav').addEventListener('click', function (e) {
  this.style.backgroundColor = randomColor();
  console.log('NAV', e.target, e.currentTarget);
});
```

### ❖ 13. Advanced DOM and Events → 11. Event Delegation Implementing Page Navigation

Review the video and take notes

## ❖ 13. Advanced DOM and Events → 12. DOM Traversing

So Dom traversing is basically walking through the Dom. Which means that we can select an element

based on another element. And this is very important because sometimes we need to select elements

relative to a certain other element. For example, a direct child or a direct parent element.

Or sometimes we don't even know the structure of the Dom at runtime. And in all these cases, we need Dom traversing.

```
// DOM Traversing
const h1 = document.querySelector('h1'); //this here indeed selects all the elements with
the highlight class that are children of the h1 element.

// Going downwards: child
console.log(h1.querySelectorAll('.highlight'));
console.log(h1.childNodes); //really gives us, every single node of every single type that
there exists.
console.log(h1.children); // this then gives us an HTMLCollection which remembers is a live
collection, so it's updated, and so here we indeed only get the elements that are actually
inside of the h1. but this one works only for direct children.
h1.firstElementChild.style.color = 'white'; //first Child of element
h1.lastElementChild.style.color = 'orangered'; //last Child of element

// Going upwards: parents
console.log(h1.parentNode);
console.log(h1.parentElement);

h1.closest('.header').style.background = 'var(--gradient-secondary)'; //the closest parent
element that has this class. basically being the opposite of querySelector. So both receive a
query string as an input but querySelector, finds children

h1.closest('h1').style.background = 'var(--gradient-primary)';

// Going sideways: siblings
console.log(h1.previousElementSibling);
console.log(h1.nextElementSibling);

console.log(h1.previousSibling);
console.log(h1.nextSibling);

console.log(h1.parentElement.children); // selecting all siblings
```

```
//practice sample
[...h1.parentElement.children].forEach(function (el) {
  if (el !== h1) el.style.transform = 'scale(0.5)'; //scale all h1 siblings to 50%
});
```

### ❖ 13. Advanced DOM and Events → 13. Building a Tabbed Component

doing this (add eventlistener to many item) is a bad practice because what if we had like 200 tabs?

Then we would have 200 copies of this exact callback function here and that would simply slow down the page

```
const tabs = document.querySelectorAll('.operations__tab');
tabs.forEach(t => t.addEventListener('click', () => console.log('TAB')));
```

use **events delegation** to develop this.

remember that for event delegation, we need to attach the event handler on the common parent element of all the elements that we're interested in.

```
// Tabbed component
const tabs = document.querySelectorAll('.operations__tab');
const tabsContainer = document.querySelector('.operations__tab-container');
const tabsContent = document.querySelectorAll('.operations__content');

tabsContainer.addEventListener('click', function (e) {
  const clicked = e.target.closest('.operations__tab');

  // Guard clause old way
  // if (clicked) {
  //   clicked.classList.add('operations__tab--active');
  // }

  // Guard clause (modern way)
  if (!clicked) return;

  // Remove active classes
  tabs.forEach(t => t.classList.remove('operations__tab--active'));
  tabsContent.forEach(c => c.classList.remove('operations__content--active'));

  // Activate tab
  clicked.classList.add('operations__tab--active');
```

```
// Activate content area
document
.querySelector(`.operations__content--${clicked.dataset.tab}`)
.classList.add('operations__content--active');
});
```

❖ 13. Advanced DOM and Events → 14. Passing Arguments to Event Handlers

*\* mouseover is actually similar to mouseenter, with the big difference that mouseenter does not bubble*

**Review the video**

❖ 13. Advanced DOM and Events → 16. A Better Way The Intersection Observer API

**Review the video**

❖ 13. Advanced DOM and Events → 17. Revealing Elements on Scroll

**Review the video**

❖ 13. Advanced DOM and Events → 18. Lazy Loading Images

❖ **Review the video**

❖ 13. Advanced DOM and Events → 19. Building a Slider Component Part 1

❖ **Review the video**

❖ 13. Advanced DOM and Events → 20. Building a Slider Component Part 2

❖ **Review the video**

❖ 13. Advanced DOM and Events → 21. Lifecycle DOM Events

```
////////////////////////////////////
```

```
// Lifecycle DOM Events
```

```
// "DOMContentLoaded" event is fired by the document as soon as the HTML is
completely parsed, which means that the HTML has been downloaded and been
converted to the DOM tree.
```

```
document.addEventListener('DOMContentLoaded', function (e) {
  console.log('HTML parsed and DOM tree built!', e);
});
```

```
// "Load" fired by the window. As soon as not only the HTML is parsed, but also all the
images and external resources like CSS files are also loaded.
```

```

window.addEventListener('load', function (e) {
  console.log('Page fully loaded', e);
});

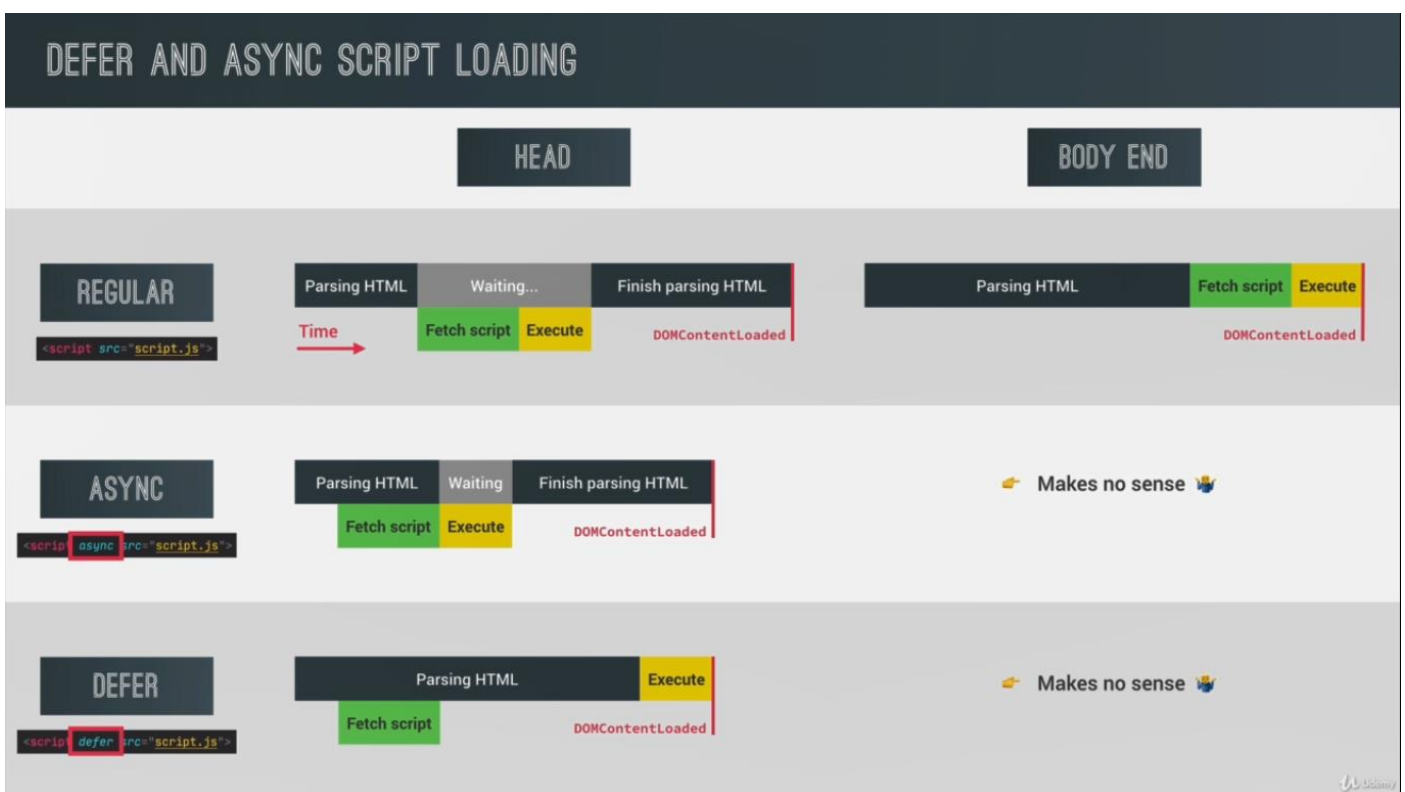
```

*//"beforeunload" fired right before close the window(current tab) dont use this- because a message like this is of course pretty intrusive*

```

window.addEventListener('beforeunload', function (e) {
  e.preventDefault();
  console.log(e);
  e.returnValue = "";
});

```



❖ 13. Advanced DOM and Events → 22. Efficient Script Loading defer and async

So here is what we already know about the three strategies, and also the three loading diagrams that we analyzed in the last slide. But let's now compare the async

and defer attributes a little bit better.

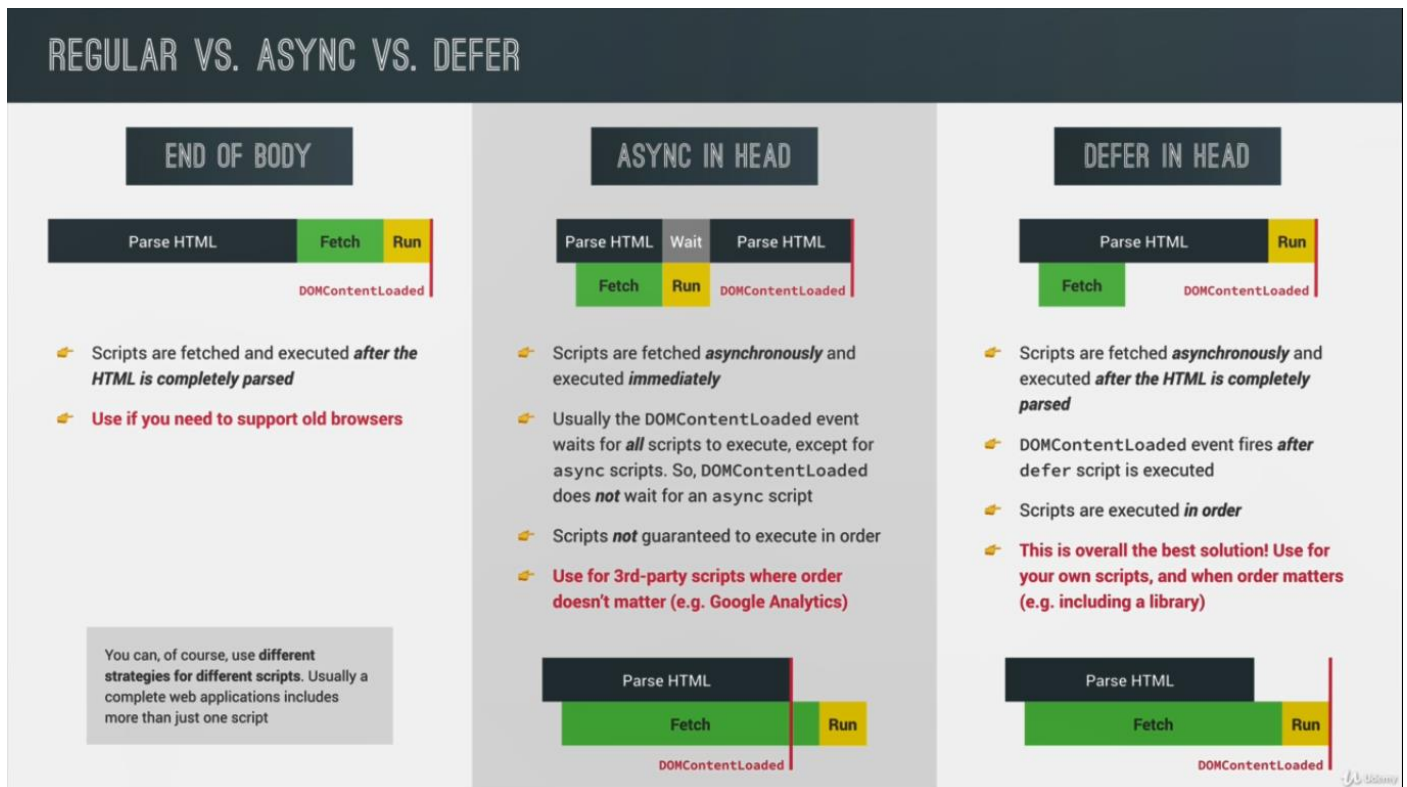
So one important thing about loading an async script is that the DOM content loaded event



will not wait for the script to be downloaded and executed. So usually, DOM content loaded, waits for all scripts to execute. But scripts loaded with `async` are an exception. So with `async`, DOM content loaded is fired off as soon as the HTML finishes parsing. And this might actually happen when a big script takes a long time to load, like in this example. So notice how the DOM content loaded event appears right after HTML parsing in both these diagrams. Now, using `defer`, on the other hand, forces, the DOM content loaded event to only get fired after the whole script has been downloaded and executed. And so this is the more traditional way that this event works.

Another very important aspect is that `async` scripts are not guaranteed to be executed in the exact order that they are declared in the code. So the script that arrives first gets executed first. On the other hand, by using `defer`, that is not the case. So using the `defer` attribute guarantees that the scripts are actually executed in the order that they are declared or written in the code. And that is usually what we want to happen. So in conclusion, and also keeping in mind, what we learned in the last slide, using `defer` in the HTML head is overall the best solution. So you should use it for your own scripts. And for scripts where the order of execution is important. For example, if your script relies on some third party library that you need to include, you will include that library before your own script, so that your script can then use the library's code. And in this case, you have to use `defer` and not `async`. Because `defer` will guarantee the correct order of execution. Now, for third party scripts, where the order does not matter, for example, an analytics software like Google Analytics, or an ad script, or something like that, then in this case, you should totally use `async`. So for any code that your own code will not need to interact with `async` is just fine. So it's a good use case for this kind of scripts. And I'm saying this because, of course, you can use different loading strategies for different scripts in your web application or website. Okay, now, what's important to note is that only modern browsers support `async` and `defer`. And they will basically get ignored in older browsers. So if you need to support all browsers, then you need to put your script tag at the end of the body and not in the head. That's because this is actually not a JavaScript feature, but an HTML5 feature. And so you can't really work around this limitation, like you can do with modern JavaScript features by transpiling, or poly-filling.

Okay, now, this should give you a pretty good idea about different ways of loading JavaScript scripts.



## ❖ 14. Object-Oriented Programming (OOP) With JavaScript → 3. What is Object-Oriented Programming

this section is about object-oriented programming, and this lecture is gonna be a very general, high level overview of this programming paradigm. So we're gonna talk about what object-oriented programming is, how it works in general, and about its four fundamental principles. So this is gonna be a really important and valuable lecture. And so let's get started. So, first of all, what is object-oriented programming?

Well, object-oriented programming, or OOP in short, is a programming paradigm that is based

on the concept of objects. And paradigm simply means the style of the code, so the how we write and organize code. Now we use objects to model, so to describe aspects of the real world, like a user or a to-do list item, or even more abstract features like an HTML component or some kind of data structure. Now, as we already know, objects can contain data, which we call properties, and also code, which we call methods.

So we can say that by using objects, we pack all the data and the corresponding behavior all into one big block. So again, that's data and corresponding behavior.

And this makes it super easy to act directly on the data. And speaking of blocks, that's exactly what objects are supposed to be. So in OOP, which is the acronym that I'm gonna use instead of object-oriented programming. Okay.

So in OOP objects are self-contained pieces of code or blocks of code, like small applications on their own.

And we then use these objects as building blocks of our applications and make objects interact with one another. Now these interactions happen through a so-called public interface, which we also call API. This interface is basically a bunch of methods

that a code outside of the objects can access and that we use to communicate with the object. Okay. So let's take a breath here because this all sounds kind of abstract, right?

But don't worry. It will make more sense once we start developing these concepts using code throughout this section. But anyway, why does OOP actually exist?

Well, this paradigm was developed with the goal of organizing code, so to make it more flexible and easier to maintain. So before OOP, we might have a bunch of codes

gathered across multiple functions, or even in the global scope without any structure. And this particular like crazy style of code

is what we usually call spaghetti code and spaghetti code makes it very hard

to maintain large code bases and let alone, add new functionalities to it. So the idea of OOP was basically created as a solution to this problem.

## WHAT IS OBJECT-ORIENTED PROGRAMMING? (OOP)


**OOP**

**Data**

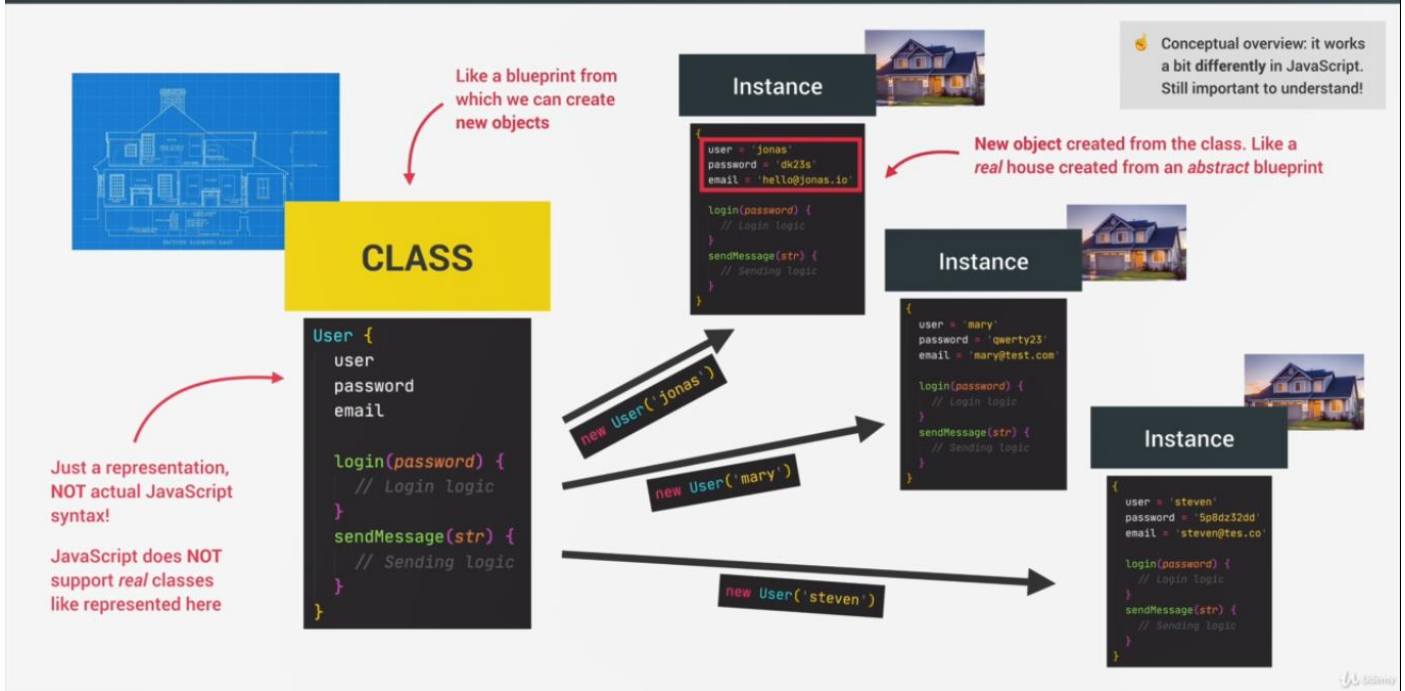
```
const user = {  
  user: 'jonas',  
  password: 'dk23s',  
  login(password) {  
    // Login logic  
  },  
  sendMessage(str) {  
    // Sending logic  
  }  
}
```

**Behaviour**

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



# CLASSES AND INSTANCES (TRADITIONAL OOP)



well, how do we actually plan and design a house?

And that's of course a very good question. Now the answer is, as you can imagine, not straightforward.

So there is not a single correct way of designing classes. There are, however, four fundamental principles that can guide us toward a good class implementation. And these principles are abstraction, encapsulation, inheritance, and polymorphism.

# PRINCIPLE 1: ABSTRACTION

Abstraction

Encapsulation

Inheritance

Polymorphism

```
Phone {
  charge
  volume
  voltage
  temperature

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



Real phone



Abstracted phone

```
Phone {
  charge
  volume

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

Details have been abstracted away

Do we really need all these low-level details?

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



# PRINCIPLE 2: ENCAPSULATION

Abstraction

Encapsulation

Inheritance

Polymorphism

NOT accessible from outside the class!

STILL accessible from within the class!

STILL accessible from within the class!

NOT accessible from outside the class!

```
User {
  user
  private password
  private email

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

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

## WHY?

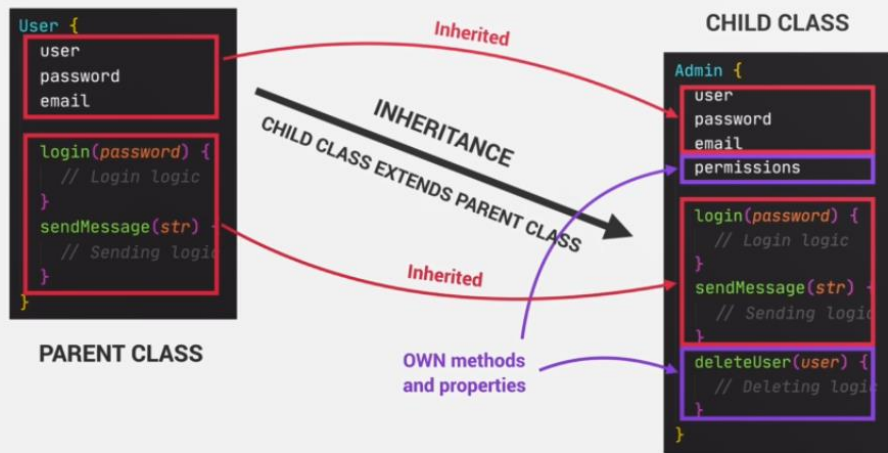
- 👉 Prevents external code from accidentally manipulating internal properties/state
- 👉 Allows to change internal implementation without the risk of breaking external code

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



# PRINCIPLE 3: INHERITANCE

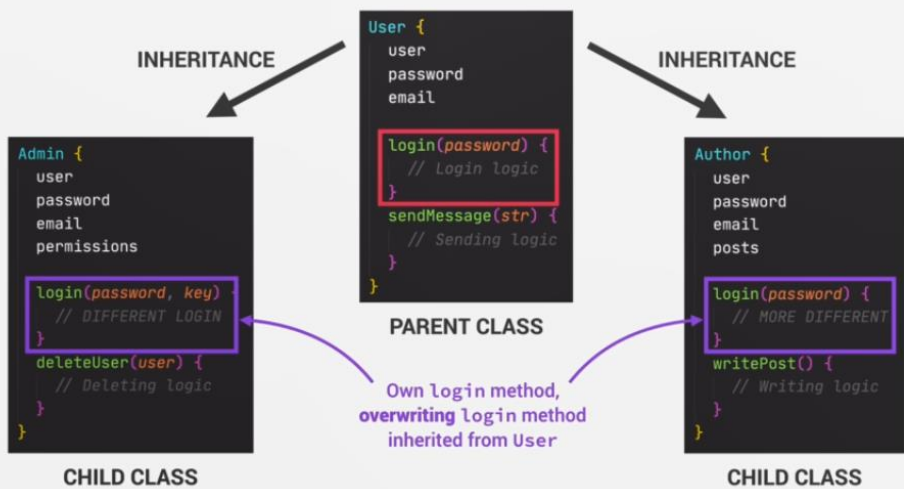
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism



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

# PRINCIPLE 4: POLYMORPHISM

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism



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

❖ **14. Object-Oriented Programming (OOP) With JavaScript** → **4. OOP in JavaScript**  
 prototypical inheritance means that all objects that are linked to a certain prototype object can use the methods and properties that are defined on that prototype. So basically, objects inherit methods and properties from the prototype which is the reason why this mechanism is called prototypical inheritance. Just note that this inheritance is actually different from the inheritance that we talked about in the last lecture. So that was one class inheriting from another class. But in this case, it's basically an instance inheriting from a class.

### OOP IN JAVASCRIPT: PROTOTYPES

#### "CLASSICAL OOP": CLASSES

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

#### OOP IN JS: PROTOTYPES

- ☛ Objects are **linked** to a prototype object;
- ☛ **Prototypal inheritance:** The prototype contains methods (behavior) that are **accessible to all objects linked to that prototype;**
- ☛ Behavior is **delegated** to the linked prototype object.

#### Example: Array

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

MDN web docs

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

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

Therefore, all arrays have access to the map method!

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

in JavaScript there are actually three different ways of doing all this: the constructor function technique, ES6 classes and also the Object.create().

## 3 WAYS OF IMPLEMENTING PROTOTYPAL INHERITANCE IN JAVASCRIPT

🤔 "How do we actually create prototypes? And how do we link objects to prototypes? How can we create new objects, without having classes?"

### 1 Constructor functions

- 👉 Technique to create objects from a function;
- 👉 This is how built-in objects like Arrays, Maps or Sets are actually implemented.

### 2 ES6 Classes

- 👉 Modern alternative to constructor function syntax;
- 👉 "Syntactic sugar": behind the scenes, ES6 classes work **exactly** like constructor functions;
- 👉 ES6 classes do **NOT** behave like classes in "classical OOP" (last lecture).

### 3 Object.create()

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

👉 The 4 pillars of OOP are still valid!

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

## ❖ 14. Object-Oriented Programming (OOP) With JavaScript → 5. Constructor Functions and the new Operator

a constructor function is actually a completely normal function. The only difference between a regular function, and a function that we call constructor function, is that we call a constructor function with the new operator.

\*\* Use the first letter capital for constructor name

\*\* Use "new" keyword to call constructor function

\*\* You should never create a method inside of a constructor function

*(That's because imagine we were gonna create a hundred or thousands or even tens of thousands*

*of person objects using this constructor function. Then what would happen, is that each of these objects would carry around this function here. So if we had a thousand objects, we would essentially create a thousand copies of this function. And so that would be terrible for the performance of our code.*

*But instead to solve this problem, we are gonna use prototypes and prototype inheritance)*

```
// Constructor Functions and the new Operator
const Person = function (firstName, birthYear) {
  // Instance properties
  this.firstName = firstName;
  this.birthYear = birthYear;
}
```



```

    // ** Never to this!
    // this.calcAge = function () {
    //   console.log(2037 - this.birthYear);
    // };
};

const jonas = new Person('Jonas', 1991);
console.log(jonas); // Person {firstName: 'Jonas', birthYear: 1991}

// four step happens behind the scenes when using "New" keyword :
// 1. New {} is created
// 2. function is called, this = {}
// 3. {} linked to prototype
// 4. function automatically return {}

```

#### ❖ 14. Object-Oriented Programming (OOP) With JavaScript → 6. Prototypes

we talked about prototypes, prototypal inheritance and delegation earlier already.

But how does all of that actually work? Well, it can be summarized like this.

So, first each and every function in JavaScript automatically has a property called prototype.

And that includes, of course, constructor functions. Now every object that's created

by a certain constructor function will get access to all the methods and properties

that we define on the constructors prototype property.

```

// Prototypes
const Person = function (firstName, birthYear) {
  // Instance properties
  this.firstName = firstName;
  this.birthYear = birthYear;
};

console.log(Person.prototype); // is empty now

```

so let's now actually add a method to this prototype property.

```

Person.prototype.calcAge = function () {
  console.log(2037 - this.birthYear);
};

console.log(Person.prototype);
// output (we have calcAge added in prototype):

```

```
// {constructor: f}
// calcAge: f ()
// constructor: f (firstName, birthYear)
// [[Prototype]]: Object
```

each object created by this constructor function will now get access to all the methods of this prototype property. And so of course, also to calcAge()

```
const jonas = new Person('Jonas', 1991);
console.log(jonas);

jonas.calcAge();
```

Never do this (in this way create a copy of calcAge() for every single instance that created by Person) , Instead of this:

```
const Person = function (firstName, birthYear) {
  // Instance properties
  this.firstName = firstName;
  this.birthYear = birthYear;

  // Never do this
  this.calcAge = function () {
    console.log(2037 - this.birthYear);
  };
};
```

Do this: (outside of constructor use [constructor Name]. Prototype.[method Name] = function(){---})

```
const Person = function (firstName, birthYear) {
  // Instance properties
  this.firstName = firstName;
  this.birthYear = birthYear;
```

```

};

// we have just 1 calcAge() for all instances
Person.prototype.calcAge = function () {
  console.log(2022 - this.birthYear);
};

const user = new Person('ali', 1993);
user.calcAge();

```

each object has a special property called: `__proto__`

```
console.log(jonas.__proto__);
```

prototype of the Jonas object is essentially the prototype property of the constructor function.

```
console.log(jonas.__proto__ === Person.prototype);
```

person dot prototype here is actually not the prototype of person.

But instead, it is what's gonna be used as the prototype of all the objects that are created with the person constructor function.

And there are actually other built in methods that we can use to prove this. So on any object, for example, object dot prototype, we can test if this is a prototype of another object.

```

console.log(Person.prototype.isPrototypeOf(jonas)); // true
console.log(Person.prototype.isPrototypeOf(matilda)); // true
console.log(Person.prototype.isPrototypeOf(Person)); // false

```

“Person.prototype” is bad naming in JS and Probably shouldn't be called prototype but instead something like “.prototypeOfLinkedObjects” or something like this.

where does this proto, property here, on the Jonas object actually come from? Well, remember the new operator that we talked about before, well, it contains this step number 3 which links the empty new object to the prototype.

```

// 1. New {} is created
// 2. function is called, this = {}
// 3. {} linked to prototype

```

```
// 4. function automatically return {}
```

And so basically, it is this step number 3 which will create this `__proto__` property. So it creates this `__proto__` property and it sets its value to the prototype property of the function that is being called.

```
console.log(jonas.__proto__ === Person.prototype);
```

gain, it sets the `proto` property on the object to the `prototype` property of the constructor function (`Person.prototype`). And so this is how JavaScript knows internally that the Jonas object is connected to `person dot prototype`.

```
console.log(jonas);
```

Output:

1. `Person {firstName: 'Jonas', birthYear: 1991}`
  1. `birthYear: 1991`
  2. `firstName: "Jonas"`
  3. `__Prototype__: Object`
    1. `calcAge: f ()`
    2. `constructor: f (firstName, birthYear)`
    3. `__Prototype__: Object`

```
Person.prototype.species = 'Homo Sapiens';  
console.log(jonas.species, matilda.species); // Jonas and Matilda objects will inherit so they  
will get access to "species" property from the prototype. But not directly  
console.log(jonas.hasOwnProperty('species')); // false
```

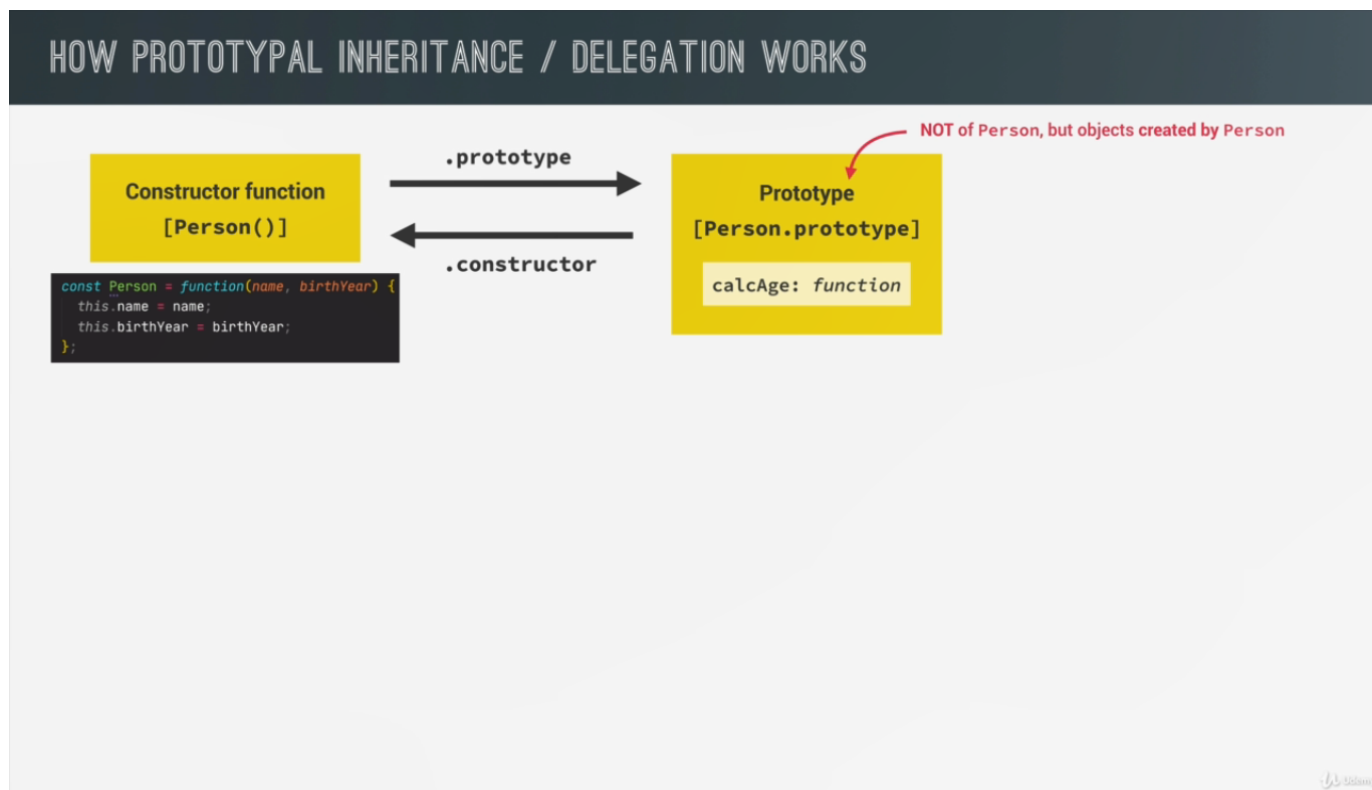
## ❖ 14. Object-Oriented Programming (OOP) With JavaScript → 7. Prototypal Inheritance and The Prototype Chain

Let's now consolidate the knowledge that we got over the last two videos in a nice diagram that brings everything together that we know so far. And everything starts with the person the constructor function

that we've been developing. Now, this constructor function has a prototype property

which is an object and inside that object, we defined the calcAge method and person dot prototype itself actually also has a reference back to person which is the constructor property. So, essentially person dot prototype dot constructor

is gonna point back to person itself. Now remember, person dot prototype is actually not the prototype of person but of all the objects that are created

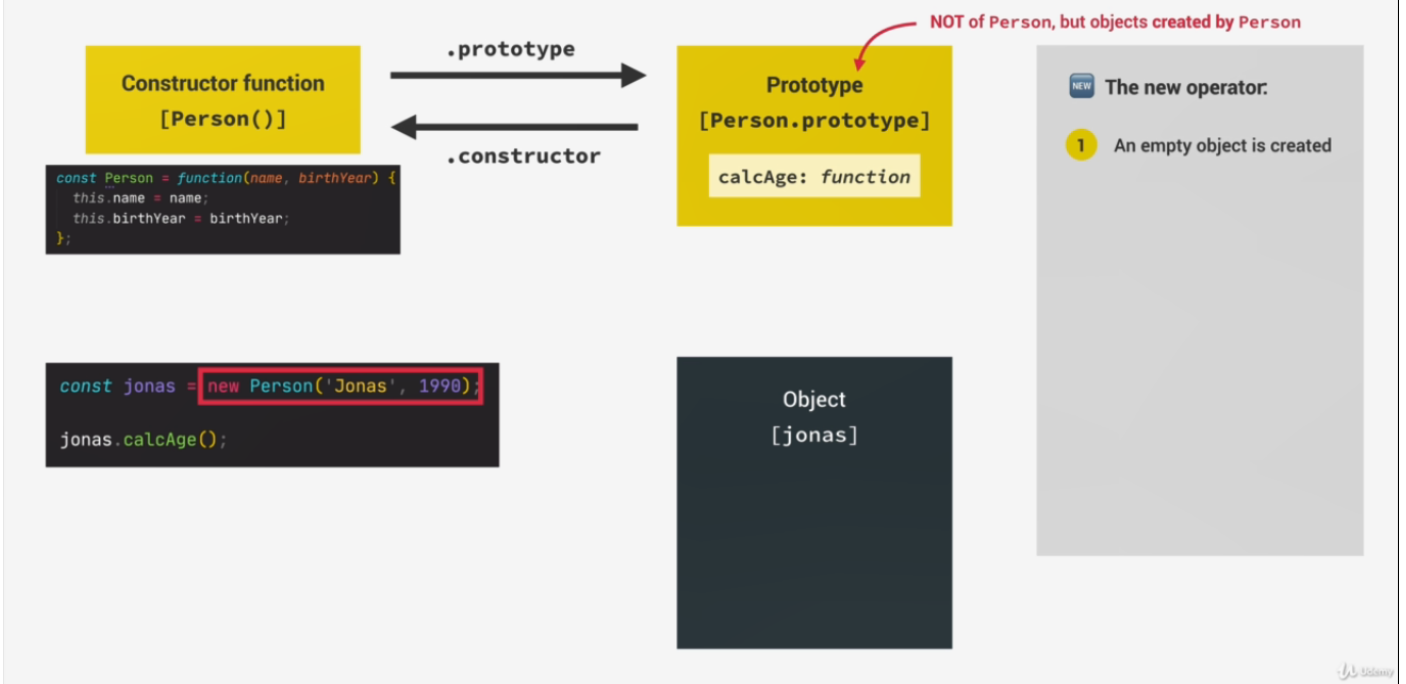


through the person function

and speaking of the created objects let's now actually analyze how an object is created

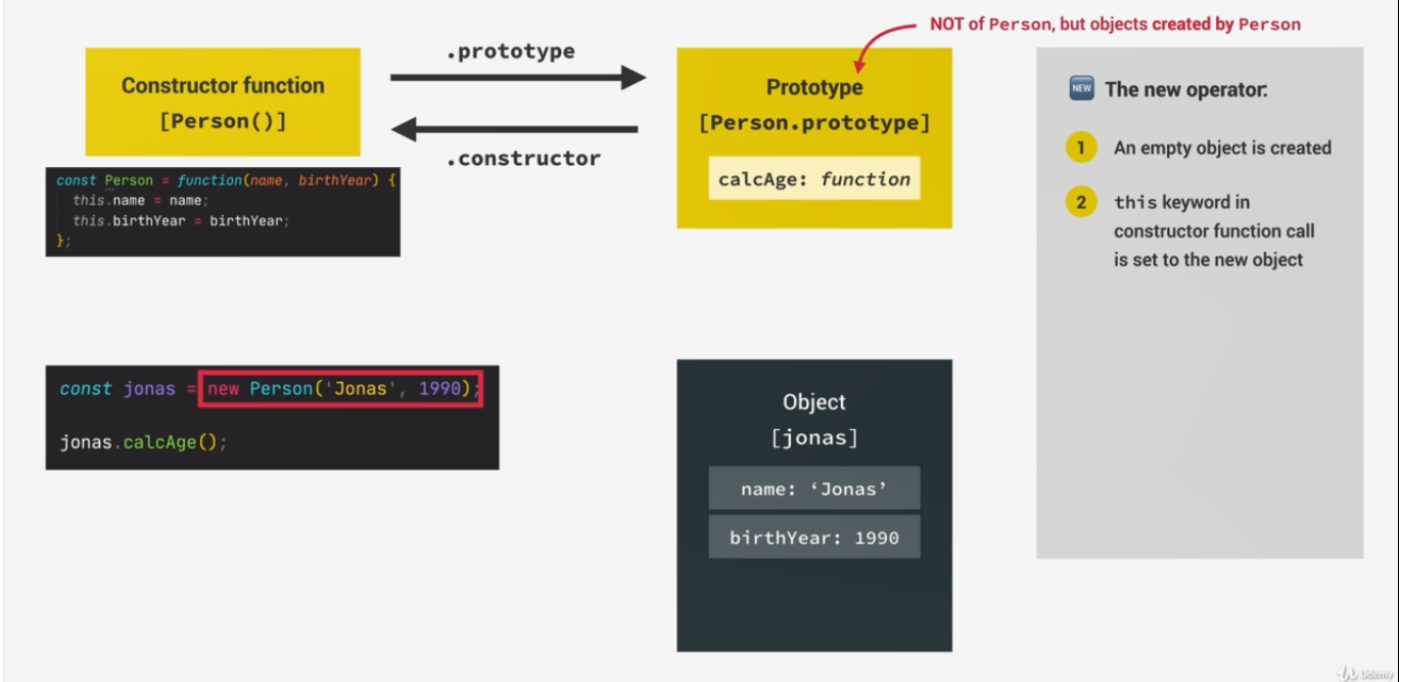
using the new operator and the constructor function. So, when we call a function, any function with the new operator the first thing that is gonna happen is that a new empty object is created instantly.

# HOW PROTOTYPAL INHERITANCE / DELEGATION WORKS

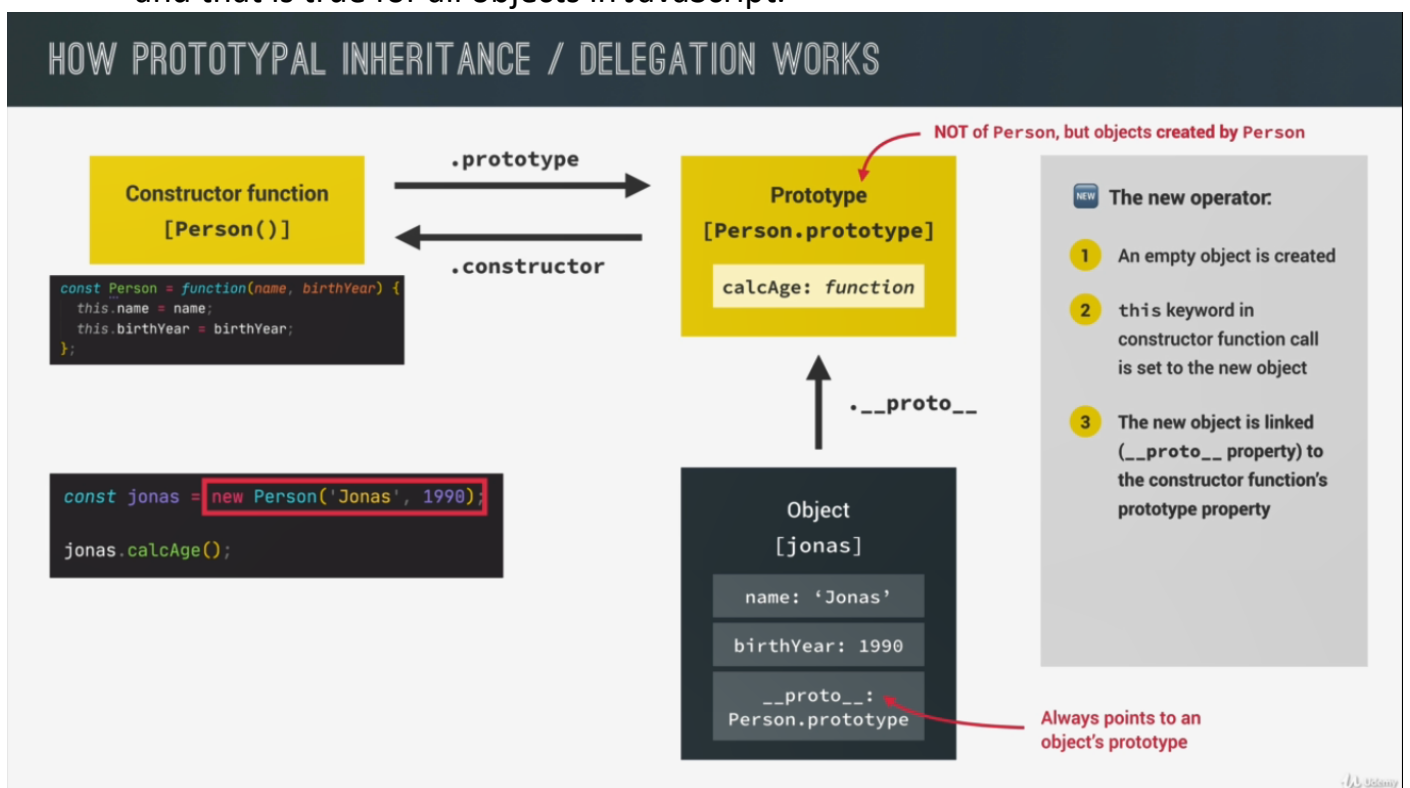


Then the `this` keyword and the function call is set to the newly created object. So, inside the function's execution context `this` is now the new empty object and that's why in the function's code we set the `name` and `birthYear` properties on the `this` keyword because doing so will ultimately set them on the new object.

# HOW PROTOTYPAL INHERITANCE / DELEGATION WORKS

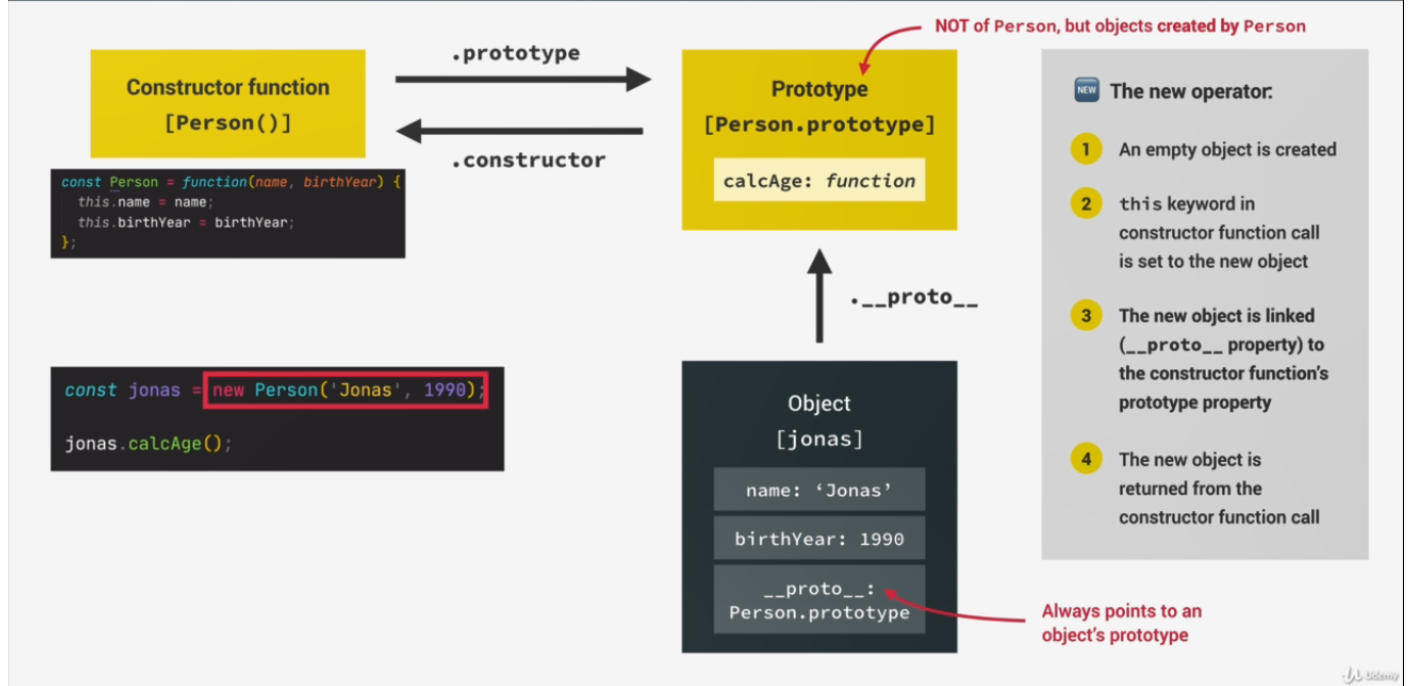


So next comes the magical step. So now the new object is linked to the constructor functions prototype property. So in this case, person dot prototype. And this happens internally by adding the underscore, underscore protal property to the new object. So, person dot prototype is now the new objects prototype which is denoted in the underscore, underscore proto property of Jonas. So again, underscore proto always points to an object prototype and that is true for all objects in JavaScript.



And finally the new object is automatically returned from the function unless we explicitly return something else. But in a constructor function like person we usually never do that.

## HOW PROTOTYPAL INHERITANCE / DELEGATION WORKS



Okay, and with this the result of the new operator and the person constructor function is a new object that we just created programmatically and that is now stored in the Jonas variable and this whole process that I just explained is how it works with function constructors and also with ES6 classes but not with the object dot create syntax that we're gonna use later. So just keep this in mind once we read the object dot create lectures. But anyway, why does this work this way and why is this technique so powerful and useful? and to answer that let's move on to the next line of code. So, here we are attempting to call the calcAge function on the jonas object. However, JavaScript can actually not find the calcAge function directly in the jonas object, right? It is simply not there and we already observed this behavior in the last lecture. Remember that? So, what happens now in this situation? Well, if a property or a method cannot be found in a certain object JavaScript will look into its prototype and there it is.



So there is the calcAge function that we were looking for and so JavaScript will simply use this one. That's how the calcAge function can run correctly

and return a result. And the behavior that we just described is what we already called prototypal inheritance or delegation.

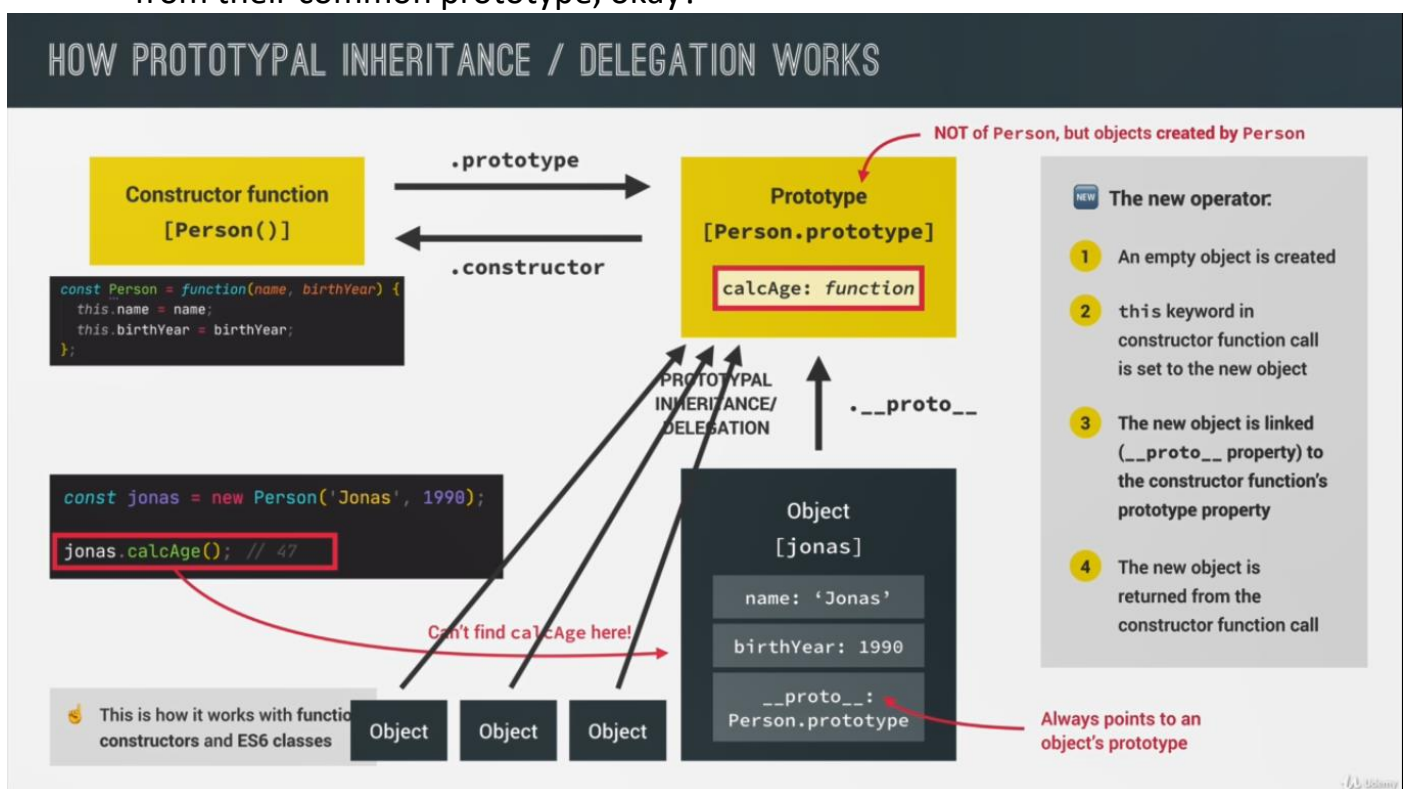
So the jonas object inherited the calcAge method from its prototype or in other words it delegated the calcAge functionality to its prototype. Now, the beauty of this

is that we can create as many person objects as we like and all of them will then inherit this method.

So we can call this calcAge method on all the person objects without the method being directly attached to all the objects themselves and this is essential for code performance.

Just imagine that we had a 1,000 objects in the code and if all of them would have to carry the calcAge function around then that would certainly impact performance.

So instead, they can all simply use the calcAge function from their common prototype, okay?



So that makes sense, right? Now the fact that Jonas is connected to a prototype and the ability of looking up methods and properties in a prototype is what we call the prototype chain. So the jonas object and its prototype basically form a prototype chain but actually the prototype chain does not end here.

So let's understand the prototype chain a bit better by zooming out and looking at the whole picture. So, here is the diagram that we already had with the person the function constructor and its prototype property and to jonas object linked to its prototype via the underscore proto property, so nothing new yet. But now let's remember that person dot prototype itself is also an object and all objects in JavaScript have a prototype, right?

Therefore, person dot prototype itself must also have a prototype.

And the prototype of person dot prototype is object dot prototype.

Why is that? Well, person dot prototype is just a simple object

which means that it has been built by the built in object constructor function and this is actually the function that is called behind the scenes whenever we create an object literal. So just an object simply with curly braces.

So essentially the curly braces are just like a shortcut to writing new object, okay?

But anyway, what matters here is that person dot prototype itself

needs to have a prototype and since it has been created

by the object constructor function its prototype is gonna be object dot prototype.

It's the same logic as with the jonas object. So, since jonas has been built by a person,

person dot prototype is the prototype of Jonas, all right?

Now this entire series of links between the objects is what is called the prototype chain

and object dot prototype is usually the top of the prototype chain

which means that its prototype is null. So its underscore proto property will simply point to null

which then marks the end of the prototype chain. So in a certain way the prototype chain

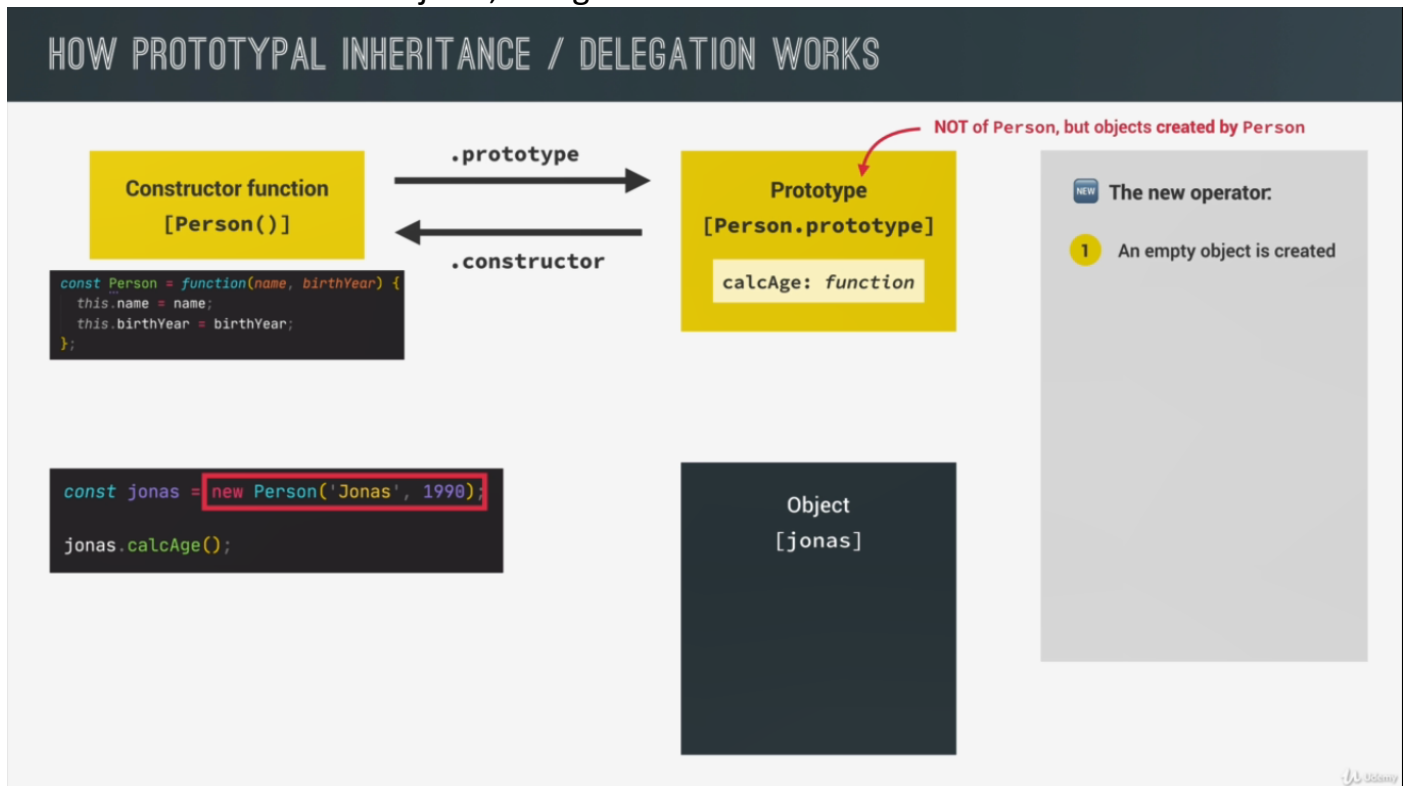
is very similar to the scope chain but with prototypes.

So, in the scope chain whenever JavaScript can find a certain variable

in a certain scope, it looks up into the next scope and a scope chain

and tries to find the variable there. On the other hand in the prototype chain

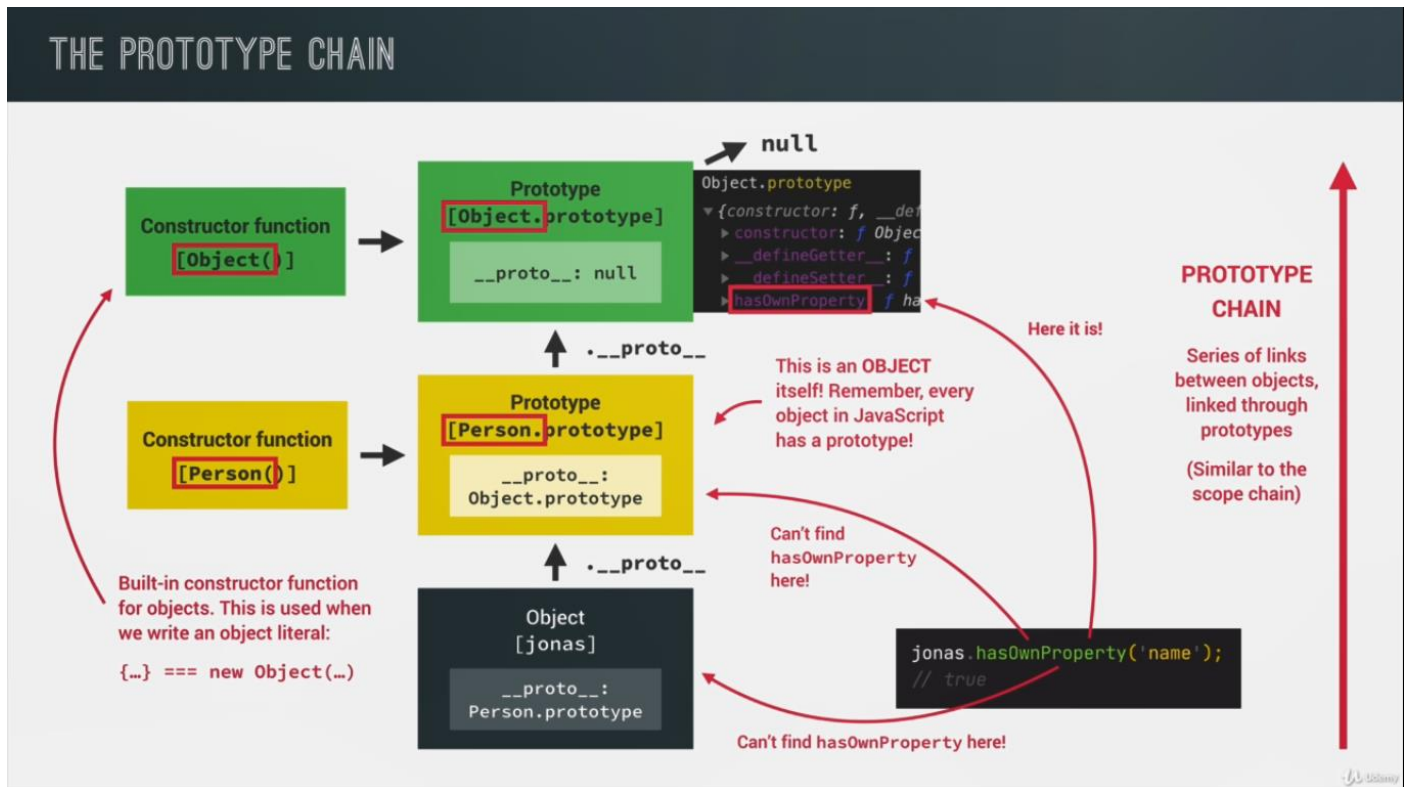
whenever JavaScript can find a certain property or method in a certain object it's gonna look up into the next prototype in the prototype chain and see if it can find it there, okay? So again the prototype chain is pretty similar to the scope chain but instead of working with scopes, it works with properties and methods in objects, all right?



And now let's actually see another example of a method lookup. To do that we call the has own property method on the jonas object. So, just like in the previous slide, JavaScript is gonna start by trying to find the called method on the object itself. But of course it can't find the has own property method on Jonas. So, according to how the prototype chain works, it will then look into its prototype which is person dot prototype. Now, we didn't define any has own property method there either and so a JavaScript is not gonna find it there and so therefore it will move up even further in the prototype chain and now look into object dot prototype and object dot prototype does actually contain a bunch

of built in methods and has own property is one of them. Great, so JavaScript can then take this one and run it on the jonas object as if has own property had been defined directly on Jonas.

And remember the method has not been copied to the jonas object. Instead, it simply inherited the method from object dot prototype through the prototype



chain.

## ❖ 14. Object-Oriented Programming (OOP) With JavaScript → 8. Prototypal Inheritance on Built-In Objects

```
// Object.prototype (top of prototype chain)
console.log(jonas.__proto__.__proto__);
console.log(jonas.__proto__.__proto__.__proto__); // null (in this case)
console.dir(Person.prototype.constructor); // back to Person. output > Person{}
```

Prototypes of arrays:

```
const arr = [3, 6, 6, 5, 6, 9, 9]; // new Array === []
console.log(arr.__proto__);
```

output:

```

▼ Array []
  ▶ at: function at()
  ▶ concat: function concat()
  ▶ constructor: function Array()
  ▶ copyWithin: function copyWithin()
  ▶ entries: function entries()
  ▶ every: function every()
  ▶ fill: function fill()
  ▶ filter: function filter()
  ▶ find: function find()
  ▶ findIndex: function findIndex()
  ▶ flat: function flat()
  ▶ flatMap: function flatMap()
  ▶ forEach: function forEach()
  ▶ includes: function includes()
  ▶ indexOf: function indexOf()
  ▶ join: function join()
  ▶ keys: function keys()
  ▶ lastIndexOf: function lastIndexOf()
  ▶ length: 0
  ▶ map: function map()
  ▶ pop: function pop()
  ▶ push: function push()
  ▶ reduce: function reduce()
  ▶ reduceRight: function reduceRight()
  ▶ reverse: function reverse()
  ▶ shift: function shift()
  ▶ slice: function slice()
  ▶ some: function some()
  ▶ sort: function sort()
  ▶ splice: function splice()
  ▶ toLocaleString: function toLocaleString()
  ▶ toString: function toString()
  ▶ unshift: function unshift()
  ▶ values: function values()
  ▶ Symbol(Symbol.iterator): function values()
  ▶ Symbol(Symbol.unscopables): Object { at: true, copyWithin: true, entries: true, _ }
  ▶ <prototype>: Object { _ }

```

```
▶ GET http://127.0.0.1:5500/favicon.ico
```

```
>>
```

```

const arr = [3, 6, 6, 5, 6, 9, 9]; // new Array === [] (using [] is same as using new array so
therefore whenever we create an array like this, it is indeed created by the array
constructor. And so that's why arr.__proto__ works behind the scenes.)
console.log(arr.__proto__);
// console.log(arr.__proto__ === Array.prototype); //true (the prototype property of the
constructor is gonna be the prototype of all the objects created by that constructor.)

```

So we know at this point, that any array inherits all their methods from it's a prototype. And therefore we can use that knowledge to extend the functionality of arrays even further. So all we would have to do is to say `array.prototype`.

And then here we can add any new method to this prototype and all the arrays will then inherit it.

```
Array.prototype.unique = function () {  
  // we added a new method to the prototype property of the array constructor.  
  return [...new Set(this)];  
};  
console.log(arr.unique()); //now all arrays will inherit this method
```

\*\*\* extending the prototype of a built-in object is generally not a good idea. I mean if you're working just on a small project on your own then I guess you could do this, but really don't get into the habit of doing this for multiple reasons. The first reason is that the next version of JavaScript might add a method with the same name that we are adding, for example this one here, but it might work in a different way. And so your code will then use that new method which, remember, works differently. And then that will probably break your code. And the second reason why you shouldn't do this is because when you work on a team of developers, then this is really gonna be a bad idea because if multiple developers implement the same method with a different name then that's just going to create so many bugs that it's just not worth doing this. So it's just a nice and fun experiment but in practice, you should probably not do it.

and we already know that all the DOM elements are behind the scenes objects. And so let's take a look at this object. Well, that doesn't really work. So that just gives us the object. So let's try a console.dir on this H1. And so now we get the actual object. So here we have all this different stuff and all these properties and methods that we already worked with. But what I'm interested in here is the prototype and you see that the prototype is an HTML heading element. All right? And so that itself will contain a lot of different stuff, one more time. So let's scroll all the way down. And so this is now an HTML element. And do you remember these names here from somewhere? Well, these are exactly the types of objects that we talked about in one of the first lectures in the DOM section. Remember that? So we had this big diagram with all these different types of elements. And so now here we can actually inspect them. And so behind the scenes these different elements are really different like constructor functions. So if we go down here we should now see element

and indeed, here it is. And so again, you can recall that diagram that we inspected there.

And then you will remember that HTML element was a child element of element and element itself was a child of node, right? And so therefore the prototype of element

is gonna be node. So if we scroll all the way down here then you see indeed node, and now this one is event target.

And so you'll see, this is a huge, huge prototype chain.

And only now we are at the end. And so the end is an object and that's it.

So this prototype chain has easily six or seven levels.

And so you see that it can go really, really deep. So if you want to go and check this out by yourself then that would probably be a really nice exercise.

And finally, let's just also console.dir some random function.

So the function doesn't matter. I just want to be able to look at the function.

Okay. And so, as I mentioned a bit earlier in this video

the function itself is also an object. And so therefore it also has a prototype.

And in this case to prototype will then contain the methods

that we have used previously on functions. So that's apply, bind and call, remember.

And so once again this is the reason why we can actually call methods on functions. It's because they are objects and objects have prototypes.

And in this case, this function prototype.

## ❖ 14. Object-Oriented Programming (OOP) With JavaScript → 10. ES6 Classes

we learned how to implement prototypical inheritance with constructor functions and then manually setting methods on the constructor functions, prototype property. But now it's time to turn our attention to ES6 classes. Which essentially allow us to do the exact same thing,

but using a nicer and more modern syntax. Now, as I mentioned earlier, classes in JavaScript do not work like traditional classes in other languages like Java or C Plus Plus.

So instead classes in JavaScript are just synthetic sugar over what we learned in the last few videos. So they still implement prototypical inheritance behind the scenes, but where they syntax that makes more sense to people coming from other programming languages. And so that was basically the goal of adding classes to JavaScript. But anyway

```
// Class expression (don't use this)
```

```

const PersonCl1 = class {}

// Class declaration
class PersonCl {
  constructor(fullName, birthYear) {
    this.fullName = fullName;
    this.birthYear = birthYear;
  }

  // Instance methods
  // Methods will be added to .prototype property
  calcAge() {
    console.log(2037 - this.birthYear);
  }
}

```

let's now implement person using a class. So we can write class and then the name of the class,

and let's actually call it PersonCL. So that stands for class and then curly braces like this.

And so this is actually a class declaration,

but just like in functions, we also have class expressions. And so that would work like this.

So class expression, and class declaration, And so then you can pick whichever you like the most.

So it would be PersonCL and then class, and then just like a function, but without the arguments, okay.

And that is because in fact classes are just a special type of functions, okay.

So although we use the class keyword here behind the scenes classes are still functions and therefore we have class expressions

and class declarations. Now, for some reason I prefer the class declaration,

and so I'm gonna use that right here. And so this is how we write a simple class

and then inside the class, the first thing that we need to do

is to add a constructor method. So just like this, and this constructor actually works in a pretty similar way as a constructor function, so as we studied previously,

but this one is actually a method of this class. And in fact it needs to be called constructor, so that is the rule. But just like in constructor functions, we pass in arguments basically for the properties

that we wanted the object to have. So that's again first name and then the birth year.

Now, the act of creating a new object

actually also works in the exact same way as before. So using the new operator, and so therefore, whenever we create a new object,



so like a new instance using the new operator, this constructor will automatically be called. So, let's actually try that right away. And this time let's create Jessica here. So new PersonCL, and so as you see everything here looks exactly the same as before. So it looks just like a regular function call. And again, we also use the new keyword here. And so therefore just like before, the disc keyword here inside of the constructor will also be set to the newly created empty object. And so just like before, we set the properties of the object like this. So this dot first name is equal to first name that we receive and the same for the birth year. Alright. So basically when we create a new instance here, then it is this constructor that is gonna be called and it will return a new object and then that will be stored here into Jessica. So let's just look this to the console. And so, in fact it looks just like before, alright. So here we basically have the properties that will be stored in the new object that we want to create, and so now it's time for the methods. And the methods we simply write like this, so we can simply add them here. And all we have to do is to write their name. So just like a regular function in here. So this is very nice and very convenient. And so let's simply do the same as before. So this dot birth year. Now, what's important to understand here is that all of these methods that we write in the class, so outside of the constructor, will be on the prototype of the objects and not on the objects themselves. So this is really just like before prototypal inheritance. And in fact, we will be able to confirm that now. So as we open up PersonCL here, and in fact, we're gonna be able to confirm that here. So as we inspect this Jessica object, when we look into it's prototype, then we will once again see the calcAge function here.

And so one more time, let me prove to you that Jessica underscore proto underscore underscore is equal to PersonCL dot prototype, and is true. And so as you see PersonCL here acts just like any other function constructor. Where the only difference that this looks a little bit nicer. So with this syntax, we don't have to manually mess with the prototype property. All you have to do is to write the methods here. So inside the class, but outside of the constructor, and then they will automatically be added to the prototype property of the class, basically. So let me actually write that here. So they will be added to the prototype property of the person class, which once again, is gonna be the prototype of the objects created by that class.

```
console.log(jessica.__proto__ === PersonCl.prototype);
```

And we can take this demonstration even further by also adding a method manually to the prototype. So that's going to work just as fine. So let's create a greet method here. So let's just say, hey, this dot first name, and then we can call this on Jessica.

```
PersonCl.prototype.greet = function () {  
  console.log(`Hey ${this.firstName}`);  
};  
jessica.greet();
```

\*\*\*

// 1. Classes are NOT hoisted (and so even if they are class declarations. So function declarations, remember are hoisted, which means we can use them before they are declared in the code. But with classes, that doesn't work.)

// 2. Classes are first-class citizens ( that means, is that we can pass them into functions and also return them from functions. And as I mentioned before, that is because classes are really just a special kind of function behind the scenes.)

// 3. Classes are executed in strict mode ( the body of a class is always executed in strict mode. so even if we didn't activate it for our entire script, all the code that is in the class will be executed in strict mode. Am I right? So keep these points in mind)

you might ask if you should use constructor functions like we have been doing, or if instead it's better to just use classes. And to first remark I want to do here is that constructor functions are not like old or deprecated syntax.

So it's 100% fine to keep using them. So one more time, this is more,

a question of personal preference. Now, if you're asking, if you should use classes without understanding prototypal inheritance, well then, the reply is definitely no.

Now, some people actually say that classes are really bad in general and that no one should ever be using them simply because they hide the true nature of JavaScript.

## ❖ 14. Object-Oriented Programming (OOP) With JavaScript → 11. Setters and Getters

Let's now talk about a feature that is actually common to all objects in JavaScript, and that's getters and setters. So every object in JavaScript can have setter and getter properties. And we call these special properties accessor properties, while the more normal properties are called data properties. So getters and setters are basically functions that get and set a value so just as the name says, but on the outside they still look like regular properties. And so let's first take a look at getters and setters

\*\*\*Now it is not **mandatory** to specify a setter when we have a getter for the same property.

Okay, so just a getter or just a setter would be enough.

```
// Setters and Getters
const account = {
  owner: 'Jonas',
  movements: [200, 530, 120, 300],

  get latest() {
    return this.movements.slice(-1).pop();
  },

  set latest(mov) {
    this.movements.push(mov);
  },
};

console.log(account.latest); // 300

account.latest = 50;
console.log(account.movements); // >(5) [200, 530, 120, 300, 50]
```

so in a nutshell this is how getters and setters work for any regular object in JavaScript. Now however, classes do also have getters and setters, and they do indeed work in the exact same way.

And so let's try them out now here in our person class.

```
// Class declaration
```

```

class PersonCl {
  constructor(fullName, birthYear) {
    this.fullName = fullName;
    this.birthYear = birthYear;
  }

  // Instance methods
  // Methods will be added to .prototype property
  calcAge() {
    console.log(2037 - this.birthYear);
  }

  greet() {
    console.log(`Hey ${this.fullName}`);
  }

  get age() {
    return 2037 - this.birthYear;
  }
}

const jessica = new PersonCl('Jessica Davis', 1996);
console.log(jessica.age); //usage of get set

```

setters and getters can actually be very useful for data validation and as an example, let's try some validation with the name.

And so in this case then we actually want to set this.fullName to the name that was received.

But if not, we want to give an alert. So the given name is not a full name.

So in this case what's really important to understand is that we are creating a setter for a property name

that does already exist. So fullName is already a property that are trying to set here, but then we also have the setter. And so now what's gonna happen is that each time this code here is executed, so whenever we set the fullName on the this keyword, then actually this method here,

so this setter is gonna be executed. And so that name that we pass in as fullName will then become this name. All right, let's check that out actually. And so now as we create Jessica here,

you will see and indeed it, we saw Jessica Davis here, but now we got this crazy error here of maximum call stack size exceeded. Now that's a very cryptic error message, but what

happens here is that there is a conflict. So right now both the setter function and this constructor function are trying to set the exact same property name. And so that gives origin to this weird error. So what we need to do instead is to here create a new property name. And the convention for doing that, so when we have a setter which is trying to set a property that does already exist, then here as a convention we add an underscore.

So again, this is just a convention, it's not a JavaScript feature.

So it's really just a different variable name to avoid that naming conflict.

However, now when we do this, we are actually creating a new variable, so a `fullName` variable.

So let's try this now actually. So if we try to look at Jessica Davis you see that right now the property that exists is `underscore fullName`. And so right now we cannot do

`jessica.fullName`

because that simply doesn't exist. And so to fix this we now also need to create a getter for the `fullName` property. And so that will simply return the `underscore fullName`.

So let's see. So return this `_fullName`.

```
// ES6 Classes

// Class expression
// const PersonCl = class {}

// Class declaration
class PersonCl {
  constructor(fullName, birthYear) {
    this.fullName = fullName;
    this.birthYear = birthYear;
  }

  // Instance methods
  // Methods will be added to .prototype property
  calcAge() {
    console.log(2037 - this.birthYear);
  }

  greet() {
    console.log(`Hey ${this.fullName}`);
  }

  get age() {
    return 2037 - this.birthYear;
  }

  // Set a property that already exists
```

```

set fullName(name) {
  if (name.includes(' ')) this._fullName = name;
  else alert(`${name} is not a full name!`);
}

get fullName() {
  return this._fullName;
}
}

const jessica = new PersonCl('Jessica Davis', 1996);
console.log(jessica.age);

```

\*\*\* Now we don't need to use getters or setters, and many people actually don't, but yeah, as I just said sometimes it's just nice to be able to use these features and especially when we need like a validation like this

## ❖ 14. Object-Oriented Programming (OOP) With JavaScript → 12. Static Methods

Now a good example to understand what a static method actually is, is the build in Array.from method.

```

Array.from(document.querySelectorAll('h1'))
[1,2,3].from() // Error -

```

that is because this from method here is really attached to the entire constructor (Array) and not to the prototype property of the constructor. And so therefore all the Arrays do not inherit this method. We also say that the from method is in the Array name space.

And so maybe you can imagine that it should be pretty easy to implement static methods or selfs. And let's do that for both or constructor function and also for the class.

constructor function:

```

Person.hey = function () {
  console.log('Hey there 🙌');
  console.log(this); //output: Person{} (object that is calling the method)
};

Person.hey(); // Hey there 🙌
jonas.hey(); // error - jonas cant access - not inherited

```

Class:

```
// Class
class PersonCI {
  constructor(fullName, birthYear) {
    this.fullName = fullName;
    this.birthYear = birthYear;
  }

  // Instance methods
  // Methods will be added to .prototype property
  calcAge() {
    console.log(2037 - this.birthYear);
  }

  greet() {
    console.log(`Hey ${this.fullName}`);
  }

  get age() {
    return 2037 - this.birthYear;
  }

  // Set a property that already exists
  set fullName(name) {
    if (name.includes(' ')) this._fullName = name;
    else alert(`${name} is not a full name!`);
  }

  get fullName() {
    return this._fullName;
  }

  // Static method
  static hey() {
    console.log('Hey there 🙌');
    console.log(this);
  }
}
```

## ❖ 14. Object-Oriented Programming (OOP) With JavaScript → 13. Object.create

we learned about constructor functions and ES6 classes. But there is actually a third way of implementing prototypal inheritance or delegation, as we can also call it. And that third way is to use a function called `Object.create`, which works in a pretty different way than constructor functions and classes work. Now, with `Object.create`, there is still the idea of prototypal inheritance. However, there are no prototype properties involved. And also no constructor functions, and no `new` operator. So instead, we can use `Object.create` to essentially manually set the prototype of an object, to any other object that we want. Okay, so if we can set the prototype to any object, let's actually create an object that we want to be the prototype of all the person objects.

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

const steven = Object.create(PersonProto);
console.log(steven); // {} (empty object that has link to PersonProto)

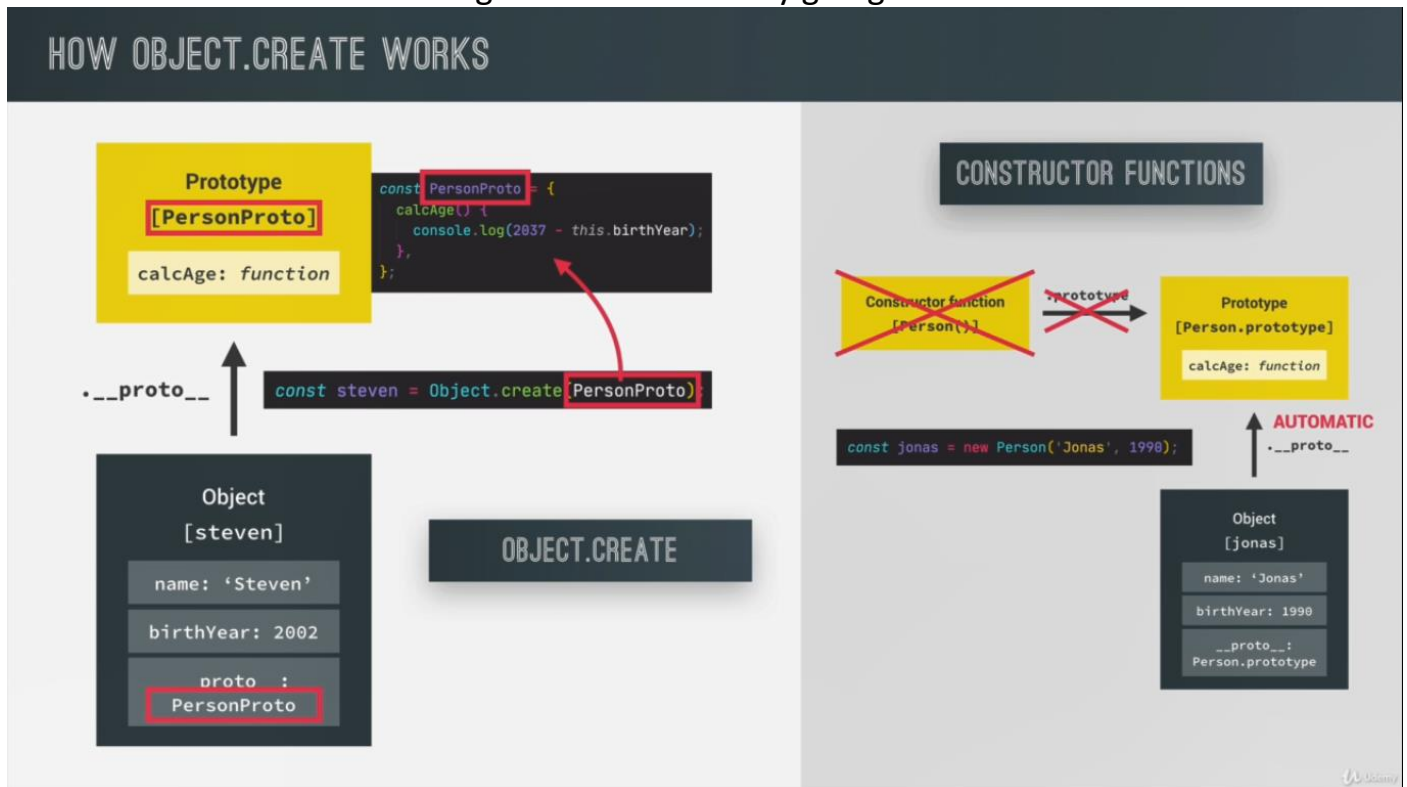
console.log(steven.__proto__ === PersonProto); // true
```

But now we don't have any properties on the object yet. So let's quickly fix that. So I'm doing it here, just like I would do in any other object literal. And so this is not ideal, of course. But we're gonna fix that in a minute. Because of course, we want actually a programmatic way of creating new objects, instead of having to do it like this.

```
steven.name = 'Steven';
steven.birthYear = 2002;
steven.calcAge();
```



And so let's take a look at a diagram of what's really going on here.



So here at the right side, we have the way it works where de constructor functions, just as we have been doing it up until this point. So when we use the new operator in constructor functions or classes, it automatically sets the prototype of the instances to the constructors, prototype property. So this happens automatically. And so that's nothing new at this point for you. Now, on the other hand, with Object.create, we can set the prototype of objects manually to any object that we want. And in this case, we manually set the prototype of the Steven object to the person proto object. And that's it. Now the two objects are effectively linked through the proto property, just like before. So now looking at properties, or methods in a prototype chain, works just like it worked in function constructors, or classes. And so the prototype chain, in fact, looks exactly the same here. The big difference is that we didn't need any constructor function, and also no prototype property at all, to achieve the exact same thing. So this is actually a bit more straightforward, and a bit more natural. And I guess, that it might also be easier to understand. However, the reason why I'm showing you this Object.create technique, right at the end, is because in practice,

in the real world, this is actually the least used way of implementing prototypical inheritance.

However, it's still very important to know exactly how `Object.create` works, because you will still stumble upon this in the real world. And even more importantly,

we will need `Object.create` to link prototypes in the next lecture, in order to implement inheritance between classes. So with that, we're gonna take object oriented programming to a whole new level. And the `Object.create` function is gonna be crucial in that, as we will see.

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

  init(firstName, birthYear) {
    this.firstName = firstName;
    this.birthYear = birthYear;
  },
};

// doing this is a little bit weird ()
// const steven = Object.create(PersonProto);
// console.log(steven); // {} (empty object that has link to PersonProto)
// steven.name = 'Steven';
// steven.birthYear = 2002;
// steven.calcAge();

// So if we're serious about using Object.create, we should implement a function that
// basically does this work for us.
const sarah = Object.create(PersonProto);
sarah.init('Sarah', 1979);
sarah.calcAge();
```

## ❖ 14. Object-Oriented Programming (OOP) With JavaScript → 15. Inheritance Between 'Classes': Constructor Functions

So over the last couple of lectures, we explored how prototypal inheritance works in JavaScript.

And we did that using a couple of different techniques. So we used constructor functions, ES6 classes, and object dot create.

Now all of these techniques basically allow objects to inherit methods from its prototype.

So to delegate their behavior to their prototype, right?

But now it's time to turn our attention to more real inheritance. So in the way that we learned in the very first lecture of the section. So what I'm talking about is real inheritance

between classes and not just prototypal inheritance between instances and a prototype property like we've been doing so far. And I'm using class terminology here because this simply makes it easier to understand what we're gonna do. But of course, you already know that real classes do not exist in JavaScript. But anyway here is what we're gonna do. So we will create a new student class and make it inherit from the person class that we have been using so far. So person will be the parent class and student will be the child class. that's because student is basically a subtype of a person.

So a student is also a person, right? But it is a more specific person.

And so this is an ideal child class. Now this is really useful because with this inheritance set up,

we can have specific methods for the student, but then the student can also use generic person methods, like the calcAge method that we have been using.

And that's basically the idea of inheritance that we're gonna implement in this lecture.

Now, just like before we will start by implementing this using constructor functions.

So in this lecture, we will inherit between classes using constructor functions, and this is gonna be a bit of work, but it will allow you to understand exactly how we set up the prototype chain in order to allow inheritance between the prototype properties

of two different constructor functions. Then in the next lecture,

we're gonna do the same thing using ES6 classes, which as you would expect is a lot easier.

And finally, of course, we will go back to using object dot create as well. All right, but enough talk, let's put this into practice now.

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

```

};

Person.prototype.calcAge = function () {
  console.log(2037 - this.birthYear);
};

const Student = function (firstName, birthYear, course) {
  this.firstName = firstName; // same as Person
  this.birthYear = birthYear; // same as Person
  this.course = course;
};

```

having duplicate code is never a good idea. First because it violates the "don't repeat yourself" principle, but second and even worse in this case is that imagine that the implementation of person here changes in the future, then that change will not be reflected in the student. So instead of having this duplicate code here, let's simply call the person function.

```

const Student = function (firstName, birthYear, course) {
  Person(firstName, birthYear);
  this.course = course;
};

```

do you think that this is gonna work? Well let's see.

And then I'm gonna to explain to you why this doesn't work. So the problem here is that we are now actually calling this person constructor function as a regular function call. So we are not using this new operator to call this person function constructor. And so therefore this function call here is simply a regular function call. And remember that in a regular function call, the this keyword is set to undefined. And so therefore that's why we get this error here, that it cannot set first name on undefined. So instead of simply calling the person function here, we need to manually set the this keyword as well.

So do you remember how we can call a function? And at the same time set the this keywords inside that function? Well, we can simply use the call method. So the call method will indeed call this function, but we will be able to specify the this keywords here as the first argument in this function. And so in this case, we want the this Keyword in **Student** function to simply be the this keyword inside **Person** function here

```

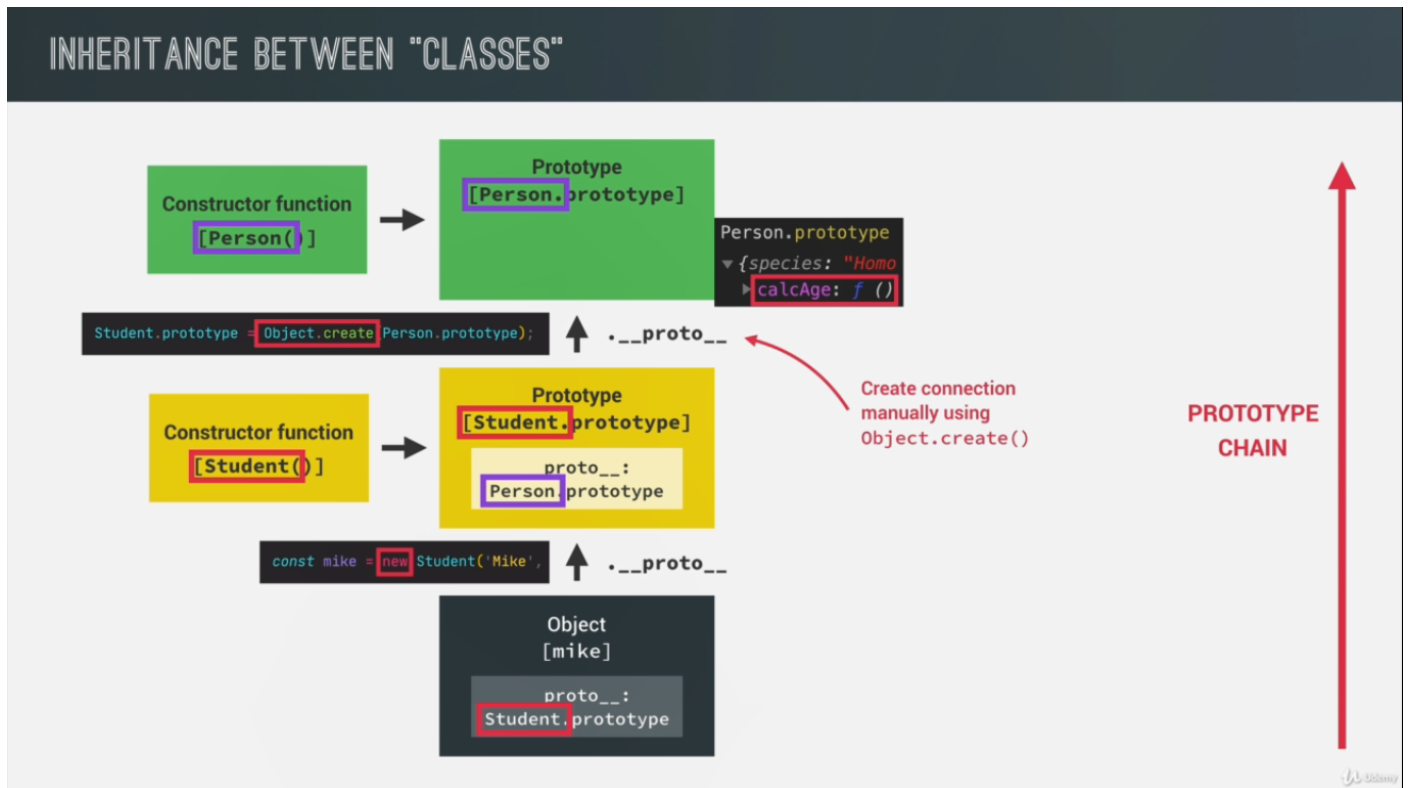
const Student = function (firstName, birthYear, course) {

```

```

Person.call(this, firstName, birthYear);
this.course = course;
};

```



So far, this is what we have built. So it's simply the student constructor function and its prototype property. And then the mike object linked to its prototype. And that prototype is of course the constructor functions prototype property. Now this link between instance and prototype has been made automatically by creating the mike object with the new operator. So all of this is what we had already learned. So this is nothing new at this point Right? Now a student is also a person. And so we want student and person to be connected like this.

So we really want the student class to be the child class and inherit from the person class, which will then function as the parent class. So this way, all instances of student could also get access to methods from the person's prototype property, like the calcAge method through the prototype chain. And that's the whole idea of inheritance. Its the child classes can share behavior

from their parent classes. So looking at this diagram, basically what we want to do is to make person dot prototype,

the prototype of student dot prototype. Or in other words, we want to set the underscore protal property

of student dot prototype to person dot prototype. And if this sounds confusing,

So that's why I created it. Okay? Now, anyway, we are gonna have to create this connection manually.

And to do this, so to link these two prototype objects, we are gonna use object dot create because defining prototypes manually is exactly what object.create() does. Great. So let's go do that.

### Bad - doesn't work: (right diagram in the image)

```
// Inheritance Between "Classes": Constructor Functions
const Person = function (firstName, birthYear) {
  this.firstName = firstName;
  this.birthYear = birthYear;
};

Person.prototype.calcAge = function () {
  console.log(2037 - this.birthYear);
};

const Student = function (firstName, birthYear, course) {
  Person.call(this, firstName, birthYear);
  this.course = course;
};

// Linking prototypes
⇒ Student.prototype = Person.prototype // doesn't work (see the diagram)

Student.prototype.introduce = function () {
  console.log(`My name is ${this.firstName} and I study ${this.course}`);
};

const mike = new Student('Mike', 2020, 'Computer Science');
mike.introduce();
```

### good – does work: (left diagram in the image)

```
// Inheritance Between "Classes": Constructor Functions
```

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

```
Person.prototype.calcAge = function () {
  console.log(2037 - this.birthYear);
};
```

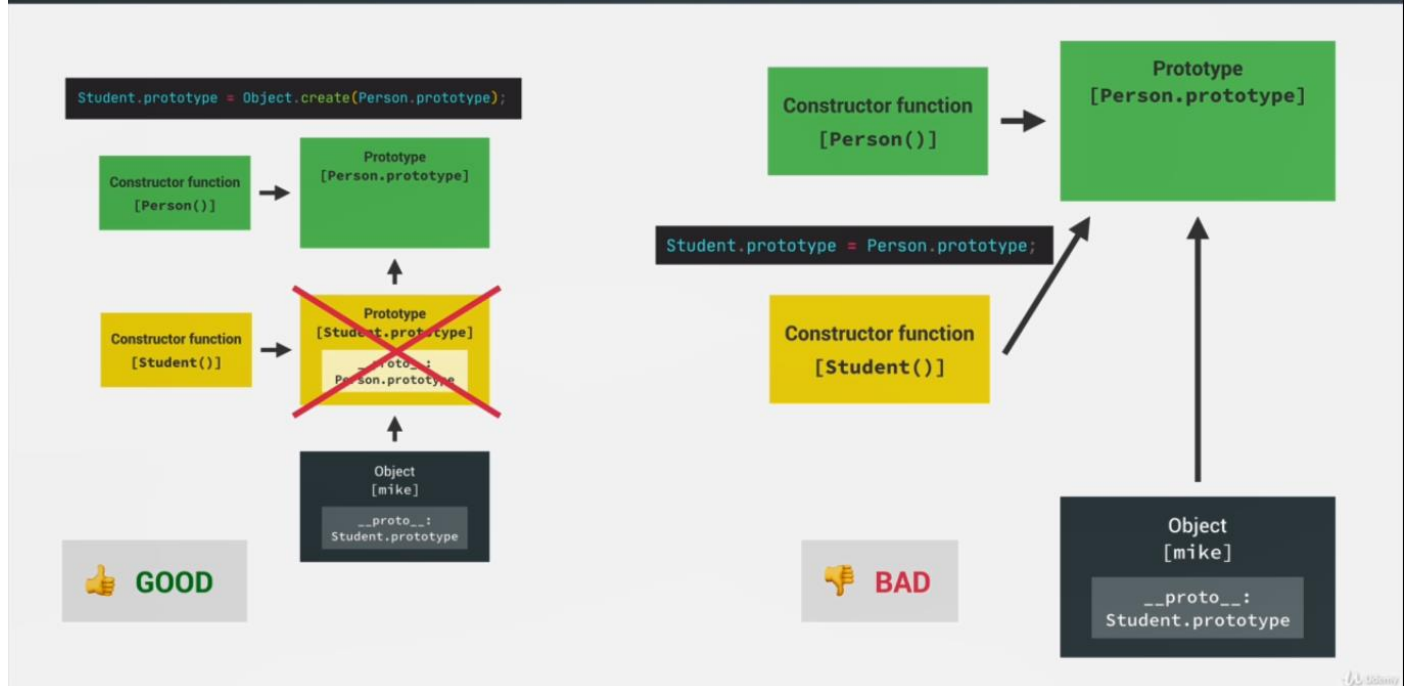
```
const Student = function (firstName, birthYear, course) {
  Person.call(this, firstName, birthYear);
  this.course = course;
};
```

// Linking prototypes

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

```
Student.prototype.introduce = function () {
  console.log(`My name is ${this.firstName} and I study ${this.course}`);
};
const mike = new Student('Mike', 2020, 'Computer Science');
mike.introduce();
```

## INHERITANCE BETWEEN "CLASSES"



So now we can say :

```
mike.calcAge();
```

So when we do mike dot calcAge, we are effectively doing a property or a method lookup. So that is JavaScript

trying to find the requested property or method. Now, in this case, as we know, the calcAge method is of course

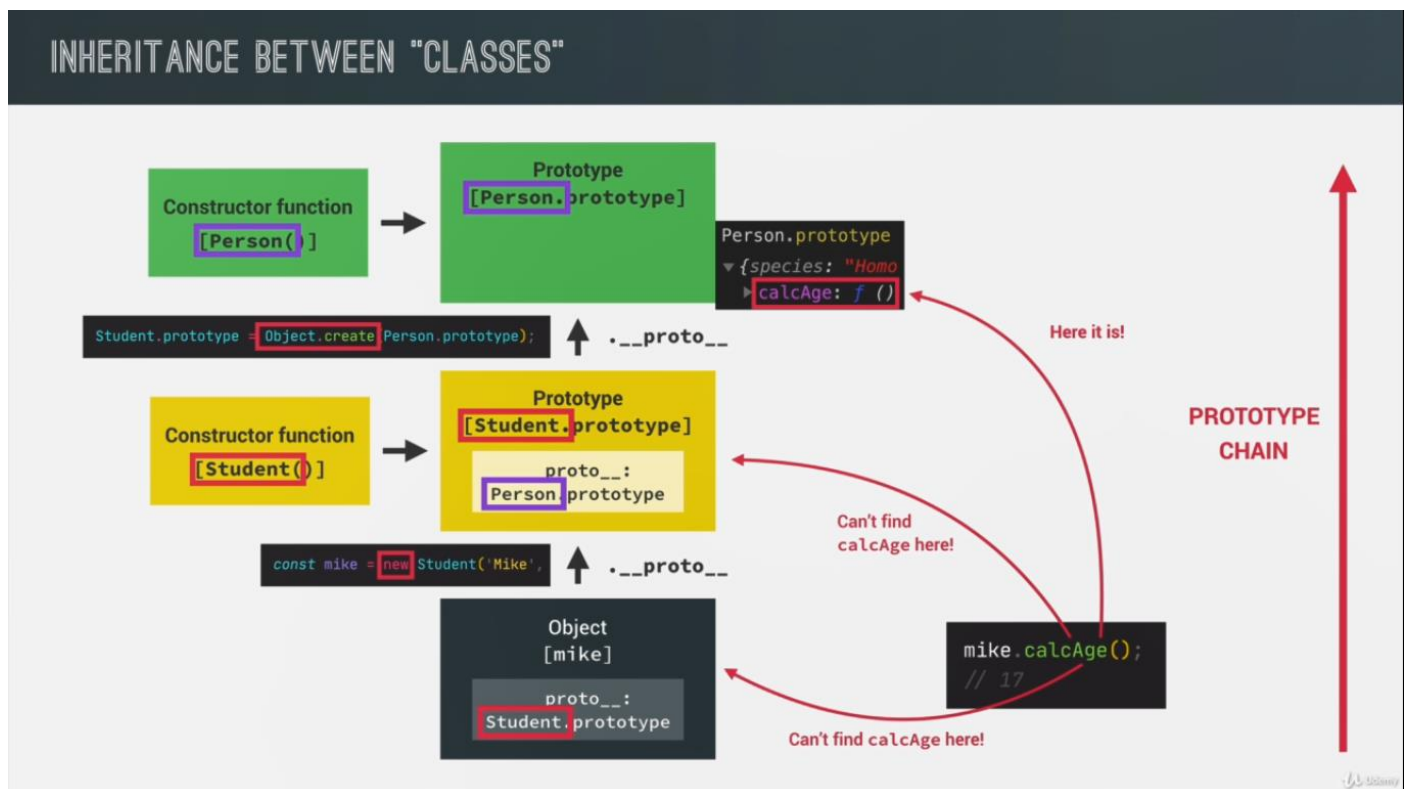
not directly on the mike object. It's also not in mike's prototype. That's where we defined the introduced method,

but not calcAge. Right? so just like before, whenever we try to access a method, that's not on the object's prototype,

then JavaScript, will look up even further in the prototype chain and see if it can find a method

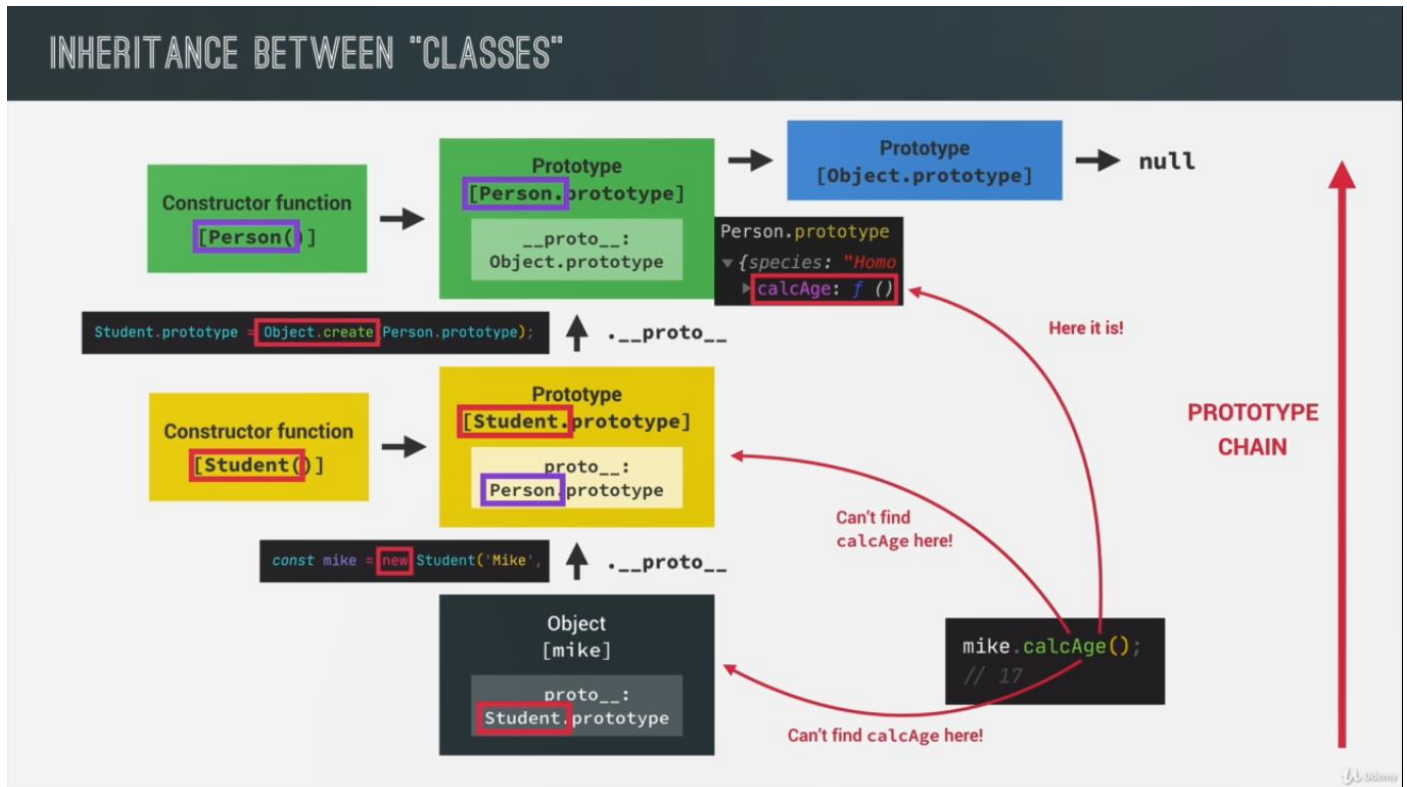
so in the parent prototype. And that's exactly what happens here. So JavaScript will finally find the calcAge

in person dot prototype, which is exactly where we defined it. And that's the whole reason why we set up the prototype chain like this





let's also quickly complete the prototype chain. So just like before object.prototype will sit on top of the prototype chain. (blue box)



So when we take a look at student dot prototype dot constructor, then remember that ideally this should point back to the student, right? But here it points back, apparently to person. So actually we should use console dot dir

And so you see that JavaScript now, thinks that the constructor of student or prototype is person here.

And the reason for that is that we set the prototype property of the student using object dot create. And so this makes it so that the constructor of student dot prototype is still person.

So we need to fix this because sometimes it's important to rely on this constructor property. And so if we want to rely on that, it should indeed be correct. But that's easy to fix. We can just say student dot prototype dot constructor and set it to student.

```
Student.prototype.constructor = Student;
```

More info :

```
console.log(mike instanceof Student); // true
```

```
console.log(mike instanceof Person); // true
console.log(mike instanceof Object); // true
```

## ❖ 14. Object-Oriented Programming (OOP) With JavaScript → 17. Inheritance Between Classes: ES6 Classes

go ahead and implement the exact same thing that we did in the last video, but this time using ES6 classes instead of constructor functions.

to implement inheritance between ES6 classes, we only need two ingredients.

- We need the **extends** keywords
- we need the **super** function.

**extends** keyword will then link to prototypes behind the scenes without us even having to think about that.

Then of course, we still need a constructor. And this one will just like before receive the same arguments as the parent class, but then some additional ones.

```
class PersonCl {
  constructor(fullName, birthYear) {
    this.fullName = fullName;
    this.birthYear = birthYear;
  }

  // Instance methods
  calcAge() {
    console.log(2037 - this.birthYear);
  }

  greet() {
    console.log(`Hey ${this.fullName}`);
  }

  get age() {
    return 2037 - this.birthYear;
  }

  set fullName(name) {
    if (name.includes(' ')) this._fullName = name;

```

```

else alert(`${name} is not a full name!`);
}

get fullName() {
  return this._fullName;
}

// Static method
static hey() {
  console.log('Hey there 🙌');
}
}

```

```

⇒ class StudentCI extends PersonCI {
  constructor(fullName, birthYear, course) {
    // PersonCI.call() // => we dont need to do that
    // we do instead calling super
    super(fullName, birthYear);
    this.course = course;
  }
}

```

we actually don't even need to manually call like **personci.call()** like we did before in the constructor function. Remember? so here we don't need to do that. What we do instead is to call the super function. And so super is basically the constructor function of the parent class. So the idea is still similar to what we did in the constructor function, but here it all happens automatically.

now all we do is to pass in the arguments in super (for the constructor of the parent class). Same as parameters that are also specified in constructor of parent class

using super in constructor of the child class, this always needs to happen first. that's because this call to the super function is responsible for creating the this keyword in this subclass. And so therefore, without doing this, we wouldn't be able to access the this keyword to do this: **this.course = course;**

So always first the call to the super so to the parents class constructor. we will then be able to access the this keyword.

```

class StudentCI extends PersonCI {
  constructor(fullName, birthYear, course) {
    // PersonCI.call() // => we dont need to do that
    // we do instead calling super
    super(fullName, birthYear);
  }
}

```

```

    this.course = course;
  }
}

```

If we don't need to use this class, then don't need to set constructor:

```

class StudentCI extends PersonCI {
  // if we dont need to use this keyword,we dont need to constructor function
}
const martha = new StudentCI('Martha Jones', 2012);
martha.calcAge(); //Marta has access to parent class items and (even we dont have
constructor in student class) we have correct output -> 25

```

all students has inheritance(for example Marta):

1. StudentCI {\_fullName: 'Martha Jones', birthYear: 2012}
  1. birthYear: 2012
  2. \_fullName: "Martha Jones"
  3. age: 25
  4. fullName: "Martha Jones"
  5. [[Prototype]]: PersonCI
    1. constructor: class StudentCI
    2. introduce: f introduce()
    3. age: 25
    4. fullName: "Martha Jones"
    5. [[Prototype]]: Object

Overriding a parent method in child class :

```

class StudentCI extends PersonCI {
  constructor(fullName, birthYear, course) {
    // Always needs to happen first!
    super(fullName, birthYear);
    this.course = course;
  }

  introduce() {
    console.log(`My name is ${this.fullName} and I study ${this.course}`);
  }

  ⇒ calcAge() {
    console.log(
      `I'm ${

```

```

    2037 - this.birthYear
  } years old, but as a student I feel more like ${
    2037 - this.birthYear + 10
  }`
);
}
}

const martha = new StudentCI('Martha Jones', 2012, 'Computer Science');
martha.calcAge(); //output-> I'm 25 years old, but as a student I feel more like 35

```

this new method overrode the one that was already there in the prototype chain (method coming from the parent class). And again, that's because this new calcAge() method here appears first in the prototype chain. And so therefore it is essentially overriding the method coming from the parent class. And we can also say that this calcAge() method here is shadowing the one that is in the parent class.

#### ❖ 14. Object-Oriented Programming (OOP) With JavaScript → 18. Inheritance Between Classes: Object.create

And now finally lets look at how we can use Object.create in order to implement a complex prototype chain. So similar to what we implemented before with classes and constructor functions.

So here, PersonProto. So this object up here used to be the prototype of all the new person objects. But now we basically want to add another prototype in the middle of the chain. So between PersonProto and the object. And so what we're going to do is to make student inherit directly from person. So we're going to create now an object

```

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

  init(firstName, birthYear) {
    this.firstName = firstName;
    this.birthYear = birthYear;
  }
};

```

```
},  
};
```

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

⇒ `const StudentProto = Object.create(PersonProto);` // And so now we can use this `StudentProto` to create new students.

```
const jay = Object.create(StudentProto);
```

And so now the `StudentProto` object that we just created earlier, is now the prototype of the `jay` object.

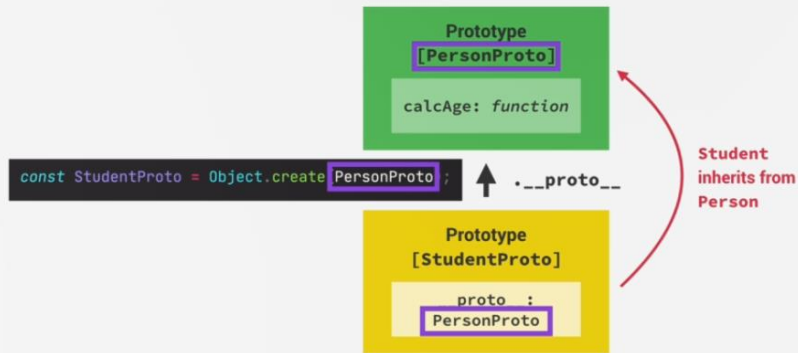
So again, the `StudentProto` object is now the prototype of `jay`, and the `PersonProto` object is in turn

the prototype of `StudentProto`. And so therefore, `PersonProto` is a parent prototype of `jay`, which means that it's in its prototype chain.

So it all starts with the `PersonProto` object, which used to be the prototype of all person objects,

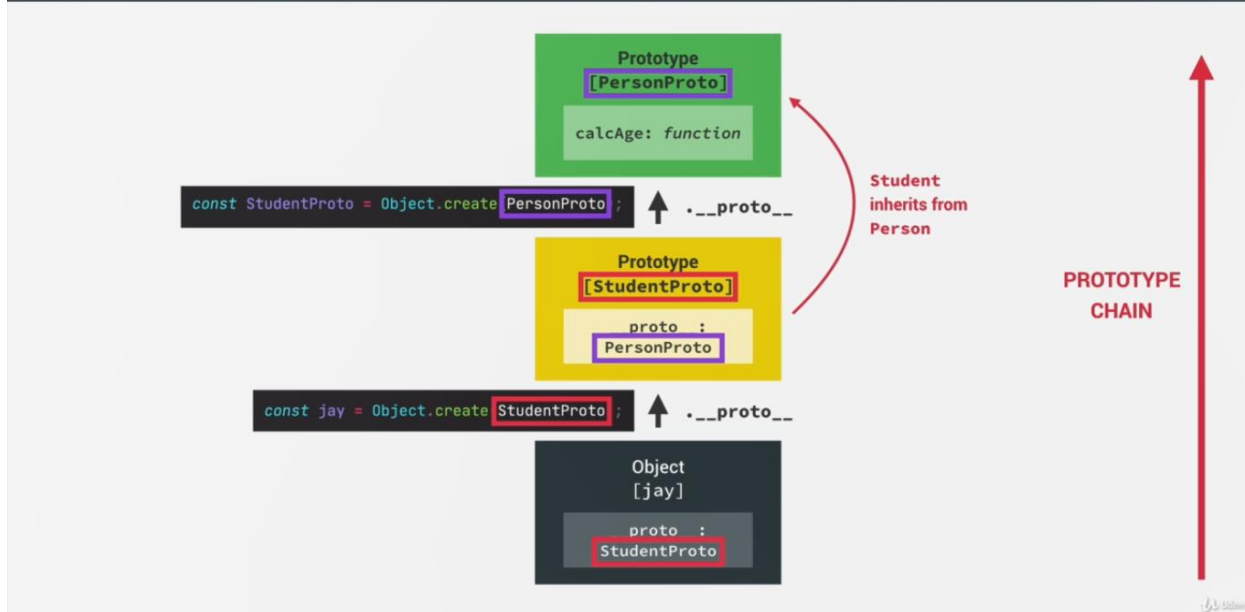
but now using `Object.create`, we make it so that `PersonProto` will actually become the prototype of `StudentProto`. And what this does is that now basically student inherits from person. And so it is, we already established the parent child relationship that we were looking for.

## INHERITANCE BETWEEN "CLASSES": OBJECT.CREATE



Now to finish, all we need to do is to use `Object.create` again, but this time to create a new actual student object. And of course we make the student. So `jay` in this course inherits from `StudentProto`, which is now `jay`'s prototype. And like this, we created a nice and simple prototype chain. So `jay` inherits from `StudentProto`, which in turn inherits from `PersonProto`, and therefore the `jay` object will be able to use all the methods that are contained in `StudentProto` and `PersonProto`.

## INHERITANCE BETWEEN "CLASSES": OBJECT.CREATE



// Inheritance Between "Classes": Object.create

```
const PersonProto = {  
  calcAge() {  
    console.log(2037 - this.birthYear);  
  },  
  
  init(firstName, birthYear) {  
    this.firstName = firstName;  
    this.birthYear = birthYear;  
  },  
};  
  
const steven = Object.create(PersonProto);
```



```

const StudentProto = Object.create(PersonProto);
StudentProto.init = function (firstName, birthYear, course) {
  PersonProto.init.call(this, firstName, birthYear);
  this.course = course;
};

StudentProto.introduce = function () {

  console.log(`My name is ${this.firstName} and I study ${this.course}`);
};

const jay = Object.create(StudentProto);
jay.init('Jay', 2010, 'Computer Science');
jay.introduce();
jay.calcAge();

```

And that's actually it. So in this version, we don't even worry about constructors anymore, and also not about prototype properties, and not about the new operator. So it's really just objects linked to other objects. And it's all really simple and beautiful, if you ask me. And in fact, some people think that this pattern is a lot better than basically trying to fake classes in JavaScript, because faking classes in the way that they exist in other languages like Java or C plus plus is exactly what we do by using constructor functions, and even ES6 classes. But here, in this technique that I just showed you with Object.create, we are, in fact, not faking classes.

All we are doing is simply linking objects together, where some objects then serve as the prototype of other objects. And personally, I wouldn't mind if this was the only way of doing OOP in JavaScript, but as I mentioned earlier, ES6 classes and constructor functions are actually way more used in the real world. But in any case, it's still super important and valuable

that you learned all of these three techniques now, because you will see them all in the real world still.

And this also allows you to think about this on your own and choose the style that you like the best,

but again, in the real world, and especially in modern JavaScript, you will mostly see ES6 classes being used now. And so, since I want to prepare you for the real world, I will start using classes from this point on.

## ❖ 14. Object-Oriented Programming (OOP) With JavaScript → 19. Another Class Example

```

class account {
  constructor(owner, currency, pin) {
    this.owner = owner;
    this.currency = currency;
    this.pin = pin;
    this.movements = [];
    this.locale = navigator.language;

    console.log(`Thanks ${owner}`);
  }
}

const acc1 = new account('jonas', 'EUR', 1111);

```

```

→ acc1.movements.push(250);
→ acc1.movements.push(-140);
console.log(acc1);

```

it's not a good idea at all to do this. So, instead of interacting with a property like this (interacting with properties directly for ex: `acc1.movements.push(-140);`), it's a lot better to create methods. And that is especially true for important properties, this will for sure avoid bugs in the future, as your application grows. So, let's now actually create a deposit and a withdrawal method here.

```

class account {
  constructor(owner, currency, pin) {
    this.owner = owner;
    this.currency = currency;
    this.pin = pin;
    this.movements = [];
    this.locale = navigator.language;

    console.log(`Thanks ${owner}`);
  }
}

```

```

⇒ // Public interface
deposit(val) {
  this.push(val);
  return this;
}

```

```

⇒ withdraw(val) {

```

```

    this.deposit(-val);
    return this;
  }
}

const acc1 = new account('jonas', 'EUR', 1111);

// acc1.movements.push(250);
// acc1.movements.push(-140);
⇒ acc1.deposit(250);
⇒ acc1.withdraw(250);

console.log(acc1);

```

simply the fact that we have these methods, doesn't make it impossible to still do this. And the same goes, for example, for the pin.

So, of course, we can access the pin from outside of the account. So, you see, but probably it shouldn't be accessible from outside of the class, but yeah,

of course, right now it is accessible. (encapsulation and data privacy in next session)

## ❖ 14. Object-Oriented Programming (OOP) With JavaScript → 20. Encapsulation Protected Properties and Methods

first, remember that encapsulation basically means to keep some properties and methods private inside the class so that they are not accessible from outside of the class. Then the rest of the methods are basically exposed as a public interface, which we can also call API.

So this is essential to do in anything more than a toy application.

Now, there are two big reasons why we need encapsulation and data privacy. So first it is to prevent code from outside of a class to accidentally manipulate or data inside the class. So this is also the reason why we implement a public interface. So we are not supposed to manually mess with this property and therefore we should encapsulate it.

Now, the second reason is that when we expose only a small interface so a small API consisting only of a few public methods then we can change all the other internal methods with more confidence. Because in this case, we can be sure that external code does not rely on these private methods. And so therefore our code will not break when we do internal changes.

So that's what encapsulation and data privacy are and the reasons for it.

And so let's now actually implement it. However JavaScript classes actually do not yet support real data privacy and encapsulation.

And so in this lecture, we will basically fake encapsulation

by simply using a convention. And for now, all we will do is to add this underscore in front of the property name and that's it.

Now we need to then go ahead and change it everywhere. So that's here and yeah, that's actually it. So again this does not actually make the property truly private because this is just a convention. So it's something that developers agree to use and then everyone does it this way.

But since it is not truly private we call this protected

```
class account {  
  constructor(owner, currency, pin) {  
    this.owner = owner;  
    this.currency = currency;  
    this.pin = pin;  
    → this._movements = [];
```

```
    this.locale = navigator.language;
```

```
    console.log(`Thanks ${owner}`);  
  }
```

```
  // Public interface
```

```
  deposit(val) {  
    this.push(val);  
    return this;  
  }
```

```
  withdraw(val) {  
    this.deposit(-val);  
    return this;  
  }  
}
```

```
const acc1 = new account('jonas', 'EUR', 1111);
```

```
→ acc1._movements.push(250); // still work work bot developers knows this is a protected  
property
```

```
→ acc1._movements.push(-140); // still work bot developers knows this is a protected  
property
```

```
acc1.deposit(250);  
acc1.withdraw(250);
```

Now, if we still wanted to give access to the movements array from the outside then we would have to implement a public method for that. So let's do that actually right here. So let's say get movements.

And of course we could also create a getter here instead of this method, but let's just keep it simple. So it's very common to actually have a method that is called get or set instead of using a real getter or a real setter.

And so all that this will do is to return this.\_movements.

```
// Public interface
getMovements() {
  return this.#movements;
}
```

And so this one everyone can still at least access the movements but they cannot override them.

```
console.log( acc1.sgetMovements())
```

## ❖ 14. Object-Oriented Programming (OOP) With JavaScript → 21. Encapsulation Private Class Fields and Methods

let's now implement truly private class fields and methods.

So private class fields and methods are actually part of a bigger proposal for improving and changing JavaScript classes which is simply called Class fields.

Now this Class fields proposal is currently at stage three. And so right now it's actually not yet part of the JavaScript language. However, being at stage three

means that it's very likely that at some point, it will move forward to stage number four.

And then it will actually become a part of the JavaScript language.

And that's probably gonna happen some point soon in the future.

And that's why I decided to already include class fields in this course.

And in fact, some parts of this proposal actually already work in Google Chrome, but other parts don't. At least not at the time of recording this video.

Now for starters, why is this proposal actually called Class fields?

Well, in traditional OOP languages like Java and C++, properties are usually called fields.

So what this means is that with this new proposal, JavaScript is moving away from the idea that classes are just syntactic sugar over constructor functions.

Because with this new class features classes actually start to have abilities that we didn't previously have with constructor functions.

Now to many developers consider this to be a big problem but personally, I'm not sure if it is such a big deal for the average JavaScript developer.

So as long as you still understand how prototypal inheritance and function constructors work then I believe that you will be fine.

But anyway, no matter if you end up using these new class features, let's now start exploring them.

So in this proposal, there are actually four different kinds of fields and methods, and actually it's even eight.

But in this video, I'm just gonna focus on these four:

- public fields
- private fields
- public methods
- private methods.
- there's also the static version of the same four

### public fields:

So we can think of a field as a property that will be on all instances.

So that's why we can also call this a public instance field.

So basically in our example here, the two fields could be the movements and the locale.

Because these are basically two properties

that are gonna be on all the objects that we create with this class.

```
locale = navigator.language;  
_movements = []
```

And then here, we actually need to write a semi colon which is kind of weird because between these methods we do not need commas or semi-colons. What's also weird is that this kind of looks like a variable here right? But we don't have to declare it using like const or let, these public fields here are gonna be present on all the instances that we are creating through the class. So they are not on the prototype.

So this is important to understand.

So all these methods that we add in the class (out of constructor)

they will always be added to the prototype, right?

But again the fields here, they are on the instances.

So `locale = navigator.language;` is exactly same as `this.locale = navigator.language;` (in the constructor)

And therefore these public fields they're also referenceable by the this keyword and they are also referenceable via the this keyword.

## private fields :

so that properties are really truly not accessible from the outside.

So just like in the last lecture, let's start

by now finally making the movements array private.

So let's move it here. And now I will remove the underscore

and then I will use the # symbol. And so this is the syntax that makes a field private in this new class proposal.

So let's try and reload to see what happens now.

And indeed we get some error. And the reason for that is that the property is now really called #movements. So we need to change that everywhere.

So both here and here. Now, that's it. Give it a safe and now it all works.

```
#movements = [];  
#pin;
```

So if we try to read account1.movements then we get a syntax error.

```
console.log(acc1.#movements); // Error cant accessible outside of class
```

Right now only Google Chrome actually supports these private class fields.

And so make sure to also test your code in Google Chrome. But anyway, the movements are now truly private and no longer accessible outside of class.

At least not by their property. We do still have of course, the getMovements() method in our public API. And so this one we can still use to get the movement.

Now the next candidate to make private is this pin.

So in the last lecture we protected it

but now just like the movements, we want to convert it

to a truly private field. However, this time the situation is a bit different.

Because now we are actually setting the pin based on this input value to the constructor.

However, we can not define a field in the constructor.

So the fields, they really have to be out here

outside of any method. So what we have to do is to create the field out here.

So the private field again with hash,

and then don't set it to anything. And so this is essentially just

like creating an empty variable. So in the beginning, this will be set to undefined and then here we can redefine that value basically.

And so one more time we can see that these class fields are really just

like any other property. That's why later down here we can then access it on the this keyword and set it to the value that we received. So let's try that.

And indeed the pin here is now also a protected field or actually a private field. And so when we try to access it, we will no longer be able to do that.

### public methods:

next up we have public methods.

And actually that is nothing new at this point. So all these methods here that we have been using are indeed public methods.

```
getMovements() {  
  return this.#movements;  
}  
  
deposit(val) {  
  this.#movements.push(val);  
  return this;  
}  
  
withdraw(val) {  
  this.deposit(-val);  
  return this;  
}
```

### private methods:

And private methods, as we already mentioned earlier are very useful to hide the implementation details from the outside. And that's why in the previous lecture, we already made this method and protect it with this underscore.

And so let's now grab this and put it down here and now to make a private method, the syntax is exactly the same as private fields. So just like with the hash.

Now, the big problem here, is that right now no browser actually supports this.

```
#approveLoan(val) {  
  return true;  
}
```

\*\*\*We don't have error but Google Chrome right now basically sees this as a private class field and not as a method.

### Static:



And so usually we use this for helper functions. Because these static methods will not be available on all the instances, but only on the class itself.

```
static helper() {  
  console.log('Helper');  
}
```

### Account.helper(

there is also a static version for all the other three ones.

But I'm not gonna show them to you now in this video

because they are really less important, and if you want you can, of course easily test them out by yourself.

### Review Of whole Code:

```
// Encapsulation: Protected Properties and Methods  
// Encapsulation: Private Class Fields and Methods
```

```
// 1) Public fields  
// 2) Private fields  
// 3) Public methods  
// 4) Private methods  
// (there is also the static version)
```

```
class Account {  
  // 1) Public fields (instances)  
  locale = navigator.language;  
  
  // 2) Private fields (instances)  
  #movements = [];  
  #pin;  
  
  constructor(owner, currency, pin) {  
    this.owner = owner;  
    this.currency = currency;  
    this.#pin = pin;  
  
    // Protected property  
    // this._movements = [];  
    // this.locale = navigator.language;  
  
    console.log(`Thanks for opening an account, ${owner}`);  
  }  
}
```

```

// 3) Public methods

// Public interface
getMovements() {
  return this.#movements;
}

deposit(val) {
  this.#movements.push(val);
  return this;
}

withdraw(val) {
  this.deposit(-val);
  return this;
}

requestLoan(val) {
  // if (this.#approveLoan(val)) {
  if (this._approveLoan(val)) {
    this.deposit(val);
    console.log(`Loan approved`);
    return this;
  }
}

static helper() {
  console.log('Helper');
}

// 4) Private methods
// #approveLoan(val) {
_approveLoan(val) {
  return true;
}
}

const acc1 = new Account('Jonas', 'EUR', 1111);

// acc1._movements.push(250);
// acc1._movements.push(-140);
// acc1.approveLoan(1000);

```

```

acc1.deposit(250);
acc1.withdraw(140);
acc1.requestLoan(1000);
console.log(acc1.getMovements());
console.log(acc1);
Account.helper();

// console.log(acc1.#movements);
// console.log(acc1.#pin);
// console.log(acc1.#approveLoan(100));

// Chaining
acc1.deposit(300).deposit(500).withdraw(35).requestLoan(25000).withdraw(4000);
console.log(acc1.getMovements());
*/

```

## ❖ 14. Object-Oriented Programming (OOP) With JavaScript → 22. Chaining Methods

Do you remember how we chained array methods one after another, for example filter map and reduce? So by chaining these methods, we could first filter an array, then map the result. And finally reduce the results of the map,

all in one line of code. And we can actually implement the same ability of chaining methods in the methods of our class. And so let's go do that now. And actually, this is extremely easy to do.

```

acc1.deposit(300).deposit(500).withdraw(35).requestLoan(25000).withdraw(4000);
console.log(acc1.getMovements());

```

we have error `acc1.deposit(300)` return nothing, Because , the deposit method does return undefined because we're not returning anything explicitly

so `.deposit(500)` called on an undefined, So what we need to do is to call deposit actually on an account.

And so what we want is for the result of this here to be an account, we add “return this” to methods. Because this is of course, the current object. And now we do the same (adding return this) on all of them.

**\*\* So all we have to do is to return the object itself at the end of a method that we want to be chainable.\*\***

```
class Account {
  // 1) Public fields (instances)
  locale = navigator.language;

  // 2) Private fields (instances)
  #movements = [];
  #pin;

  constructor(owner, currency, pin) {
    this.owner = owner;
    this.currency = currency;
    this.#pin = pin;

    console.log(`Thanks for opening an account, ${owner}`);
  }

  // 3) Public methods

  // Public interface
  getMovements() {
    return this.#movements;
  }

  deposit(val) {
    this.#movements.push(val);
    ⇒ return this;
  }

  withdraw(val) {
    this.deposit(-val);
    ⇒ return this;
  }

  requestLoan(val) {
    // if (this.#approveLoan(val)) {
    if (this._approveLoan(val)) {
      this.deposit(val);
      console.log(`Loan approved`);
    }
    ⇒ return this;
  }
}
```

```

}

static helper() {
  console.log('Helper');
}

// 4) Private methods
// #approveLoan(val) {
  _approveLoan(val) {
    return true;
  }
}

const acc1 = new Account('Jonas', 'EUR', 1111);

acc1.deposit(250);
acc1.withdraw(140);
acc1.requestLoan(1000);
console.log(acc1.getMovements());
console.log(acc1);
Account.helper();

// Chaining
acc1.deposit(300).deposit(500).withdraw(35).requestLoan(25000).withdraw(4000);
console.log(acc1.getMovements());

```

❖ 14. Object-Oriented Programming (OOP) With JavaScript → 23. ES6 Classes  
 Summary

**Public field (similar to property, available on created object)** → `university`

**Private fields (not accessible outside of class)** → `#studyHours`

**Static public field (available only on class)** → `static numSubjects`

**Call to parent (super) class (necessary with extend). Needs to happen before accessing this** → `super(fullName, birthYear)`

**Instance property (available on created object)** → `this.startYear`

**Redefining private field** → `this.#course`

**Public method** → `introduce()`

**Referencing private field and method** → `this.#makeCoffe()`

**Private method (⚠ Might not yet work in your browser. "Fake" alternative: \_ instead of #)** → `#makeCoffe()`

**Getter method** → `get testScore()`

**Setter method (use \_ to set property with same name as method, and also add getter)** → `set testScore(score)`

**Static method (available only on class. Can not access instance properties nor methods, only static ones)** → `static printCurriculum()`

**Creating new object with new operator** → `const student = new Student(...)`

**Parent class** → `Person`

**Inheritance between classes, automatically sets prototype** → `extends Person`

**Child class** → `Student`

**Constructor method, called by new operator. Mandatory in regular class, might be omitted in a child class** → `constructor(...)`

- Classes are just "syntactic sugar" over constructor functions
- Classes are **not** hoisted
- Classes are **first-class** citizens
- Class body is always executed in **strict mode**

## ❖ 14. Object-Oriented Programming (OOP) With JavaScript → 24. Coding Challenge #3 vs #4

// Coding Challenge #3

1. Use a constructor function to implement an Electric Car (called EV) as a CHILD "class" of Car. Besides a make and current speed, the EV also has the current battery charge in % ('charge' property);
2. Implement a 'chargeBattery' method which takes an argument 'chargeTo' and sets the battery charge to 'chargeTo';
3. Implement an 'accelerate' method that will increase the car's speed by 20, and decrease the charge by 1%. Then log a message like this: 'Tesla going at 140 km/h, with a charge of 22%';
4. Create an electric car object and experiment with calling 'accelerate', 'brake' and 'chargeBattery' (charge to 90%). Notice what happens when you 'accelerate'! HINT: Review the definition of polymorphism 😊

DATA CAR 1: 'Tesla' going at 120 km/h, with a charge of 23%

// Coding Challenge #4

1. Re-create challenge #3, but this time using ES6 classes: create an 'EVCI' child class of the 'CarCI' class
2. Make the 'charge' property private;
3. Implement the ability to chain the 'accelerate' and 'chargeBattery' methods of this class, and also update the 'brake' method in the 'CarCI' class. They experiment with chaining!

*DATA CAR 1: 'Rivian' going at 120 km/h, with a charge of 23%*

```

// Coding Challenge #3
const Car = function (make, speed) {
  this.make = make;
  this.speed = speed;
};

Car.prototype.accelerate = function () {
  this.speed += 10;
  console.log(`${this.make} is going at
${this.speed} km/h`);
};

Car.prototype.brake = function () {
  this.speed -= 5;
  console.log(`${this.make} is going at
${this.speed} km/h`);
};

const EV = function (make, speed,
charge) {
  Car.call(this, make, speed);
  this.charge = charge;
};

// Link the prototypes
EV.prototype =
Object.create(Car.prototype);

EV.prototype.chargeBattery = function
(chargeTo) {
  this.charge = chargeTo;
};

EV.prototype.accelerate = function () {
  this.speed += 20;
  this.charge--;
  console.log(
    `${this.make} is going at
${this.speed} km/h, with a charge of
${this.charge}`
  );
};

const tesla = new EV('Tesla', 120, 23);
tesla.chargeBattery(90);
console.log(tesla);
tesla.brake();
tesla.accelerate();

```

```

// Coding Challenge #4
class CarCl {
  constructor(make, speed) {
    this.make = make;
    this.speed = speed;
  }

  accelerate() {
    this.speed += 10;
    console.log(`${this.make} is going at
${this.speed} km/h`);
  }

  brake() {
    this.speed -= 5;
    console.log(`${this.make} is going at
${this.speed} km/h`);
    return this;
  }

  get speedUS() {
    return this.speed / 1.6;
  }

  set speedUS(speed) {
    this.speed = speed * 1.6;
  }
}

class EVCl extends CarCl {
  #charge;

  constructor(make, speed, charge) {
    super(make, speed);
    this.#charge = charge;
  }

  chargeBattery(chargeTo) {
    this.#charge = chargeTo;
    return this;
  }

  accelerate() {
    this.speed += 20;
    this.#charge--;
    console.log(
      `${this.make} is going at
${this.speed} km/h, with a charge of ${
        this.#charge
      }`
    );
    return this;
  }
}

```



```
// Coding Challenge #4 _ Continue

const rivian = new EVCl('Rivian', 120, 23);
console.log(rivian);
// console.log(rivian.#charge);
rivian
  .accelerate()
  .accelerate()
  .accelerate()
  .brake()
  .chargeBattery(50)
  .accelerate();

console.log(rivian.speedUS);
```

## ❖ 15. Mpty App OOP, Geolocation, External Libraries, and More! → 3. 4. How to Plan a Web Project

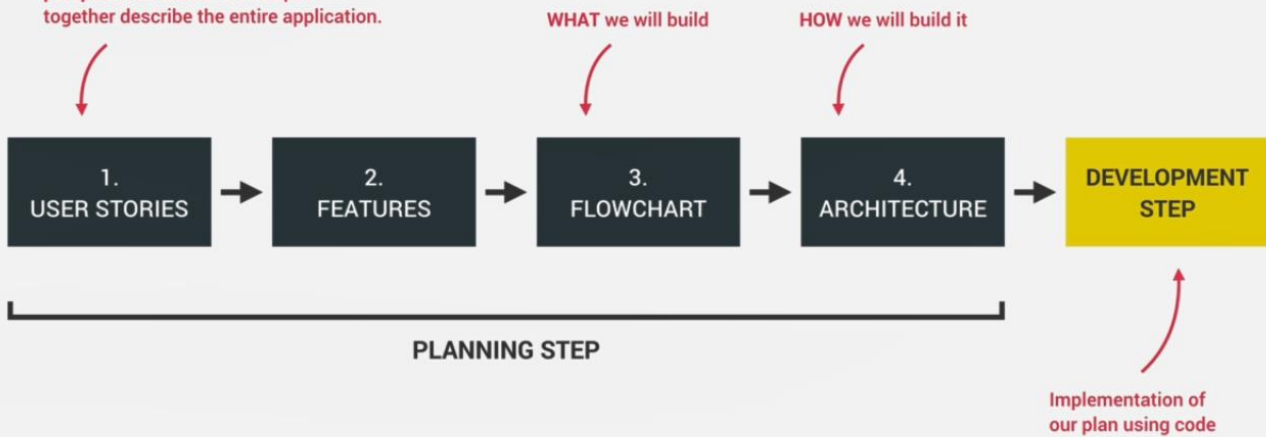
of the whole application, which will allow us developers to determine the exact features that we need to implement in order to make the user stories actually work as intended. Then to visualize the different actions that a user can take, and how the program react to these actions, we usually put all these features into a nice flow chart.

And again, we actually already used flow charts in some other applications that we built before, right? Now, once we know exactly what we're gonna build, it's time to think about how we're gonna build it. And this brings us to the project's architecture. And in this context, architecture simply means how we will organize our code, and what JavaScript features we will use. So the project's architecture is essentially what holds all the code together.

It gives us a structure in which we can then develop the application's functionality.

# PROJECT PLANNING

Description of the application's functionality from the user's perspective. All user stories put together describe the entire application.



Udacity

## 1. USER STORIES



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

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

Who?

What?

Why?

Example: user, admin, etc.

- 1 As a user, I want to log my running workouts with location, distance, time, pace and steps/minute, so I can keep a log of all my running
- 2 As a user, I want to log my cycling workouts with location, distance, time, speed and elevation gain, so I can keep a log of all my cycling
- 3 As a user, I want to see all my workouts at a glance, so I can easily track my progress over time
- 4 As a user, I want to also see my workouts on a map, so I can easily check where I work out the most
- 5 As a user, I want to see all my workouts when I leave the app and come back later, so that I can keep using there app over time

Udacity

## 2. FEATURES

### USER STORIES

- 1 Log my **running** workouts with location, distance, time, pace and steps/minute
- 2 Log my **cycling** workouts with location, distance, time, speed and elevation gain
- 3 See all my workouts at a glance
- 4 See my workouts on a map
- 5 See all my workouts when I leave the app and come back later

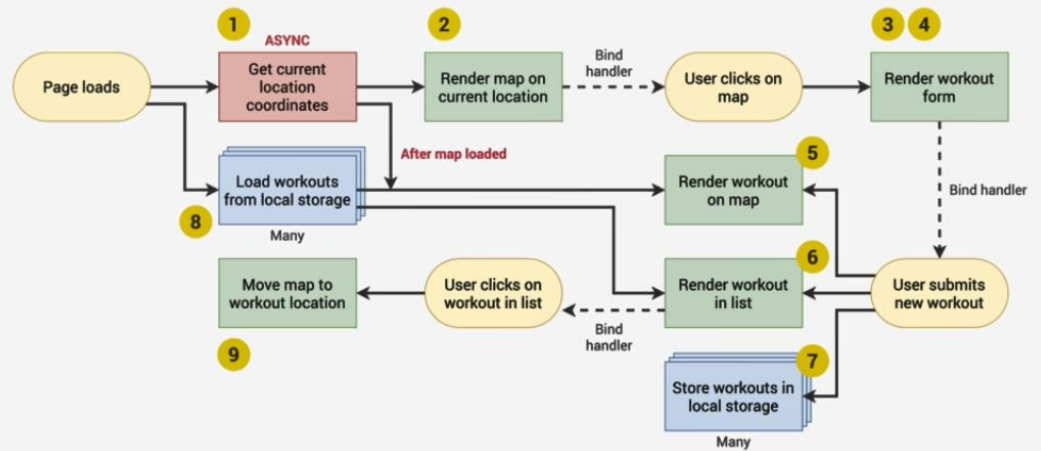
### FEATURES

- Map where user clicks to add new workout (best way to get location coordinates)
- Geolocation to display map at current location (more user friendly)
- Form to input distance, time, pace, steps/minute
- Form to input distance, time, speed, elevation gain
- Display all workouts in a list
- Display all workouts on the map
- Store workout data in the browser using local storage API
- On page load, read the saved data from local storage and display

## 3. FLOWCHART

### FEATURES

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



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

FOR NOW, LET'S JUST  
START CODING 🥰

❖ 15. Mpty App OOP, Geolocation, External Libraries, and More! → 5. Using the Geolocation API

To get user location use `navigator.geolocation.getCurrentPosition()`. This function here takes as an input 2 Callback functions. And the first one is to Callback function that will be called on success. So whenever the browser successfully got the coordinates of the current position of the user and to second callback is the Error Callback

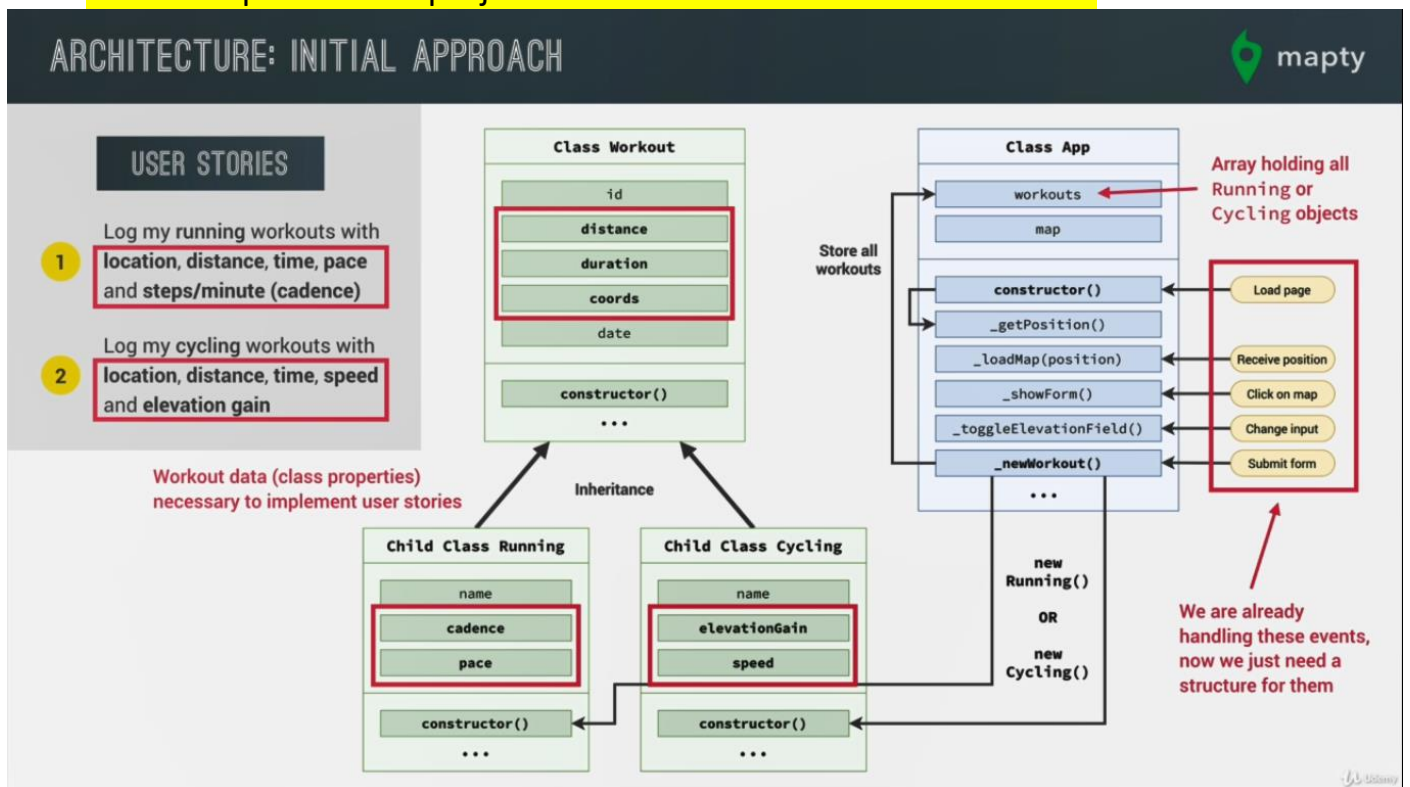
which is the one that is gonna be called when there happened an error while getting the coordinates.

```
if (navigator.geolocation) {  
  navigator.geolocation.getCurrentPosition(  
    function (position) {  
      const { latitude } = position.coords; //using destructuring (detect same named variable  
on object ant put it to created variable)  
      const { longitude } = position.coords;  
  
      const sampleGoogleMapsPosition =  
`https://www.google.com/maps/@${latitude},${longitude}`; // ceateing sample google  
map valid link  
      console.log(sampleGoogleMapsPosition);  
    },  
  ),  
}
```

```
function () {
  console.log('Could not get your position');
}
);
}
```

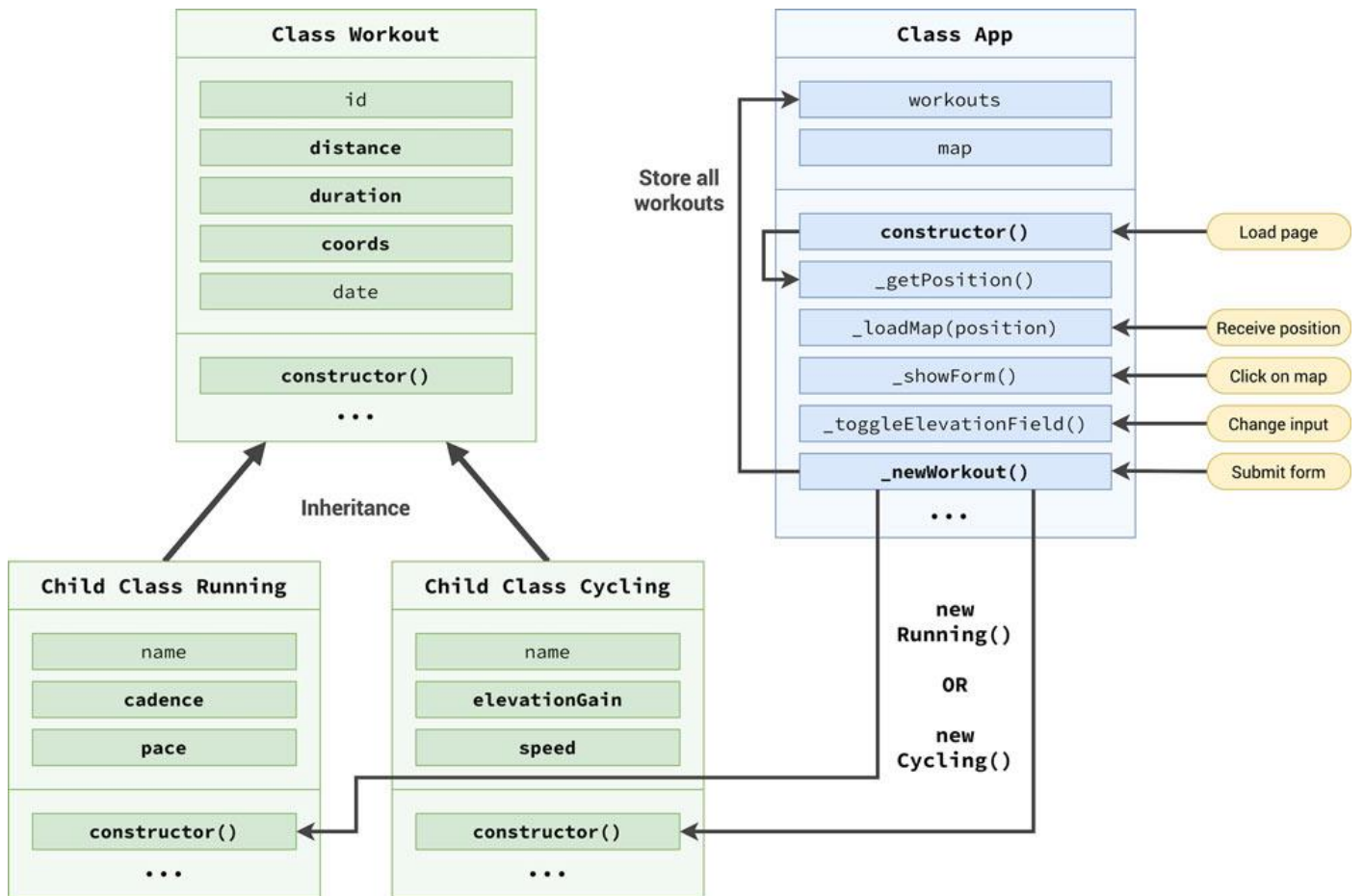
❖ 15. Mapty App OOP, Geolocation, External Libraries, and More! → 9. Project Architecture

Has some point about project architecture – retake and take notes

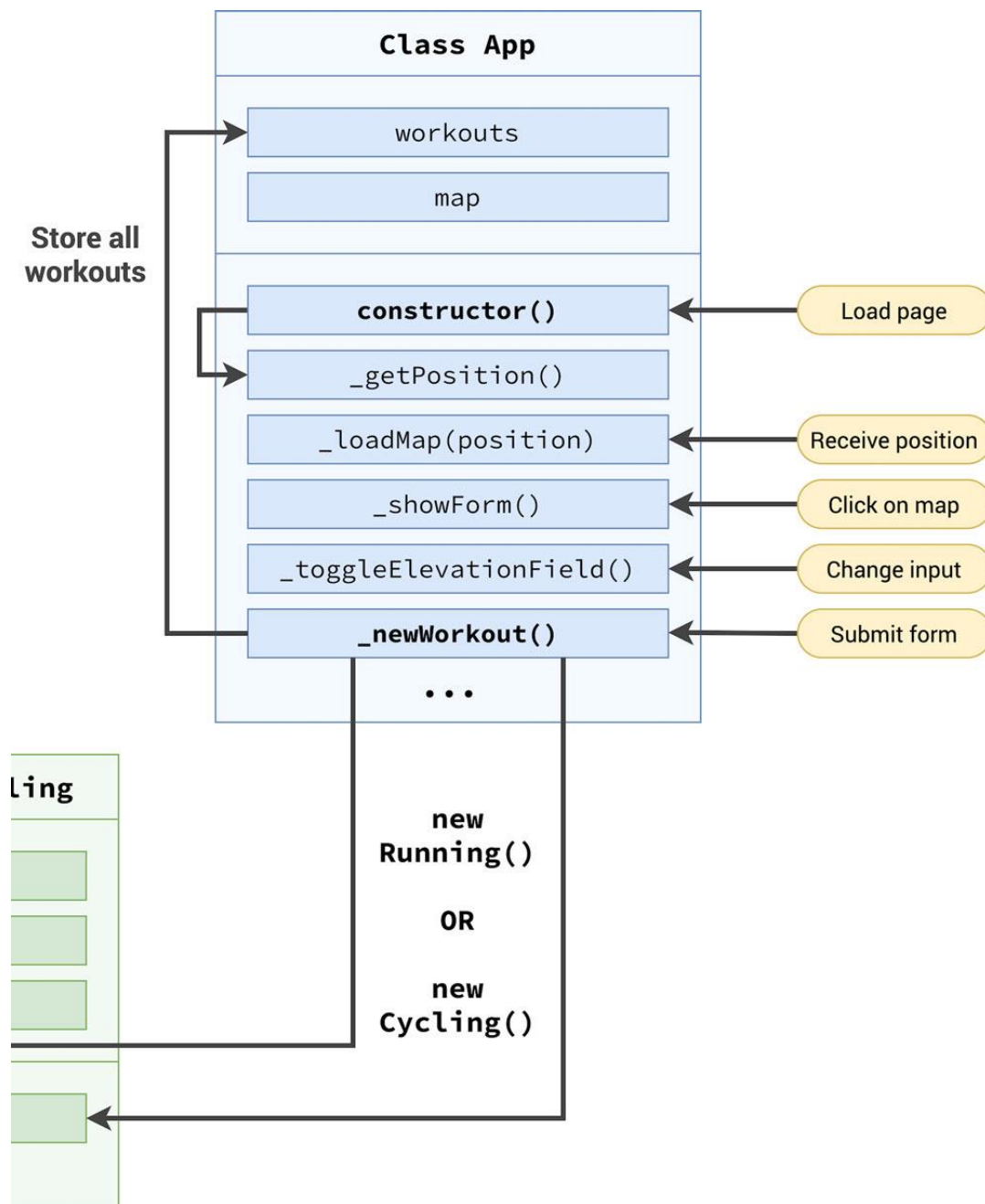


❖ 15. Mapty App OOP, Geolocation, External Libraries, and More! → 10. Refactoring for Project Architecture

Mapty architecture part-1 :



We work on this part in this video:



**Before Refactoring:**

```
const form = document.querySelector('.form');
const containerWorkouts = document.querySelector('.workouts');
const inputType = document.querySelector('.form__input--type');
const inputDistance = document.querySelector('.form__input--distance');
const inputDuration = document.querySelector('.form__input--duration');
const inputCadence = document.querySelector('.form__input--cadence');
```

```

const inputElevation = document.querySelector('.form__input--elevation');
let map, mapEvent;

if (navigator.geolocation) {
  navigator.geolocation.getCurrentPosition(
    function (position) {
      const { latitude } = position.coords; //using destructuring (detect same named variable on object ant put it to created variable)
      const { longitude } = position.coords;

      const sampleGoogleMapsPosition =
`https://www.google.com/maps/@${latitude},${longitude}`; // ceateing sample google map valid link
      console.log(sampleGoogleMapsPosition);

      const coords = [latitude, longitude];
      map = L.map('map').setView(coords, 13);

      L.tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
        attribution:
          '&copy; <a href="https://www.openstreetmap.org/copyright">OpenStreetMap</a>
contributors',
      }).addTo(map);

      // Handling click on map
      map.on('click', function (mapE) {
        mapEvent = mapE;
        form.classList.remove('hidden');
        inputDistance.focus();
      });
    },
    function () {
      console.log('Could not get your position');
    }
  );
}

form.addEventListener('submit', function (e) {
  e.preventDefault();

  // clear input
  inputCadence.value =
  inputDuration.value =

```



```

inputCadence.value =
inputElevation.value =
  ";
const { lat, lng } = mapEvent.latlng;
L.marker([lat, lng])
  .addTo(map)
  .bindPopup(
    L.popup({
      maxWidth: 250,
      minWidth: 100,
      autoClose: false,
      closeOnClick: false,
      className: 'running-popup',
    })
  )
  .setPopupContent('Workout')
  .openPopup();
});

inputType.addEventListener('change', function () {
  inputElevation.closest('.form__row').classList.toggle('form__row--hidden');
  inputCadence.closest('.form__row').classList.toggle('form__row--hidden');
});

```

### After Refactoring:

```

const form = document.querySelector('.form');
const containerWorkouts = document.querySelector('.workouts');
const inputType = document.querySelector('.form__input--type');
const inputDistance = document.querySelector('.form__input--distance');
const inputDuration = document.querySelector('.form__input--duration');
const inputCadence = document.querySelector('.form__input--cadence');
const inputElevation = document.querySelector('.form__input--elevation');

class App {
  #map;
  #mapEvent;
  constructor() {
    this._getPosition();
    form.addEventListener('submit', this._newWorkout.bind(this));
    inputType.addEventListener('change', this._toggleElevationField);
  }
}

```

```

_getPosition() {
  if (navigator.geolocation)
    navigator.geolocation.getCurrentPosition(
      this._loadMap.bind(this),
      function () {
        console.log('Could not get your position');
      }
    );
}

_loadMap(position) {
  const { latitude } = position.coords; //using destructuring (detect same named variable on object ant put it to created variable)
  const { longitude } = position.coords;

  const sampleGoogleMapsPosition =
`https://www.google.com/maps/@${latitude},${longitude}`; // ceateing sample google map valid link
  console.log(sampleGoogleMapsPosition);

  const coords = [latitude, longitude];
  this.#map = L.map('map').setView(coords, 13);

  L.tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
    attribution:
      '&copy; <a href="https://www.openstreetmap.org/copyright">OpenStreetMap</a>
    contributors',
  }).addTo(this.#map);

  // Handling click on map
  this.#map.on('click', this._showForm.bind(this));
}

_showForm(mapE) {
  this.#mapEvent = mapE;
  form.classList.remove('hidden');
  inputDistance.focus();
}

_toggleElevationField() {
  inputElevation.closest('.form__row').classList.toggle('form__row--hidden');
  inputCadence.closest('.form__row').classList.toggle('form__row--hidden');
}

_newWorkout(e) {

```

```

e.preventDefault();

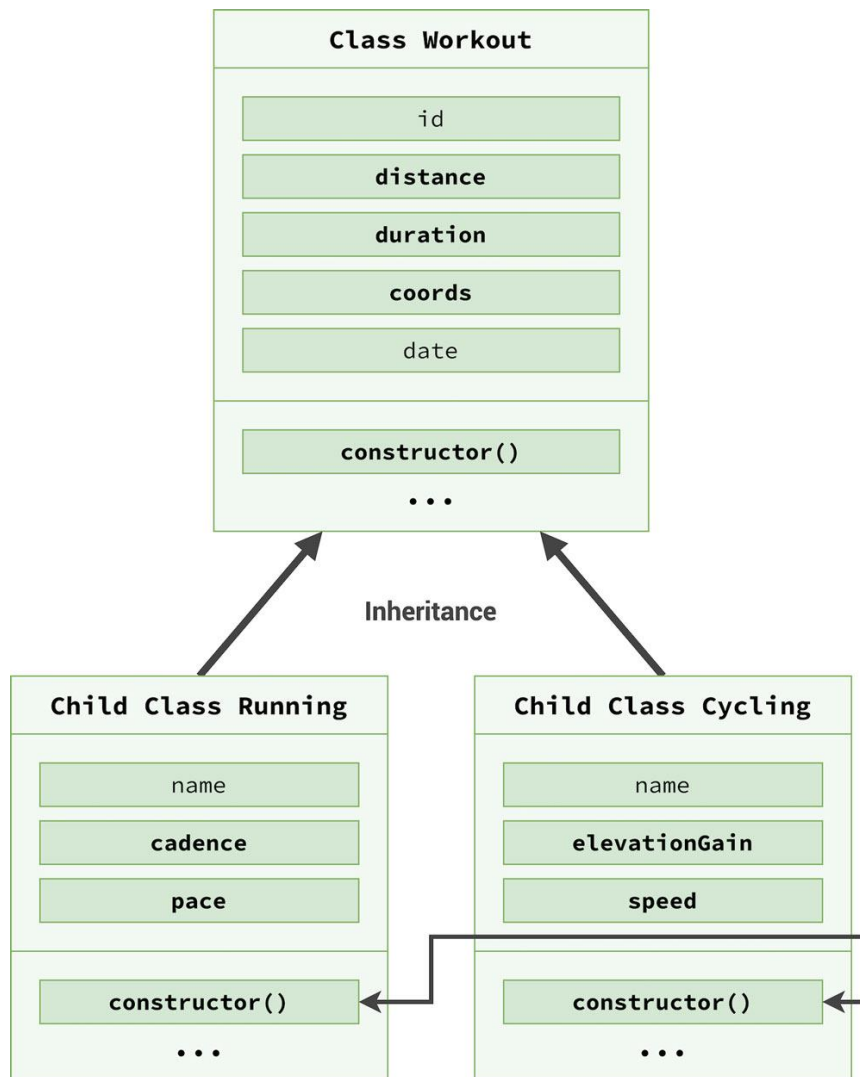
// clear input
inputCadence.value =
  inputDuration.value =
  inputCadence.value =
  inputElevation.value =
  "";
const { lat, lng } = this.#mapEvent.latlng;
L.marker([lat, lng])
  .addTo(this.#map)
  .bindPopup(
    L.popup({
      maxWidth: 250,
      minWidth: 100,
      autoClose: false,
      closeOnClick: false,
      className: 'running-popup',
    })
  )
  .setPopupContent('Workout')
  .openPopup();
}
}

const app = new App();

```

## ❖ 15. Mapty App OOP, Geolocation, External Libraries, and More! → 11. Managing Workout Data Creating Classes

So in this video we're gonna implement classes to manage the data about our cycling and running workouts that are coming from the user interface.



❖ 15. Mapty App OOP, Geolocation, External Libraries, and More! → 12. Creating a New Workout

\*\* the if else is not really that much used anymore. And I think that actually it looks a lot cleaner like this

so to simply have 2 if statements.

\*\* guard clause means is that we will basically check for the opposite of what we are originally interested in

and if that opposite is true, then we simply return the function immediately.

**Development video**

❖ 15. Mapty App OOP, Geolocation, External Libraries, and More! → 13. Rendering Workouts

**Development video**

\*\*\* .toFixed(1) fix decimal part to 1 digit??

❖ **15. Mapty App OOP, Geolocation, External Libraries, and More! → 14. Move to Marker On Click**

**Development video**

❖ **15. Mapty App OOP, Geolocation, External Libraries, and More! → 15. Working with localStorage**

ocal storage is basically a place in the browser, where we can store data that will stay there even after we close the page. So basically the data is basically linked to the URL on which we are using the application.

\*\* so it is only advised to use for small amounts of data that's because local storage is blocking

```
localStorage.setItem('name','value')// set data to local Storage
```

```
localStorage.getItem('name');// get data from local storage
```

```
localStorage.removeItem('name') // remove data from local storage based on key
```

```
JSON.stringify(Obj) // convert any object in JavaScript to a string
```

```
JSON.parse(localStorage.getItem('name')) // JSON.parse: convert local data to object
```

**\*\* when we converted our objects to a string (like when we store and recover local storage data), and then back from the string to objects, we lost the prototype chain.**

```
location.reload() // reload the page
```

\*\* take look at **location** object and its methods

❖ **16. Asynchronous JavaScript Promises, AsyncAwait, and AJAX → 3. Asynchronous JavaScript, AJAX and APIs**

what asynchronous JavaScript actually is, and also learn about the most popular use cases of asynchronous JavaScript, which is basically to make so-called Ajax calls to APIs.

Now, to understand what asynchronous JavaScript code actually is, we first need to understand what synchronous code is. So basically the opposite of asynchronous. So most of the codes that we've been writing so far in the course

has been synchronous code, and synchronous simply means that the code is executed line by line, in the exact order of execution that we defined in our code, just like in this small example. So as the first line of code is reached in the execution, it is simply executed in the execution of thread. Now don't worry about this execution of thread. It's not important here, it's just to make a point of synchronous versus asynchronous code, as you will see in the next slide. All you need to know is that the execution of thread is part of the execution context, which does actually execute the code in the computer's processor. But anyway, then the next line of code is executed and then the next one, all in sequence. So each line of code always waits for the previous line to finish execution. Now this can create problems when one line of code takes a long time to run. For example, in this current line of code, we have an alert statement, which creates this alert window. Now, as we've experienced in the past, this alert window will block the code execution, right? So nothing will happen on the page until we click that OK Button. And only then, the code can continue executing. And so the alert statement is a perfect example of a long running operation, which blocks execution of the code.

SYNCHRONOUS CODE

BLOCKING

THREAD OF EXECUTION

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

SYNCHRONOUS

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

So again, only after we click Okay, the window disappears and the next line can run.

So this is hopefully a nice illustration of the problem with synchronous code. Now, most of the time synchronous code is fine and makes perfect sense. But imagine that execution would have to wait for example, for a five second timer to finish. That would just be terrible, right? Because meanwhile, nothing on the page would work during these five seconds. And so that's where asynchronous code comes into play.

## SYNCHRONOUS CODE

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

**BLOCKING**

**THREAD OF EXECUTION**

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

**SYNCHRONOUS**

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

So this example contains the five-second timer that I just mentioned. And if you want, you can pause the video here for a minute and analyze this code before we move on. Now, anyway, the first line of code is still synchronous here, and we also move to the second line in a synchronous way.

## ASYNCHRONOUS CODE

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

Example: Timer with callback



Udacity

But here we encountered the set timeout function, which will basically start a timer in an asynchronous way.

So this means that the timer will essentially run in the background without preventing the main code from executing. We also register a callback function, which will not be executed now,

but only after the timer has finished running. And we have actually already done this many times

before in practice, right? Now this callback function that I just mentioned is asynchronous JavaScript.

And it is asynchronous because it's only going to be executed after a task that is running in the background finishes execution. And in this case, that is the timer. So this callback that we just talked about is registered, and then we immediately move on to the next line.

So the main code is not being blocked and execution does not wait for the asynchronous timer

to finish its work. And that's the big difference between synchronous and asynchronous code.

So previously we had to wait for the user to click on the alert window to continue execution.

And again, that's because alert is blocking synchronous code, but now with this timer,



the callback is actually asynchronous. And so it's only going to be executed after the timer has finished.

And so therefore we say, that it's non-blocking because in the meantime, the rest of the code can keep running normally. Now, when the timer finally finishes after five seconds,

## ASYNCHRONOUS CODE

Asynchronous

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

CALLBACK WILL RUN AFTER TIMER

Example: Timer with callback

THREAD OF EXECUTION

"BACKGROUND"

Timer running

(More on this in the lecture on Event Loop)

- Asynchronous code is executed **after a task that runs in the "background" finishes;**
- Asynchronous code is **non-blocking;**
- Execution doesn't wait for an asynchronous task to finish its work;

### ASYNCHRONOUS

Now, when the timer finally finishes after five seconds, the callback function will finally be executed as well.

So you'll see that this callback runs after all the other code, even though in the code, it doesn't appear at the end. And so basically an action was deferred

into the future here in order to make the code non-blocking.

And actually we already saw this behavior happening before when we first learned about timers,

we just didn't know that this is called asynchronous and non-blocking code.

So in summary, asynchronous programming is all about coordinating the behavior of our program

over a certain period of time. And this is essential to understand. So asynchronous literally means

not occurring at the same time. And so that's what asynchronous programming is all about.

All right.

# ASYNCHRONOUS CODE



## ASYNCHRONOUS

Coordinating behavior of a program over a period of time

- Asynchronous code is executed **after** a task that runs in the **"background"** finishes;
- Asynchronous code is **non-blocking**;
- Execution doesn't wait for an asynchronous task to finish its work;

Udacity

\*\*

Now, as we saw in this example, we need a callback function to implement this asynchronous behavior, right?

However, that does not mean that callback functions automatically make code asynchronous.

That is just not the case, okay? For example, the Array map method accepts a callback function as well, but that doesn't make the code asynchronous. Only certain functions such as set timeout work in an asynchronous way.

We just have to know which ones do and which ones don't, okay? But please understand this very important fact that callback functions alone do not make code asynchronous, that's essential to keep in mind.

\*\*

let's see another example. So this example is about loading an image.

So the first two lines run in a synchronous way, one after the other. Now in the second line, we set the source attribute of the image that we selected in the first line. And this operation is actually asynchronous.

So setting the source attribute of any image is essentially loading an image in the background

while the rest of the code can keep running. And this makes sense, right?

Imagine that it's a huge image, we wouldn't want our entire code to wait for the image to load.

And that's why setting the source attribute was implemented in JavaScript in an asynchronous way. Now, once the image has finished loading, a load event will automatically be emitted by JavaScript. And so we can then listen for that event in order to act on it. Listening for the load event

is exactly what we do here in the next line as well. So here we use add event listener and to register a callback function

for the load event. So just like in the previous example, we provide a callback function that will be executed

once the image has been loaded and not right away, because again, all this code is non-blocking.

So instead of blocking, execution moves on right to the next line immediately.

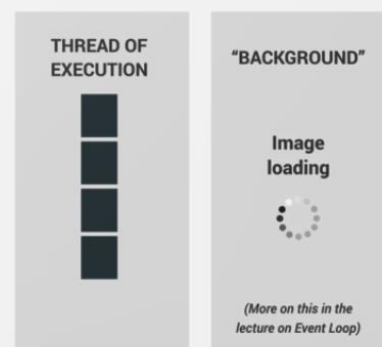
## ASYNCHRONOUS CODE

Asynchronous

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

CALLBACK WILL RUN AFTER IMAGE LOADS

Example: Asynchronous image loading with event and callback



## ASYNCHRONOUS

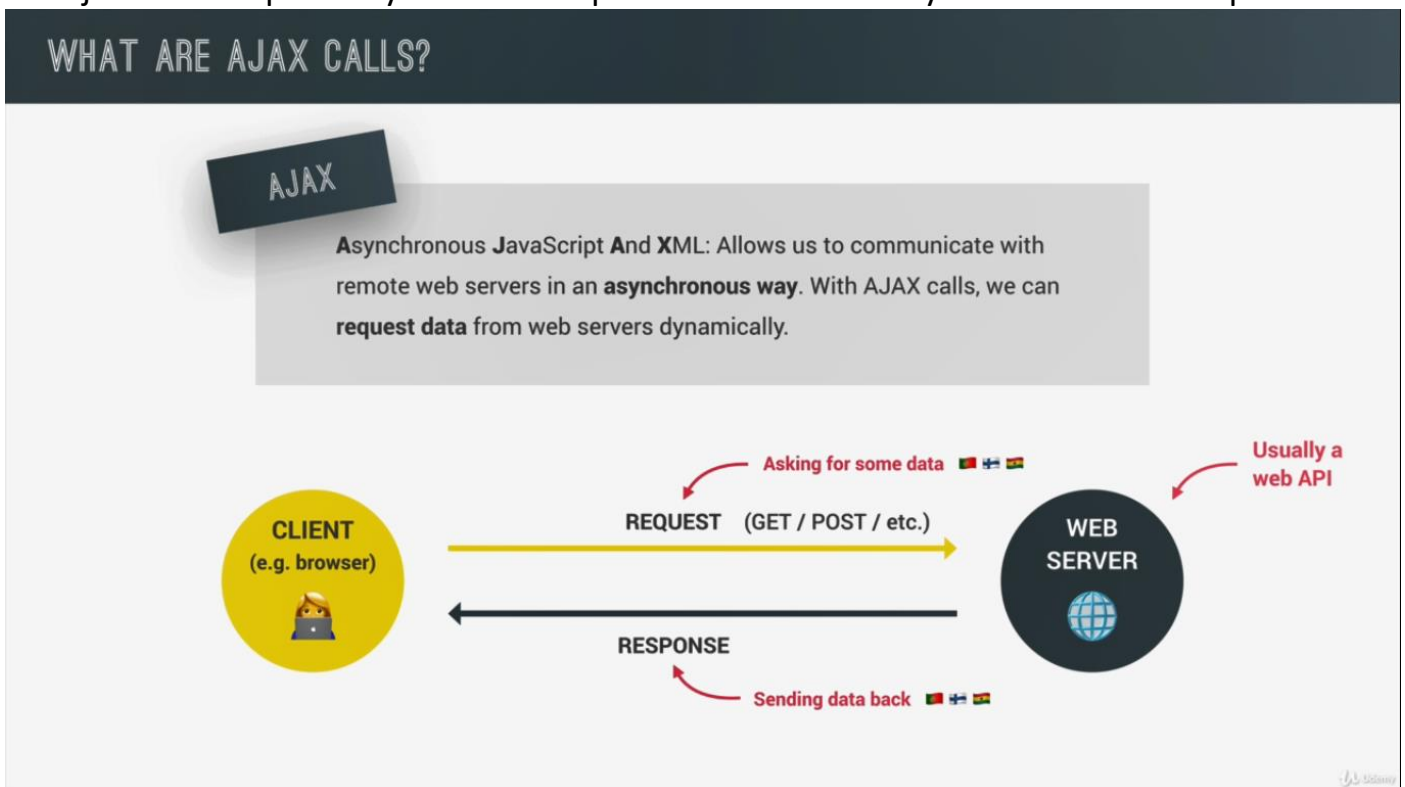
Coordinating behavior of a program over a period of time

- Asynchronous code is executed **after** a task that runs in the "background" finishes;
- Asynchronous code is **non-blocking**;
- Execution doesn't wait for an asynchronous task to finish its work;
- Callback functions alone do **NOT** make code asynchronous!

\*\*\*

the fact that event listeners alone do not make code asynchronous, just like callback functions alone, do also not make code asynchronous. For example, an event listener listening for a click on a button is not doing any work in the background. It's simply waiting for a click to happen, but it's not doing anything. And so there is no asynchronous behavior involved at all.

Ajax calls are probably the most important use case of asynchronous JavaScript:



let's now see what an API and web APIs actually are:

# WHAT IS AN API?

## API

➤ **Application Programming Interface:** Piece of software that can be used by another piece of software, in order to allow applications to talk to each other;

➤ There are be many types of APIs in web development:

DOM API    Geolocation API    Own Class API    **"Online" API**

➤ **"Online" API:** Application running on a server, that receives requests for data, and sends data back as response;

➤ We can build **our own** web APIs (requires back-end development, e.g. with node.js) or use **3rd-party** APIs.

There is an API for everything

AJAX

~~XML~~

XML data format



JSON data format

```
{
  "publisher": "101 Cookbooks",
  "title": "Best Pizza Dough Ever",
  "source_url": "https://www.101cookbooks.com/recipes_id/30794/",
  "image_url": "https://static.101cookbooks.com/social_thumb/101_cookbooks_pizza_dough.jpg",
  "publisher_url": "https://www.101cookbooks.com/"
}
```

Most popular API data format

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

\*\*online APIs → I came up with myself

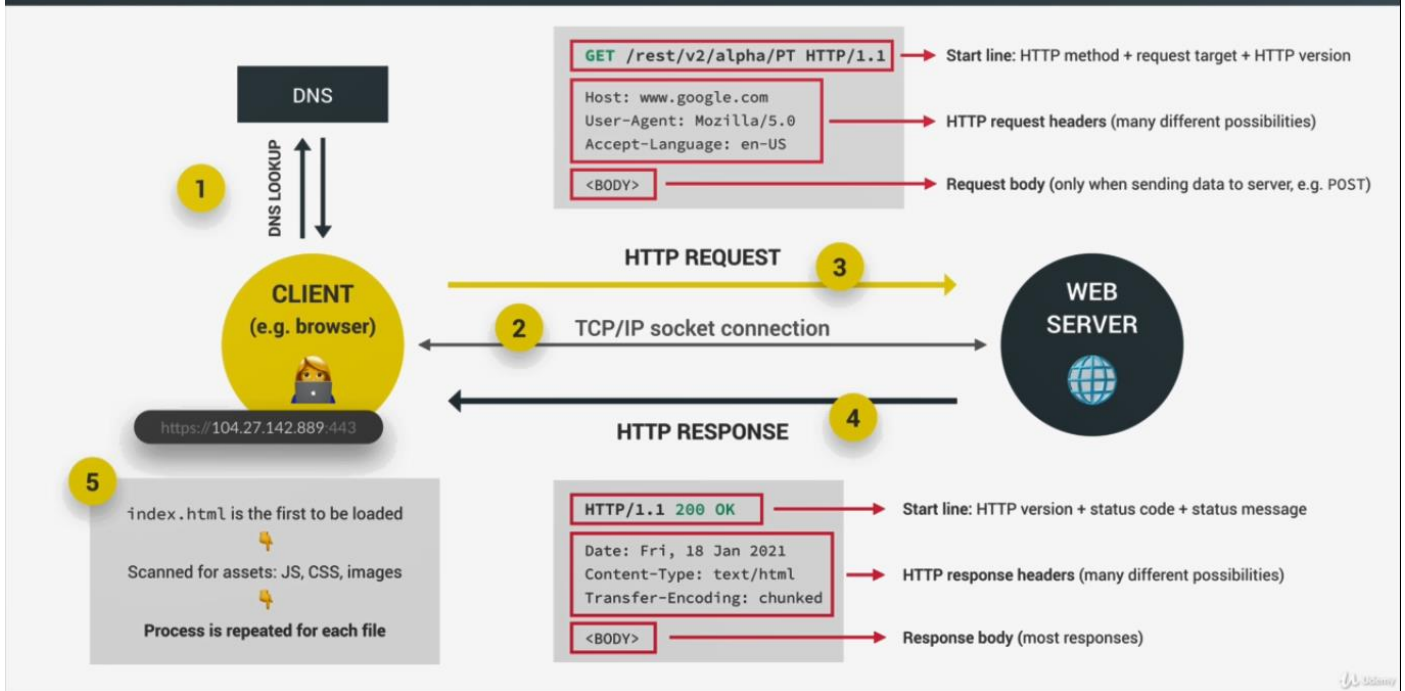
## ❖ 16. Asynchronous JavaScript Promises, AsyncAwait, and AJAX → 4. 4. Our First AJAX Call XMLHttpRequest

```
const getTotalData = function () {
  const req = new XMLHttpRequest();
  req.open('GET', 'https://restcountries.com/v3.1/all');
  req.send();

  req.addEventListener('load', function () {
    const myData = JSON.parse(this.responseText);
    console.log(myData);
  });
};
```

## ❖ 16. Asynchronous JavaScript Promises, AsyncAwait, and AJAX → 5. [OPTIONAL] How the Web Works Requests and Responses

# WHAT HAPPENS WHEN WE ACCESS A WEB SERVER



## ❖ 16. Asynchronous JavaScript Promises, AsyncAwait, and AJAX → 6. Welcome to Callback Hell

we have one AJAX call here that depends on another one. And so what we have here is one callback function

inside of another one. So you see here, we attach the first callback function. And then inside of that, we have yet another one. So in other words, here, we have nested callbacks. But now imagine that we wanted

to do more requests in sequence, like the neighbor of the neighbor of the neighbor, and like 10 times over.

So in that case, we would end up with callbacks inside of callbacks inside of callbacks, like 10 times.

And actually, for that kind of structure. And for that kind of behavior, we have a special name.

And that special name is callback hell. So basically, callback hell is when we have a lot of nested callbacks

in order to execute asynchronous tasks in sequence. And in fact, this happens for all asynchronous tasks,

which are handled by callbacks. And not just AJAX calls.

So for example, let's say we have a set timeout function. And then here, we want to log something to the Console,

like 1 second passed, but then also, we want to start another timeout. Let's just set 1000 here.

So let's copy this one here. And so as I just said, let's say, that here we wanted to start a new timer

after the first timer has finished. So here, we can say 2 seconds passed. And we can even add another one.

So never mind about what just happened in the Console.

And here, why not add yet another one? And so now they should appear like one, two, three, four.

And so here to of course, we have callback hell. And in fact, callback hell is pretty easy to identify

by this triangular shape that is formed here, you see that. And also the same is starting to appear here.

So if we had more callbacks in here, then we would start to see a lot more indentation here. And then this triangular shape, would also appear there. Now, the problem with callback hell

is that it makes our code look very messy. But even more important, it makes our code harder to maintain, and very difficult to understand, and to reason about, And fortunately for us, since ES6, there is actually a way of escaping callback hell by using something called promises.

## ❖ 16. Asynchronous JavaScript Promises, AsyncAwait, and AJAX → 7. Promises and the Fetch API

### WHAT ARE PROMISES?

#### PROMISE

👉 **Promise:** An object that is used as a placeholder for the future result of an asynchronous operation.

↓ Less formal

👉 **Promise:** A container for an asynchronously delivered value.

↓ Less formal

👉 **Promise:** A container for a future value.

Example: Response from AJAX call

👉 We no longer need to rely on events and callbacks passed into asynchronous functions to handle asynchronous results;

👉 Instead of nesting callbacks, we can **chain promises** for a sequence of asynchronous operations: **escaping callback hell** 🦄



**Promise** that I will receive money if I guess correct outcome

👉 I buy lottery ticket (promise) right now



🎰 Lottery draw happens asynchronously



💰 If correct outcome, I receive money, because it was promised

since promises work with asynchronous operations, they are time sensitive. So they change over time.

And so promises can be in different states and this is what they call the cycle of a promise.

So in the very beginning, we say that a promise is pending.

And so this is before any value resulting from the asynchronous task is available.

Now, during this time, the asynchronous task is still doing its work

in the background. Then when the task finally finishes,

we say that the promise is settled and there are two different types of settled promises and that's fulfilled promises and rejected promises. So a fulfilled promise is a promise that has successfully resulted in a value just as we expect it. For example, when we use the promise to fetch data

from an API, a fulfilled promise successfully gets that data,

and it's now available to being used. On the other hand, a rejected promise means

that there has been an error during the asynchronous task. And the example of fetching data from an API,

an error would be for example, when the user is offline and can't connect

to the API server. Now going back to the analogy of our lottery ticket,

the lottery draw is basically the asynchronous task, which determines the result.

Then once the result is available, the ticket would be settled. Then if we guessed the correct outcome,

the lottery ticket will be fulfilled and we get our money. However, if we guessed wrong,

then the ticket basically gets rejected. And all we did was waste our money.

Now these different states are very important to understand because when we use promises in our code,

we will be able to handle these different states in order to do something as a result

of either a successful promise or a rejected one. Another important thing about promises is that

a promise is only settled once. And so from there, the state will remain unchanged forever.

So the promise was either fulfilled or rejected, but it's impossible to change that state.

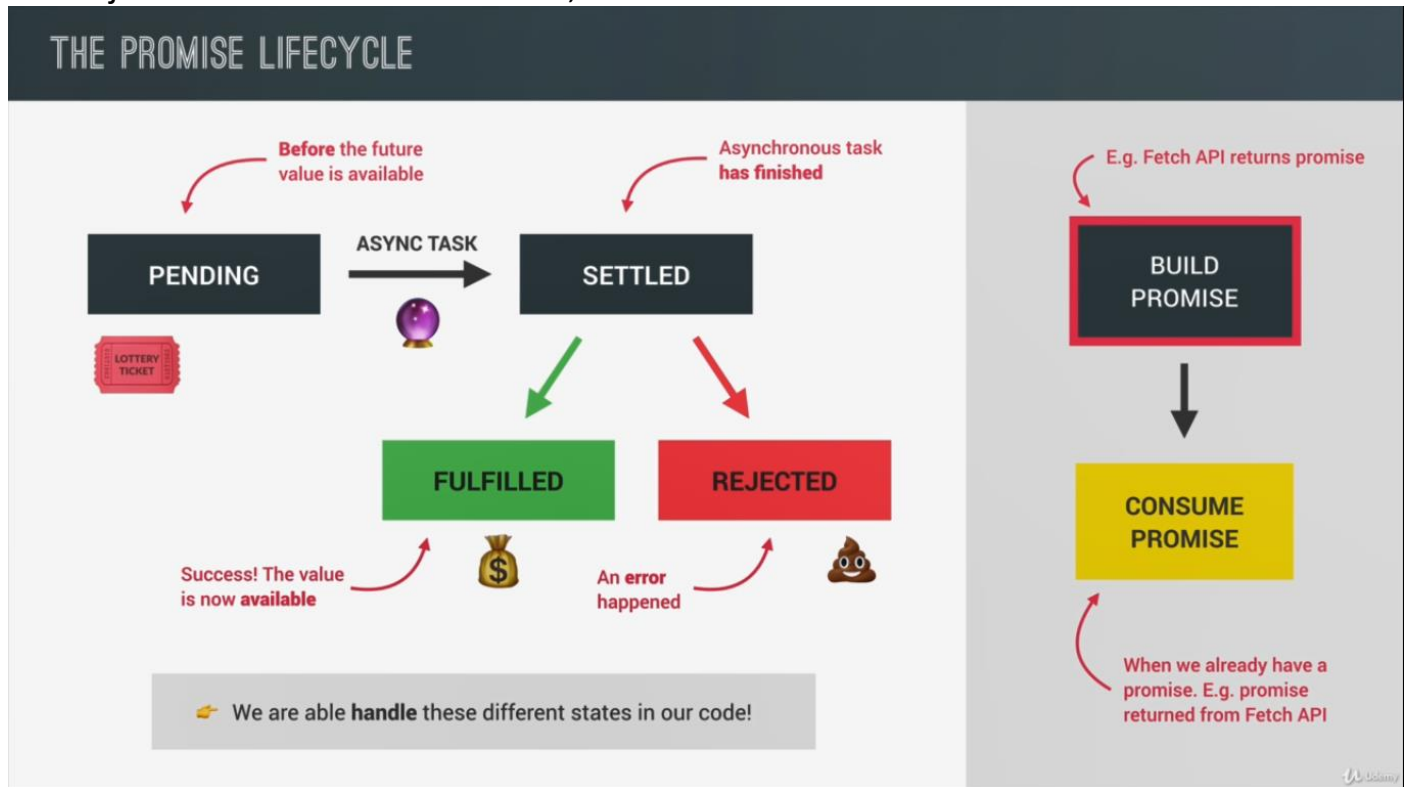
Now, these different states that I showed you here are relevant and useful when we use a promise to get a result,

which is called, to consume a promise. So we consume a promise when we already have a promise, for example, the promise that was returned from the fetch function, right at the beginning of this video, remember. But in order for a promise to exist in the first place, it must first be built. So it must be created in the case of the fetch API, it's the fetch function

that builds the promise and returns it for us to consume. So in this case, we don't have to build the promise ourselves in order to consume it. Now, most of the time we will actually



just consume promises, which is also the easier and more useful part. And so that's what we will do in the next couple of videos. But sometimes we also need to build a promise and to not just consume it. And of course, we will also learn how to do that a bit later.



## ❖ 16. Asynchronous JavaScript Promises, AsyncAwait, and AJAX → 8. Consuming Promises

we will consume the promise that was returned by the fetch function.

calling the fetch function like this, will then immediately return a promise.

So as soon as we start the request and in the beginning, this promise is of course still pending because the asynchronous task of getting the data, is still running in the background.

So just as we learned in the last lecture. Now, of course, at a certain point the promise will then be settled and either in a fulfilled or in a rejected state, but for now let's assume success.

**So assume that the promise will be fulfilled and that we have a value available to work with.** And so to handle this fulfilled state, we can use the then method that is available on all promises. Now into the then method, we need to pass a callback function that we want to be executed as soon as the promise is actually fulfilled. So as soon as the result is available. So a function and then this function will actually receive one argument once it's called by JavaScript and that argument is the resulting value of the fulfilled promise.

```

const getCountryData = function (country) {
  fetch(`https://restcountries.com/v3.1/name/${country}`).then(function (
    response
  ) {
    console.log(response);
  });
};

getCountryData('iran');

```

So in order to be able to actually read this data from the body, we need to call the json method on the response. Okay, so json is a method that is available on all responses of the fetch method. So what I mean is this, so response dot json. the problem here is that this json function itself, is actually also an asynchronous function. And so what that means, is that it will also return a new promise. And that's all a bit confusing and I really don't know why it was implemented like this, but this is just how it works. So anyway, what we need to do now here is to actually return this promise from here. Okay, because remember this here will be a new promise. Okay, and so now we need to handle that promise as well. All right, and so the way we do that is to then call another then right here. So we need another callback function,

```

const getCountryData = function (country) {
  fetch(`https://restcountries.com/v3.1/name/${country}`)
  .then(function (response) {
    console.log(response);
    return response.json(); // the json method here is a method that is available on all the
response objects that is coming from the fetch function, so all of the resolved values
  })
  .then(function (data) {
    console.log(data);
  });
};

getCountryData('portugal');

```

then we can create a highly simplified version as well:

```
const getCountryData = function (country) {  
  fetch(`https://restcountries.com/v3.1/name/${country}`)  
  .then((response)=> response.json())  
  .then((data) => renderCountry(data[0]));  
};  
getCountryData('portugal');
```

I hope that you agree that this here, is actually a lot nicer but beside being nicer, the code is also easier to read and to reason about and as I mentioned before, that in itself is very important.

\*\*\*you might be thinking, well, we're using callbacks here, right. And that is actually true. So promises do not get rid of callbacks, but they do in fact get rid of **callback hell**.

## ❖ 16. Asynchronous JavaScript Promises, AsyncAwait, and AJAX → 9. Chaining Promises

Actually when return a promise (line 9), we will be able to chain a new then method (line 11) base on previous returned promise. So that's a bit similar to what we did in line 4 (returning response.json() in arrow function), which remember also returned to promise.

```
1 const getCountryData = function (country) {  
2   //country 1  
3   fetch(`https://restcountries.com/v3.1/name/${country}`)  
4   .then(response => response.json())  
5   .then(data => {  
6     renderCountry(data[0]);  
7     const neighbour = data[0].borders[0];  
  
8     if (!neighbour) return;  
  
   //country2  
9     return fetch(`https://restcountries.com/v3.1/alpha/${neighbour}`);  
10  })  
11  .then(response => response.json())  
12  .then(data => renderCountry(data, 'neighbour'));  
13 };  
  
14 getCountryData('portugal');
```

So actually the then method always returns a promise, no matter if we actually return anything or not. But if we do return a value, then that value will become the fulfillment value of the return promise. In line 9 if we return 123, in next then we have 123 as fulfilled value.

So right now we have four steps here, even, but of course we could extend this as much as we want. So even if we wanted the neighbor of the neighbor of the neighbor, like 10 countries, we could easily do this by chaining all these promises one after another and all without the callback hell. So here, instead of the callback, hell we have what we call a flat chain of promises.

\*\*\*\*

I want to show you a pretty common mistake that many beginners make, which is to basically chain then method directly onto a new **nested promise**.

```
1 const getCountryData = function (country) {
  //country 1
2 fetch(`https://restcountries.com/v3.1/name/${country}`)
3 .then(response => response.json())
4 .then(data => {
5   renderCountry(data[0]);
6   const neighbour = data[0].borders[0];
7   if (!neighbour) return;
8
  //country2
9   fetch(`https://sample.com/alpha/${neighbour}`).then(response => response.json())
10  })
11 .then(response => response.json())
12 .then(data => renderCountry(data, 'neighbour'));
13 };
14
15 getCountryData('portugal');
```

So instead of returning the new promise, they then chain theme then method inside of previous promise (see line 9). Now this actually does still work, but then we are in fact back to **callback hell**.

## ❖ 16. Asynchronous JavaScript Promises, AsyncAwait, and AJAX → 10. Handling Rejected Promises

let's talk about how to handle errors in promises.

remember that a promise in which an error happens is a rejected promise. And so in this video, we're gonna learn how to handle promise rejections. Now, actually the only way in which the fetch promise rejects is when the user loses his internet connection. And so for now, that's gonna be the only error that we will handle here.

*\*\*\* to simulate losing the internet connection we can go to Network tab on chrome dev tools and then change the speed to Offline. However, when we then reload the page then basically everything will disappear*

We move `getCountryData('iran')` to a Btn click event and set the browser to offline, we now have an Uncaught promise, and so because we have failed to fetch.

And so at this point for the first time the promise that's returned from the fetch function was actually rejected. And so let's now handle that rejection. Now there are two ways of handling rejections and the first one is to pass a second callback function into the then method. So the first callback function here is always gonna be called for the fulfilled promise. so for a successful one. But we can also pass in a second callback which will be called when the promise was rejected.

So let's do that. And this callback function will be called with an argument which is basically the error itself. And so let's simply alert the error.

```
const getCountryData = function (country) {
  // country1
  fetch(`https://restcountries.com/v2/name/${country}`)
    .then(
      response => response.json(),
      err => alert(err)
    )
    .then(data => {
      renderCountry(data[0]);

      const neighbour = data[0].borders[0];
      console.log('border:', data[0].borders[0]);
      if (!neighbour) return;

      // country2
      return fetch(`https://restcountries.com/v2/alpha/${neighbour}`);
    })
    .then(response => response.json())
    .then(data => renderCountry(data, 'neighbour'));
};
```

now actually we handled the error by displaying the alert window.

And the error that we saw previously in console is now gone. So now, in fact, we no longer have this **Uncaught** error because we did actually **catch** the error.

So handling the error is also called catching the error.

But now what if there was actually no error in fetch 1 promise? So basically what if this fetch promise was actually fulfilled but then the 2 one was rejected. Well then we would also have to catch an error.

So we would have to come here get this function and paste it #3 and also handle the error right there:

```
const getCountryData = function (country) {  
  // country1  
  1 fetch(`https://restcountries.com/v2/name/${country}`)  
    .then(  
      response => response.json(),  
      err => alert(err)  
    )  
    .then(data => {  
      renderCountry(data[0]);  
  
      const neighbour = data[0].borders[0];  
      console.log('border:', data[0].borders[0]);  
      if (!neighbour) return;  
  
      // country2  
      2 return fetch(`https://restcountries.com/v2/alpha/${neighbour}`);  
    })  
    .then(  
      response => response.json(),  
      3 err => alert(err)  
    )  
    .then(data => renderCountry(data, 'neighbour'));  
};
```

However, that is a little bit annoying and so in fact there is a better way of basically handling all these errors globally just in one central place. So this is a lot nicer just having one callback in the then and then instead we can handle all the errors no matter where they appear in the chain right at the end of the chain by adding a catch method.

```
const getCountryData = function (country) {  
  // country1  
  fetch(`https://restcountries.com/v2/name/${country}`)  
    .then(response => response.json())
```

```

.then(data => {
  renderCountry(data[0]);

  const neighbour = data[0].borders[0];
  console.log('border:', data[0].borders[0]);
  if (!neighbour) return;

  // country2
  return fetch(`https://restcountries.com/v2/alpha/${neighbour}`);
})
.then(response => response.json())
.then(data => renderCountry(data, 'neighbour'))
→ .catch(err => alert(err));
};

```

When using the catch, errors basically propagate down the chain until they are caught, and only if they're not caught anywhere then we get that Uncaught error

usually simply logging the error to the console is not enough in a real application so instead of just logging something to the console let's also display an error message for the user to see.

```

.catch(err => {
  console.error(err);
  renderError(`Something went wrong ${err.message}. try again`);
});

```

actually this error (we call it err in this sample) that is generated is a real JavaScript object. So we can create errors in JavaScript with a constructor, for example, just like a map or a set.

And any error in JavaScript that was created like this contains the message property.

Now there is one more quick method that is also available on all promises.

So besides then and catch there is also the finally method.

Finally And its callback function that we defined

will always be called whatever happens with the promise.

So no matter if the promise is fulfilled or rejected this callback function is gonna be called always.

**So that's the difference:**

- **then** method is only called when the promise is fulfilled
- **catch** only called while the promise is rejected

- **finally** called when the promise is fulfilled or rejected

the finally method is not always useful, but sometimes it actually is. So we use this method for something that always needs to happen no matter the result of the promise. And one good example of that is to hide a loading spinner.

```
fetch(`https://restcountries.com/v2/name/${country}`)
  .then(response => response.json())
  .then(data => {
    renderCountry(data[0]);

    const neighbour = data[0].borders[0];
    console.log('border:', data[0].borders[0]);
    if (!neighbour) return;

    // country2
    return fetch(`https://restcountries.com/v2/alpha/${neighbour}`);
  })
  .then(response => response.json())
  .then(data => renderCountry(data, 'neighbour'))
  .catch(err => {
    console.error(err);
    renderError(`Something went wrong ${err.message}. try again`);
  })
→ .finally(() => {
  countriesContainer.style.opacity = 1;
});
```

\*\*\*

the fetch promise only rejects when there is no internet connection, but with a 404 error like this which is not a real error but well it kind of is. But anyway with this 404 the fetch promise will still get fulfilled. So there is no rejection and so our catch handler cannot pick up on this error.

## ❖ 16. Asynchronous JavaScript Promises, AsyncAwait, and AJAX → 11. Throwing Errors Manually



In this case we called the `getCountryData` with a wrong country name like this: `getCountryData('dfgdfgdgdg')`, there was a 404 error, which is because our API couldn't find any country with this name. But the promise not reject. So we should handle the error manually.

```
const getCountryData = function (country) {  
  // country1  
  fetch(`https://restcountries.com/v2/name/${country}`)  
    .then(response => {  
→ 5   if (!response.ok) throw new Error(`country not find ${response.status}`);  
  
    return response.json();  
  })  
  .then(data => {  
    renderCountry(data[0]);  
  
    const neighbour = data[0].borders[0];  
    console.log('border:', data[0].borders[0]);  
    if (!neighbour) return;  
  
    // country2  
    return fetch(`https://restcountries.com/v2/alpha/${neighbour}`);  
  })  
  .then(response => response.json())  
  .then(data => renderCountry(data, 'neighbour'))  
  .catch(err => {  
    console.error(err);  
    renderError(`Something went wrong ${err.message}. try again`);  
  })  
  .finally(() => {  
    countriesContainer.style.opacity = 1;  
  });  
};
```

So we create the new error by using again, this constructor function, basically, and then we pass in a message, which is gonna be the error message, then we use the `throw` keyword here, which will immediately terminate the current function. So just like `return` does it. Now the effect of creating, and throwing an error in any of these `then` methods is that the promise will immediately reject. So basically, the promise returned by this `then` handler here will be a rejected promise. And that rejection will then propagate all the way down to the `catch` handler, which we already have set up.

if we got no problem in this fires fetch ,but then we get a problem in the second (line 10).

```
const getCountryData = function (country) {  
  // country1  
  fetch(`https://restcountries.com/v2/name/${country}`)  
4  .then(response => {  
5    if (!response.ok) throw new Error(`country not find ${response.status}`);  
6    return response.json();  
7  })  
  .then(data => {  
    renderCountry(data[0]);  
  
    // const neighbour = data[0].borders[0];  
10   const neighbour = 'dfdfdfhdh';  
  
    if (!neighbour) return;  
  
    // country2  
    return fetch(`https://restcountries.com/v2/alpha/${neighbour}`);  
  })  
14  .then(response => {  
15    if (!response.ok) throw new Error(`country not find ${response.status}`);  
16    return response.json();  
  })  
  .then(data => renderCountry(data, 'neighbour'))  
  .catch(err => {  
    console.error(err);  
    renderError(`Something went wrong ${err.message}. try again`);  
  })  
  .finally(() => {  
    countriesContainer.style.opacity = 1;  
  });  
};  
  
btn.addEventListener('click', function () {  
  getCountryData('switzerland');  
});
```

So we now need to go ahead, and copy line 5 code to line 15, But now, of course, we have all this duplicate code in second then. And so now, I think that it's a good time to actually create ourselves a really nice helper function. And this helper function will wrap up the fetch the error handling, and also the conversion to JSON

```
const getJSON = function (url, errorMsg = 'Something went wrong') {
  return fetch(url).then(response => {
    if (!response.ok) throw new Error(`${errorMsg} ${response.status}`);

    return response.json();
  });
};

const getCountryData = function (country) {
  // country1
  getJSON(`https://restcountries.com/v2/name/${country}`, 'country not found')
  .then(data => {
    renderCountry(data[0]);
    const neighbour = data[0].borders[0];
    // if (!neighbour) return;
    if (!neighbour) throw new Error('No neighbour found'); // if a country have no
    neighbour, create an error

    // country2
    return getJSON(
      `https://restcountries.com/v2/alpha/${neighbour}`,
      'country not found'
    );
  })

  .then(data => renderCountry(data, 'neighbour'))
  .catch(err => {
    console.log(err);
    renderError(`Something went wrong ${err.message}. try again`);
  })
  .finally(() => {
    countriesContainer.style.opacity = 1;
  });
};

btn.addEventListeners('click', function () {
  getCountryData('australia');
});

getCountryData('iran');
```

- ❖ 16. Asynchronous JavaScript Promises, AsyncAwait, and AJAX → 13. Asynchronous Behind the Scenes The Event Loop

Just review the video

- ❖ 16. Asynchronous JavaScript Promises, AsyncAwait, and AJAX → 14. The Event Loop in Practice  
**Retake**
- ❖ 16. Asynchronous JavaScript Promises, AsyncAwait, and AJAX → 15. Building a Simple Promise

So at this point of the section, you know all about consuming promises but we have never actually built our own promise.

And let's go back to our lottery example from the slides and basically simulate a lottery using a promise here. And remember that in that example, a fulfilled promise means to win the lottery while a rejected promise means to lose.

Now the promise constructor takes exactly one argument and that is the so-called executor function. So we need to pass in a function called executor

Now this executor function that we specified here is the function which will contain the asynchronous behavior that we're trying to handle with the promise. So this executor function should eventually produce a result value. So the value that's basically gonna be the future value of the promise.

whatever value we pass into the resolve function here is gonna be the result of the promise that will be available in the then handler.

Then into the reject function, we pass in the error message that we later want to be able in the catch handler

```
const lotteryPromise = new Promise(function (resolve, reject) {  
  console.log('lottery draw is happening');  
  setTimeout(function () {  
    if (Math.random() >= 0.5) {  
      resolve('You Win');  
    } else {  
      reject('you lost your money');  
    }  
  });  
});
```

```
    }  
  }, 2000);  
});
```

And so now it's time to actually try this out by consuming this promise that we just built.

```
lotteryPromise.then(res => console.log(res)).catch(err => console.log(err));
```

is to, instead of passing just a string in reject, we can also create a new error object. So basically creating a real error,

```
reject(new Error('You lost yor money'));
```

Great, so this is how we encapsulate any asynchronous behavior into a promise. So how we abstracted away in a very nice way, just like we did here. And then all we have to do

is to consume that promise like this. And so this is a really nice and helpful pattern. Now, in practice, most of the time all we actually do is to consume promises. And we usually only built promises

to basically wrap old callback based functions into promises. And this is a process that we call promisifying. So basically promisifying means to convert callback based asynchronous behavior

to promise based. Let's see that in action a little bit. And so what we're gonna do is to actually promisify the set timeout function and create a wait function.

```
// Promisifying setTimeout  
// building  
const wait = function (seconds) {  
  return new Promise(function (resolve) {  
    setTimeout(resolve, seconds * 1000);  
  });  
};  
  
// consuming  
wait(1)  
  .then(() => {  
    console.log('1 second passed');
```

```

return wait(1);
})
.then(() => {
  console.log('2 second passed');
  return wait(1);
})
.then(() => {
  console.log('3 second passed');
  return wait(1);
})
.then(() => console.log('4 second passed'));

```

Now finally dare also actually a way to very easy create a fulfilled or a rejected promise immediately.

```

Promise.resolve('abc').then(x => console.log(x));
Promise.reject(new Error('Problem!')).then(x => console.error(x));

```

## ❖ 16. Asynchronous JavaScript Promises, AsyncAwait, and AJAX → 16. Promisifying the Geolocation API

Part 2 happens first and so that's because this function here basically offloaded its work to the background. So to the web API environment in the browser, and then immediately it moved on right part 2

```

1 navigator.geolocation.getCurrentPosition(
  position => console.log(position),
  err => console.log(err)
);
2 console.log('Getting position');

```

## ❖ 16. Asynchronous JavaScript Promises, AsyncAwait, and AJAX → 18. Consuming Promises with AsyncAwait

now that you're super comfortable with consuming promises and also building promises, let's turn our attention back to actually consuming promises that's because since ES 2017, there is now an even better and easier way to consume promises, which is called a sync await. So let me show you how it works.

So we start by creating a special kind of function,

which is an a sync function. a function that will basically keep running in the

background while performing the code that inside of it, then when this function is done, it automatically returns a promise, what's important is that inside an a sync function, we can have one or more await statements, so, await, and after that we need a promise.

we can use the await keyword to basically await for the result of this promise. So basically await will stop decode execution at this point of the function until the promise is fulfilled.

```
const whereAml = async function(country){
  await fetch(`https://restcountries.com/v2/name/${country}`)
}
```

you might think isn't stopping the code, blocking the execution? Well, that's a really good question, but the answer is actually no, in this case, because stopping execution in an a sync function, which is what we have here is actually not a problem because this function is running asynchronously in the background.

And so therefore it is not blocking the main thread of execution.

So it's not blocking the call stack. And in fact, that's, what's so special about a single wait. So it's the fact that it makes our code look like regular synchronous code while behind the scenes. Everything is in fact asynchronous.

\*\*

So we can simply await until the value of the promise is returned basically. And then just assign that value to a variable meant that is something that was impossible before.

So before we had to mess with callback functions and dead

was true in callback hell, but also by consuming promises with the then method. But now with a sync await, that is just completely gone, async await is in fact, simply syntactic sugar over the then method in promises.

So of course behind the scenes, we are still using promises.

```
const wherAml = async function (country) {
  // old way
  fetch(`https://restcountries.com/v2/name/${country}`).then(res => {
```

```

    console.log(res);
  });

  // new way
  const res = await fetch(`https://restcountries.com/v2/name/${country}`);
  console.log(res);
};

```

More completed example:

```

const wherAml = async function () {
  // Geolocation
  const pos = await getPosition();
  const { latitude: lat, longitude: lng } = pos.coords;

  // Reverse geocoding
  const resGeo = await fetch(`https://geocode.xyz/${lat},${lng}?geoit=json`);
  const dataGeo = await resGeo.json();
  console.log(dataGeo);

  // Country data
  // old way
  // fetch(`https://restcountries.com/v2/name/${country}`).then(res => {
  //   console.log(res);
  // });

  // new way to line 55 fetch
  const res = await fetch(`https://restcountries.com/v2/name/${dataGeo.country}`);
  const data = await res.json();
  console.log(data);
  renderCountry(data[0]);
};

```

## ❖ 16. Asynchronous JavaScript Promises, AsyncAwait, and AJAX → 19. Error Handling With try...catch

we're gonna learn how error handling works with async /await. So with async /await, we can't use the catch method that we use before, because we can really attach it anywhere, right. So instead, we use something called a try catch statement.

And the try catch statement is actually used in regular JavaScript as well.

So it's been in the language probably since the beginning. So try catch has nothing to do with async/await. But we can still use it to catch errors in async functions.

```
try {
```



```

let y = 1;
const x = 2;
x = 3;
} catch (err) {
  alert(err.message);
}

```

But anyway, this year is just a stupid syntax error. And of course, we're not going to use try catch to find mistakes that we make in our code. And so let's know use try catch for something more useful, which is to actually handle real errors in async functions.

```

const getPosition = function () {
  return new Promise(function (resolve, reject) {
    navigator.geolocation.getCurrentPosition(resolve, reject);
  });
};

const wherAmI = async function () {
  try {
    // Geolocation
    const pos = await getPosition();
    const { latitude: lat, longitude: lng } = pos.coords;

    // Reverse geocoding
    const resGeo = await fetch(`https://geocode.xyz/${lat},${lng}?geoit=json`);
    if (!resGeo.ok) throw new Error('Problem getting location data');

    const dataGeo = await resGeo.json();
    console.log(dataGeo);

    // Country data
    // old way
    // fetch(`https://restcountries.com/v2/name/${country}`).then(res => {
    //   console.log(res);
    // });

    // new way to line 55 fetch
    const res = await fetch(
      `https://restcountries.com/v2/name/${dataGeo.country}`
    );
    if (!res.ok) throw new Error('Problem getting country');

    const data = await res.json();

```

```

    console.log(data);
    renderCountry(data[0]);
  } catch (err) {
    console.log(err);
    renderError(`@@@ ${err.message}`);
  }
};

```

## ❖ 16. Asynchronous JavaScript Promises, AsyncAwait, and AJAX → 20. Returning Values from Async Functions

what an async function actually is and how it works?

We have this code:

```

const getPosition = function () {
  return new Promise(function (resolve, reject) {
    navigator.geolocation.getCurrentPosition(resolve, reject);
  });
};

const wherAml = async function () {
  try {
    // Geolocation
    const pos = await getPosition();
    const { latitude: lat, longitude: lng } = pos.coords;

    // Reverse geocoding
    const resGeo = await fetch(`https://geocode.xyz/${lat},${lng}?geoit=json`);
    if (!resGeo.ok) throw new Error('Problem getting location data');
    const dataGeo = await resGeo.json();

    // Country data
    const res = await fetch(`https://restcountries.com/v2/name/${dataGeo.country}`);
    if (!res.ok) throw new Error('Problem getting country');
    const data = await res.json();
    renderCountry(data[0]);

    return `You are in ${dataGeo.city}, ${dataGeo.country}`;
  } catch (err) {
    console.log(err);
    renderError(`@@@ ${err.message}`);
  }
};

```

Testing on this code:

```
console.log('1: will get location'); // logged first
const city = wherAml();
console.log(city); // logged second => Promise {<pending>}
console.log('2: Finish getting location');// logged last
```

an async function always returns a promise, we get “Promise {<pending>}” in console because JavaScript has simply no way of knowing yet what is the value of function because the function is still running.

Now how we can actually get the data that we want? We can use .then()

```
console.log('1: will get location'); // logged first
```

```
27 wherAml().then(city => console.log(city)); // logged last => You are in Chicago, United States of America (string returned from promise)
```

```
console.log('2: Finish getting location');// logged second
```

### now let's think about errors:

if any error occurs here in this try block, then this return here will never be reached because the code will immediately jump here to the catch block for example in line 13 if we have this:

```
13 const res = await fetch(`https://restcountries.com/v2/name/${dataGeo.countryfffff}`);
```

And so indeed now nothing was returned from the function, we get undefined.

Now what's interesting here is that the log line 27 still worked. This console.log which has now logging “undefined” is still running, which means that this callback function(city=>...) is still running, which means that the .then() method was called, which means that the promise was actually fulfilled and not rejected.

So even though there was an error in the async function, the promise that the async function returns is still fulfilled and not rejected.

Now, if we wanted to fix that, if we want to be able to catch that error here as well, then we would have to rethrow that error right here. Rethrowing the error means to basically throw the error again so that we can then propagate it down.

And so with that, we will manually reject a promise that's returned from the async function.

there's still a problem here. And that problem is the fact that doing this here kind of makes this the philosophy of async/await with handling promises using then and catch, right? So we are mixing the old and the new way of working with promises here, all in the same code. And that's something that I personally don't like. So I prefer to always just use async functions instead of having to mix them. And so let's now go ahead and convert this to async/await as well.

it would be great if we could simply use await without the async function, but that doesn't really work, because await can only be used inside an async function. and so instead we can use an IIFE. they are immediately-invoked function expressions.

```
(async function(){  
  
})();
```

Converting this:

```
wherAml()  
  .then(city => console.log(`2: ${city}`))  
  .catch(err => console.error(`2: ${err.message} @@`))  
  .finally(  
    () => console.log('3: Finish getting location') // logged second  
  );
```

To this:

```
(async function () {  
  try {  
    const city = await wherAml();  
    console.log(`2: ${city}`);  
  } catch(err) {  
    console.error(`2: ${err.message} @@`)  
  }  
  console.log('3: Finish getting location')  
})();
```

## ❖ 16. Asynchronous JavaScript Promises, AsyncAwait, and AJAX → 21. Running Promises in Parallel

```

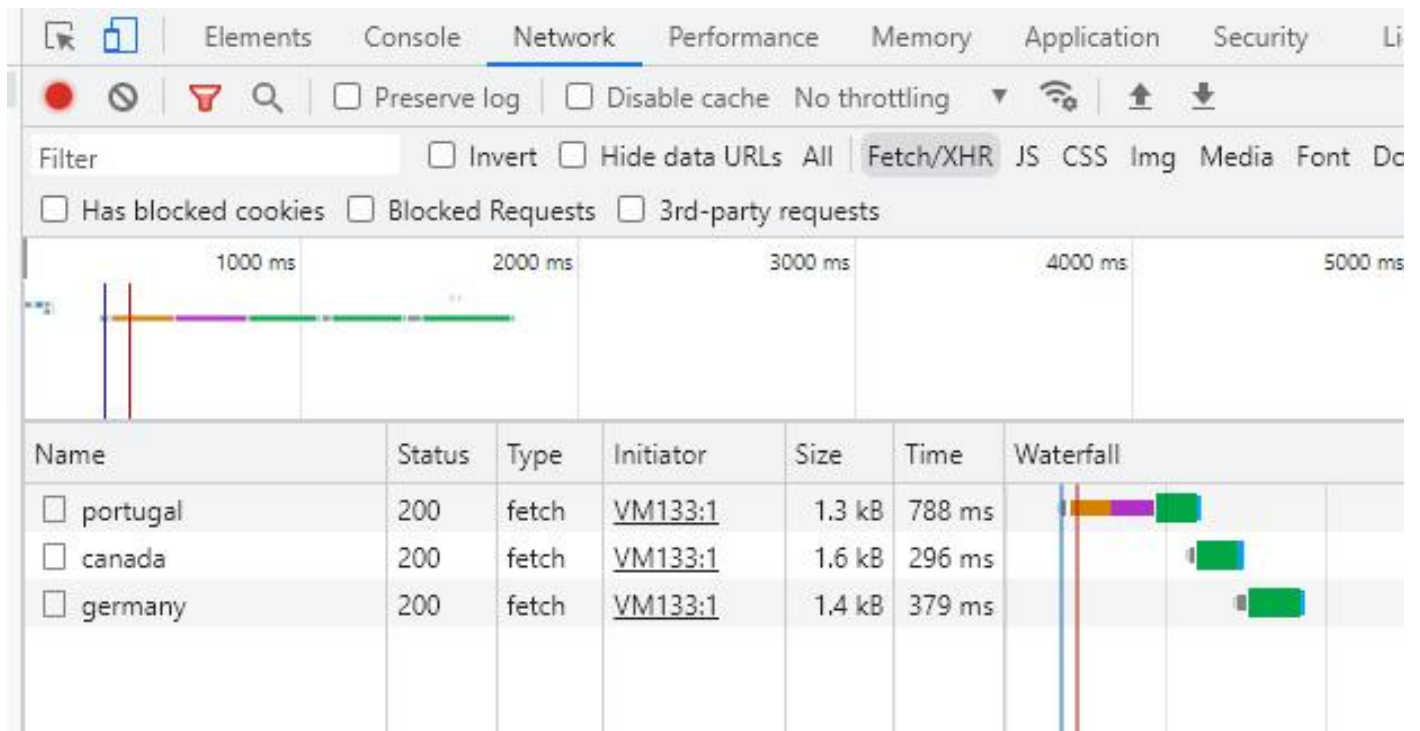
const get3countries = async function (c1, c2, c3) {
  try {
    const [data1] = await getJson(`https://restcountries.com/v2/name/${c1}`);
    const [data2] = await getJson(`https://restcountries.com/v2/name/${c2}`);
    const [data3] = await getJson(`https://restcountries.com/v2/name/${c3}`);

    console.log(data1.capital, data2.capital, data3.capital);
  } catch (err) {}
};

get3countries('portugal', 'canada', 'germany');

```

if we think about what we did here, then maybe it actually doesn't make so much sense because what we did here basically was to run all these Ajax calls one after another, even though the result of the second one here does not depend on the first one, and the result of the third one does also not depend on any of the other ones. And so actually this doesn't make much sense. Why should the second Ajax call wait for the first one?



So instead of running these promises in sequence, we can actually run them in parallel, so all at the same time. And so then we can save valuable loading time, making these three here, basically load at the same time. and for doing that, we use the promise.all combinator function, so promise.all(). this function here takes in an array of promises, and it will return a new promise,

\*\*\*\*

just one thing that's also very important to mention here is that if one of the promises rejects, then the whole promise.all actually rejects as well. So we say that promise.all short circuits when one promise rejects. So again, because one rejected promise is enough for the entire thing to reject as well.

## ❖ 16. Asynchronous JavaScript Promises, AsyncAwait, and AJAX → 22. Other Promise Combinators race, allSettled and any

We talked about promise.all() combinator.

Let's now quickly talk about the three other Promise combinators:

- **Promise.race()** :

And Promise.race, just like all other combinators, receives an array of promises and it also returns a promise. Now this promise returned by Promise.race is settled as soon as one of the input promises settles. And remember that settled simply means that a value is available, but it doesn't matter if the promise got rejected or fulfilled. And so in Promise.race, basically the first settled promise wins the race.

```
(async function () {  
  const res = await Promise.race([  
    getJson(`https://restcountries.com/v2/name/germany`),  
    getJson(`https://restcountries.com/v2/name/iran`),  
    getJson(`https://restcountries.com/v2/name/sweden`),  
  ]);  
  console.log(res[0]);  
})();
```

And so now these three promises will basically race against each other, like in a real race. Now, if the winning promise is then a fulfilled promise, then the fulfillment value of this whole race promise is gonna be the fulfillment value of the winning promise.

just keep in mind that here in Promise.race, we only get one result and not an array of the results of all the three. Now a promise that gets rejected can actually also win the race. And so we say that Promise.race short circuits whenever one of the promises gets settled.

And so again, that means no matter if fulfilled or rejected. in the real world Promise.race is actually very useful to prevent against never ending promises or also very long running promises. For example, if your user has a very bad internet connection, then a fetch requests in your application

might take way too long to actually be useful. And so we can create a special time out promise, which automatically rejects after a certain time has passed.

```
const timeout = function (sec) {
  return new Promise(function (_, reject) {
    setTimeout(function () {
      reject(new Error('Request took too long!'));
    }, sec * 1000);
  });
};

Promise.race([getJson(`https://restcountries.com/v2/name/mexico`), timeout(5)])
  .then(res => console.log(res[0]))
  .catch(err => console.error(err));
```

- **Promise.allSettled ()**

this one is a pretty new one. It is from ES2020 and it is actually a very simple one.

So it takes in an array of promises again,

and it will simply return an array of all the settled promises.

And so again, no matter if the promises got rejected or not. So it's similar to

Promise.all in regard that it also returns an array of all the results,

but the difference is that Promise.all will short circuit as soon as one promise rejects, but Promise.allSettled, simply never short circuits.

So it will simply return all the results of all the promises.

```
Promise.allSettled([
  ** Promise.resolve('success'),
  ** Promise.reject('Error'),
  ** Promise.resolve('Another success'),
]).then(res => console.log(res));
```

*// the output is:*

```
(3) [{...}, {...}, {...}]
0: {status: 'fulfilled', value: 'success'}
1: {status: 'rejected', reason: 'Error'}
2: {status: 'fulfilled', value: 'Another success'}
length: 3
[[Prototype]]: Array(0)
```

\*\* automatically creates a promise that is resolved/reject. so we don't have to wait for anything to finish

```

Promise.all([
  Promise.resolve('success'),
  Promise.reject('Error'),
  Promise.resolve('Another success'),
])
.then(res => console.log(res))
.catch(err => console.error(err));

```

```

// Output:
// Error

```

here we will simply get error and that's again... Because the .allpromise combinator will short circuit if there is one error, if there is one rejected promise. So that's the difference between these two.

- **Promise.any()**

So as always Promise.any takes in an array of multiple promises and this one will then return the first fulfilled promise and it will simply ignore rejected promises. So basically Promise.any is very similar to Promise.race with the difference that rejected promises are ignored. And so therefore the results of Promise.any is always gonna be a fulfilled promise, unless of course all of them reject, okay.

```

Promise.any([
  Promise.reject('Error'),
  Promise.resolve('success'),
  Promise.resolve('Another success'),
])
.then(res => console.log(res))
.catch(err => console.error(err));

```

```

// Output:
'Success'

```

## ❖ 16. Asynchronous JavaScript Promises, AsyncAwait, and AJAX → 23. Coding Challenge #3

**Just review the video** \_ practice the code



```

//Old code to convert:
let currentImage;
createImage('img/img-1.jpg')
  .then(img => {
    currentImage = img;
    console.log('image 1 loaded');
    return wait(2);
  })
  .then(() => {
    currentImage.style.display =
'none';
    return createImage('img/img-
2.jpg');
  })
  .then(() => {
    currentImage = img;
    console.log('image 2 loaded');
    return wait(2);
  })
  .then(() => {
    currentImage.style.display =
'none';
  })
  .catch(err => console.log(err));

```

```

// converted
// PART1
const loadNPause = async function () {
  try {
    // Load image 1
    let img = await
createImage('img/img-1.jpg');
    console.log('image 1 loaded');
    await wait(2);
    img.style.display = 'none';

    // Load image 2
    img = await createImage('img/img-
1.jpg');
    console.log('image 2 loaded');
    await wait(2);
    img.style.display = 'none';
  } catch (err) {
    console.error(err);
  }
};
loadNPause();

```

```

// PART 2
const loadAll = async function
(imgArr) {
  try {
    const imgs = imgArr.map(async img
=> await createImage(img));
    console.log(imgs);

    const imgsEl = await
Promise.all(imgs);
    console.log(imgsEl);
    imgsEl.forEach(img =>
img.classList.add('parallel'));
  } catch (err) {
    console.error(err);
  }
};

loadAll(['img/img-1.jpg', 'img/img-
2.jpg', 'img/img-3.jpg']);

```

## ❖ 17. Modern JavaScript Development Modules, Tooling, and Functional → 3. An Overview of Modern JavaScript Development

Let's start a section with a general overview of Modern JavaScript Development. So basically of how we write JavaScript today. Because when we built applications, we don't just write all of our code into one big script send that script as it is to the browser, and call it a day. Now it used to be like this, but the way we built JavaScript applications, has changed tremendously over the last couple of years. So back in the day, we used to write all our codes into one big script or maybe multiple scripts. But today, we divide our projects into multiple modules and these modules can share data between them

and make our code more organized and maintainable. Now, one great thing about modules is that we can also include 3rd-party modules into our own code. And there are thousands of open source modules, which we also call packages, that developers share on the NPM repository. And we can then use these packages for free in our own code. For example, the popular React framework or jQuery, or even the Leaflet library, that we used before in our Mapty project. All of these packages are available through NPM. Now NPM stands for Node Package Manager, because it was originally developed together with Node.js and 4Node.js. However, NPM has established itself as the go to repository for all kinds of packages in Modern JavaScript Development. Now, in order to actually download

and use and share packages, we use the NPM software installed on our computer. And this is just a simple command line interface that allows us to do all that. So basically NPM is both the repository in which packages live and a program that we use on our computers to install and manage these packages. So let's say that we are done writing our project code. So we divided it into multiple modules and we included some 3rd-party modules as well. And so now the development step is complete. However, usually that's not the end of the story. At least not when rebuilding a real world application. Instead, our project now needs to go through a build process, where one big final JavaScript bundle is built. And that's the final file, which we will deploy to our web server for production.

So basically it's the JavaScript file, that will be sent to browsers in production. And production simply means that the application is being used by real users in the real world. Now, a build process can be something really complex, but we gonna keep it simple here and only include two steps.

And the first step, we'll bundle all our modules together into one big file. This is a pretty complex process which can eliminate unused code and compress or code as well. Now this step is super important for two big reasons. First, older browsers don't support modules at all. And so code that's in a module could not be executed by any older browser. And second, it's also better for performance to send less files to the browser, and it's also beneficial that the bundling step compresses our code.

But anyway, as the second step, we do something called transpiling and polyfilling, which is basically to convert all modern JavaScript syntax and features back to old ES5 syntax, so that even older browsers can understand our code without breaking. And this is usually done using a tool called Babel. So, remember that I said right in the beginning of the course, that we were gonna do this,

and in this section, we will finally do it.

So these are the two steps of our build process, and after these two steps, we end up with that final JavaScript bundle, ready to be deployed on a server for production. Now, of course we don't perform these steps ourselves. Instead, we use a special tool to implement this build process for us. And the most common build tools available, are probably webpack and Parcel.

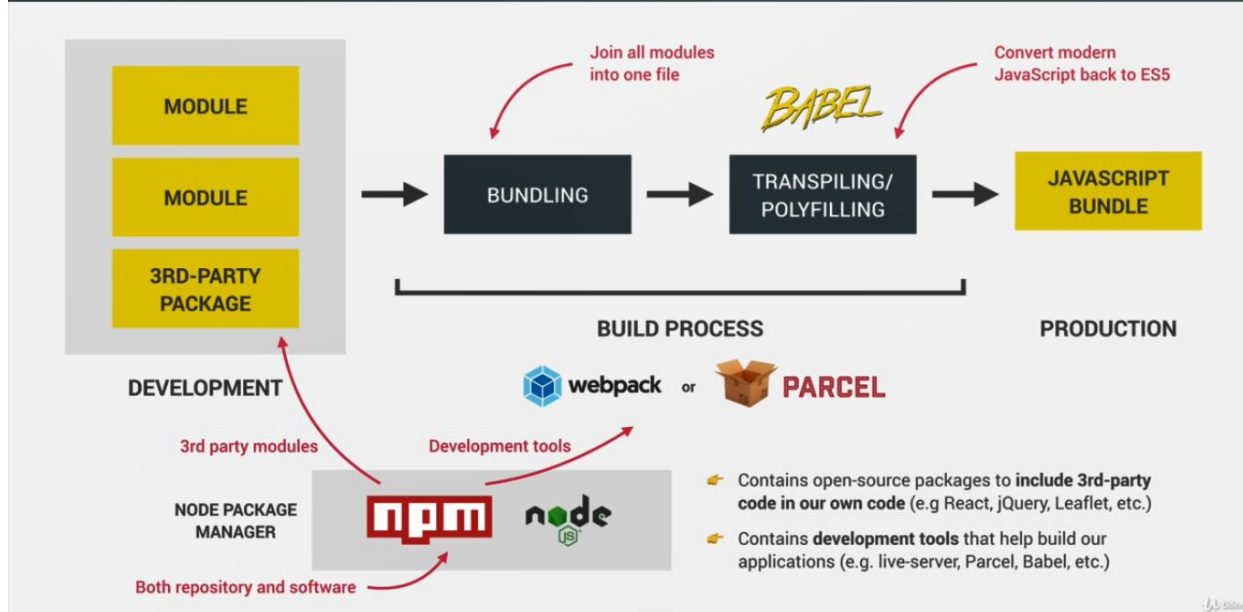
And these are called JavaScript bundlers because well, as the name says they take our raw code and transform it into a JavaScript bundle. Now Webpack is the more popular one, but it can be really hard and confusing to set it up. So that's because there's a lot of stuff that we need to configure manually, in order to make it work properly. Parcel, on the other hand is a zero configuration bundler,

which simply works out of the box. And so in this bundler, we don't have to write any set up code,

which is really amazing. So I personally absolutely love Parcel and I use it all the time, and I think that you will love it too. And so, that is the JavaScript bundler that we gonna use later in the section. Now these development tools are actually also available on NPM. So just like packages that we include in our code, we will download and manage tools using NPM as well. And these tools include the live-server

that we've been using all along, the Parcel bundler that we just talked about or Babel to transpile code back to ES5. All right, so this is a high level overview, of how we develop Modern JavaScript applications today. And if you ask me, this is really exciting stuff, because this is how professional developers actually write JavaScript today. And so in the rest of the section, you will take a big step in the direction of becoming a professional developer too.

## MODERN JAVASCRIPT DEVELOPMENT



### ❖ 17. Modern JavaScript Development Modules, Tooling, and Functional → 4. An Overview of Modules in JavaScript

In this lecture, we're gonna talk about modules:

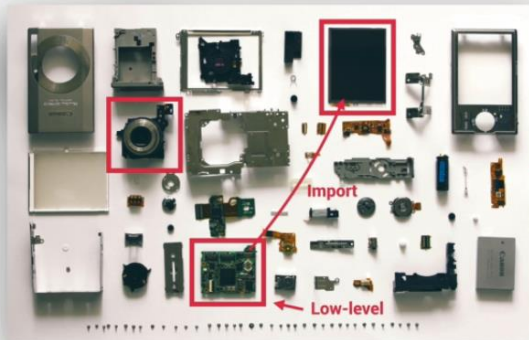
essentially, a module is a reusable piece of code that encapsulates implementation details of a certain part of our project. Now that sounds a bit like a function or even a class, but the difference is that a module is usually a standalone file. Now that's not always the case, but normally when we think of a module

we think of a separate file. So of course a module always contains some code but it can also have imports and exports. So with exports, as the name says, we can export values out of a module for example, simple values or even entire functions. And whatever we export from a module is called the public API. So this is just like classes where we can also expose a public API for other codes to consume. Now, in the case of modules, this public API is actually consumed by importing values into a module.

So just like we can export values in modules, we can usually also import values from other modules. And these other modules from which we import are then called dependencies of the importing module because the code dead is in the module dead is importing cannot work without the code, that it is importing from the external module, right?

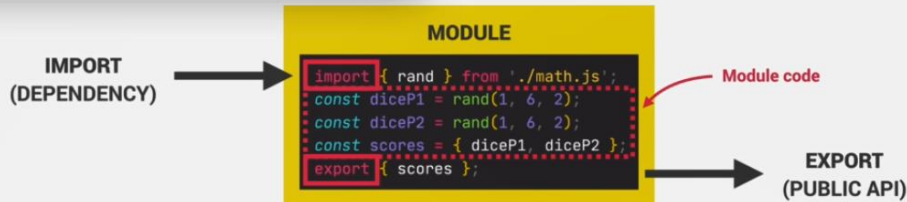
And this entire logic that I just described is true for all modules in all programming languages. So this is not specific to only JavaScript.

## AN OVERVIEW OF MODULES



WHY  
MODULES?

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



JavaScript has a native built-in module system. Now we did have modules before ES6, but we had to implement them ourselves or use external libraries. So ES6 modules are modules that are actually stored in files and each file is one module. So there is exactly one module per file. But now you might be thinking, well, scripts are usually also files, right? And that's of course true. And so let's not compare these two types of files in order to understand that there are actually huge differences between old school scripts and modern ES6 modules. The first difference is that in modules, all top level variables

are scoped to the module. So basically variables are private to the module by default. And the only way an outside module can access a value that's inside of a module is by exporting that value. So just as we learned in the last slide. But if we don't export, then no one from the outside can see the variable. Now in scripts, on the other hand, all top level variables are always global and I showed you this in the map d project, remember? And this can lead to problems like global namespace pollution, where multiple scripts try to declare variables with the same name and then

these variables collide. So private variables are the solution to this problem. And that's why ES6 modules implemented it like this. Next ES modules are always executed

in strict mode while scripts on the other hand are executed in sloppy mode by default. So with modules, there is no more need to manually declare strict mode. Also the `disc` keyword is always undefined at the top level while in scripts it points at the window object, right? Now, as we learned in the last slide, what's really special about modules is that we

can export and import values between them using this ES6 import and exports syntax. In regular scripts, importing and exporting values is just completely impossible. Now, there is something really important to note about imports and exports, which is the fact that they can only happen at the top level. So as you know, outside of any function or any if block, and we will see why that is in a second.

Also all imports are hoisted. So no matter where in a code you're importing values, it's like the import statement will be moved to the top of the file. So in practice importing values is always the first thing that happens in a module. Now, in order to link a module to an HTML file, we need to use the script tag with the type attribute set to module, instead of just a plain script tag. And finally about downloading the module files themselves. This always automatically happens in an asynchronous way. And this is true for a module loaded from HTML as well as for modules

that are loaded by importing one module into another, using the import syntax. Now regular scripts on the other hand are downloaded by default in a blocking synchronous way, unless we use the async or defer attributes on the script tag. So that's a great overview of ES6 modules,

## NATIVE JAVASCRIPT (ES6) MODULES

**ES6 MODULES**

Modules stored in files, **exactly one module per file.**

```
import { rand } from './math.js';
const diceP1 = rand(1, 6, 2);
const diceP2 = rand(1, 6, 2);
const scores = { diceP1, diceP2 };
export { scores };
```

**import and export syntax**

**Need to happen at top-level Imports are hoisted!**

	ES6 MODULE	SCRIPT
👉 Top-level variables	Scoped to module	Global
👉 Default mode	Strict mode	"Sloppy" mode
👉 Top-level this	undefined	window
👉 Imports and exports	✅ YES	❌ NO
👉 HTML linking	<script type="module">	<script>
👉 File downloading	Asynchronous	Synchronous

but now let's dig a bit deeper and really understand how modules actually import other modules behind the scenes.

And to do that, let's analyze what happens in this small code example. So here we're importing a value

called `rent` from the `math.js` module and `show` dies from the `dumb.js` module. Now, as always, when a piece of code is executed, the first step is to parse that code. Remember, so we talked about that way back.

But remember that parsing basically means to just read the code, but without executing it. And this is the moment in which imports are hoisted. And in fact, the whole process of importing modules happens before the code in the main module is actually executed. So in this example, the `index.js` module imports, the `dumb` and `math` modules in a synchronous way. What that means is that only after all imported modules have been downloaded and executed, the main `index.js` module will finally be executed as well. Now this is only possible because of top level imports and exports that's because if we only export and import values outside of any code that needs to be executed,

then the engine can know all the imports and exports during the parsing phase. So while the code is still being read

before being executed. Now, if we were allowed to import a module

inside of a function, then that function would first have to be executed before the import code happened. And so in that case, modules could not be imported in a synchronous way. So the importing module

would have to be executed first. But you might ask why do we actually want modules to be loaded in a synchronous way? Isn't synchronous bad? Well, the answer is that this is the easiest way in which we can do things

like bundling and dead code elimination. So basically deleting code that's actually not even necessary. And trust me, this is very important in large projects with hundreds of modules and that includes third party modules

from which we usually only want a small piece and not the entire module. So by knowing all dependencies between modules before execution, bundlers like `webpack` and `Parcel` can then join multiple modules together and eliminate that code.

And so essentially this is the reason why we can only import and export outside of any code that needs to be executed. So like a function or an `if` block,

## HOW ES6 MODULES ARE IMPORTED

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

← index.js

Parsing index.js

IMPORTING MODULES  
BEFORE EXECUTION

- Modules are **imported synchronously**
- Possible thanks to top-level ("static") imports, which make imports **known before execution**
- This makes **bundling** and **dead code elimination** possible

Execution index.js

but now let's move on here. So after the parsing process, HIAS figured out which modules it needs to import, then these modules are actually downloaded

from the server. And remember downloading actually happens

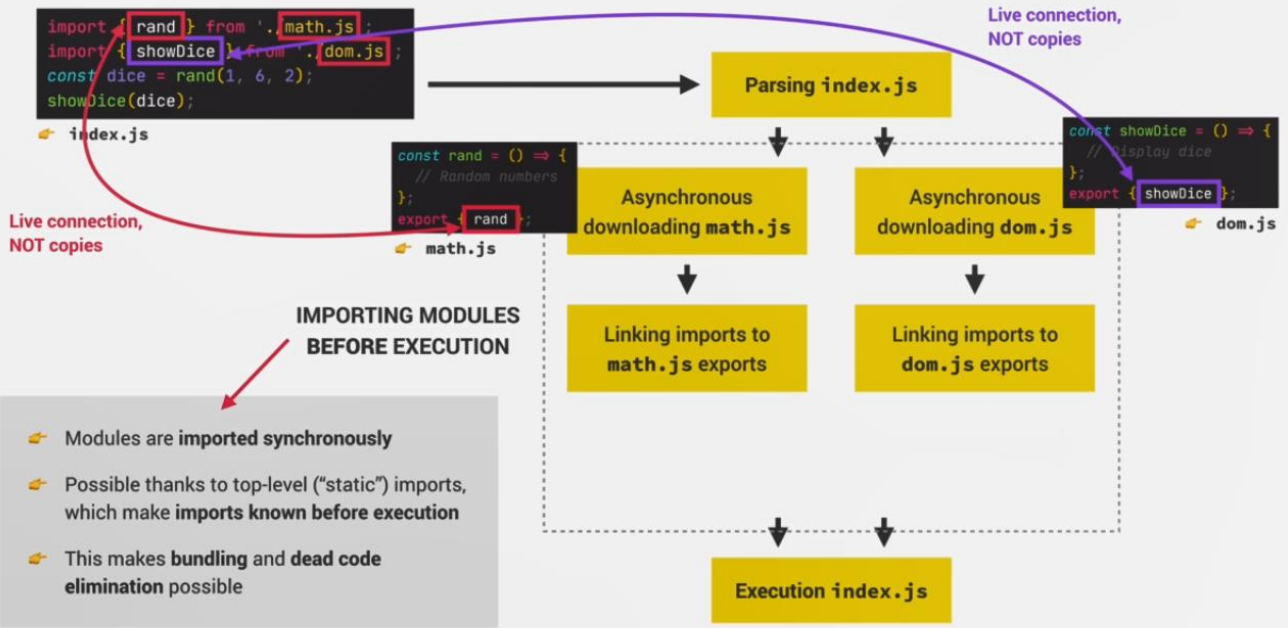
in an asynchronous way. It is only the importing operation itself that happens synchronously. Then after a module arrives, it's also parsed and the module's exports are linked to the imports in index.js. So for example, the math module exports a function called `rand`.

And this export is then connected to the `rand` import in the index.js module. And this connection is actually a live connection. So exported values are not copied to imports. Instead, the import is basically just a reference to the export at value

like a pointer. So when the value changes in the exporting module, then the same value also changes in the importing module. And this is quite important to understand because it's unique to ES6 modules. Other module systems do not work like this, but JavaScript modules do. And so you need to keep that in mind.

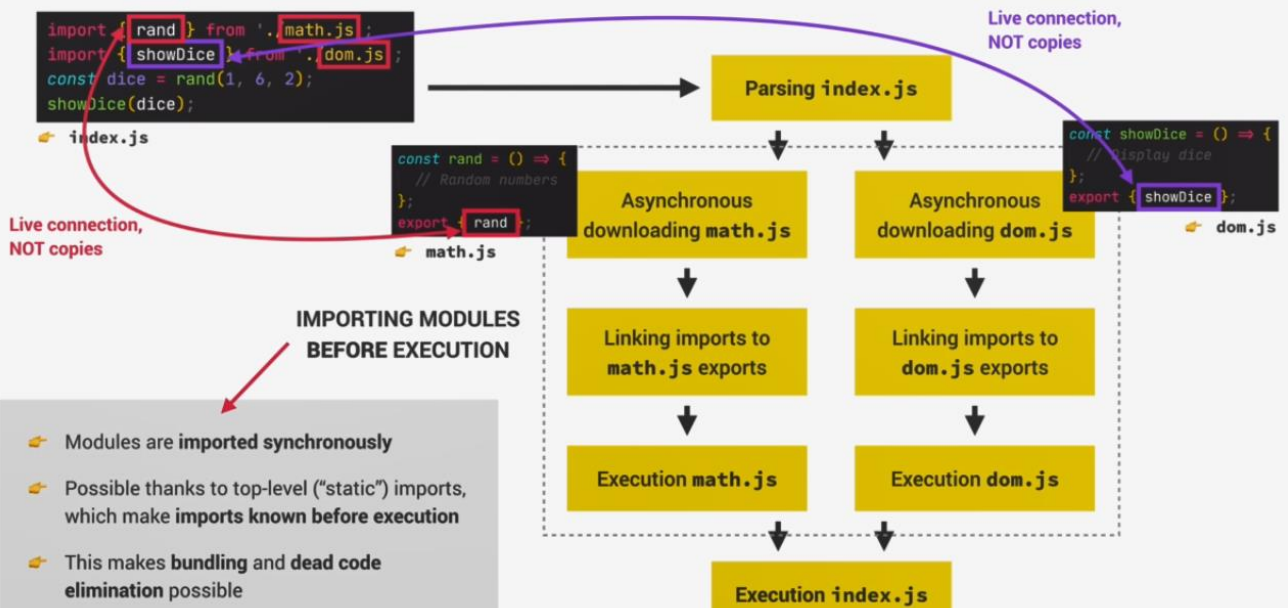


# HOW ES6 MODULES ARE IMPORTED



But anyway, next up, to code in the imported modules is executed. And with this the process of importing modules is finally finished. And so now, as I already said, it's time for the importing module to be finally executed as well. So index.js in this example.

# HOW ES6 MODULES ARE IMPORTED



## ❖ 17. Modern JavaScript Development Modules, Tooling, and Functional → 5. Exporting and Importing in ES6 Modules

\*\*\* When we want to connect a module to the HTML file, we actually need to specify the type attribute.

```
<script type="module" src="script.js"></script>
```

\*\*\* Exporting module code is executed **before** any code in the importing module.

\*\*\* All the importing statements are basically hoisted to the top. And usually we write it imports statements at the top of the file

\*\*\* And also we didn't use strict mode , and that's because all modules are executed in strict mode by default.

\*\*\* Variables that are declared inside of a module, So just like these two ones here, are actually scoped to this module. So basically inside a module, the module itself is like the top level scope. And so by default, this means that all top level variables are private inside of this variable.

If we wanted to use shCart.js variables in the script.js module then we would have to use **exports**.

Types of ES modules exports:

- Named Exports
- Default Exports

### Named Exports:

\*\*\* Use exact same name in importing module as variable name in exporting module

\*\*\* keep in mind that exports always need to happen in top level cart

```
// Exporting module (shCart.js)
console.log('Exporting module');

export const addToCart = function
(product, quantity) {
  cart.push({ product, quantity });
  console.log(`${quantity} ${product}
added to cart`);
};
```

```
// Importing module
import { addToCart } from './shCart.js';

console.log('Importing module');
addToCart('bread', 5);
```

we can also export multiple things from a module using Named Exports. And actually, that is the main use case of Named Exports

```
// Exporting module (shCart.js)
const shippingCost = 10;
const cart = [];

export const addToCart = function (product, quantity) {
  cart.push({ product, quantity });
  console.log(`${quantity} ${product} added to cart`);
};

const totalPrice = 237, totalQuantity = 23;

➔ export { totalPrice, totalQuantity };
```

```
// Importing module
➔ import { addToCart, totalPrice, totalQuantity } from './js5_shoppingCart.js';

addToCart('bread', 5);

console.log(totalPrice, totalQuantity);
```

Now actually we can change the name of the inputs as well, we can do that in the importing module (like price), or do in exporting module (like tq):

```
// Exporting module (shCart.js)
const shippingCost = 10;
const cart = [];

export const addToCart = function (product, quantity) {
  cart.push({ product, quantity });
  console.log(`${quantity} ${product} added to cart`);
};

const totalPrice = 237, totalQuantity = 23;

➔ export { totalPrice, totalQuantity as tq };
```

```
// Importing module
➔ import { addToCart, totalPrice as price, tq } from './js5_shoppingCart.js';

addToCart('bread', 5);

➔ console.log(price, tq);
```

we can also import all the exports of a module at the same time.

Use capital first letter a little bit like a class name. So that's kind of the convention when we import everything into an object like this. So basically, this here we'll create an object containing everything that is exported from the module that we will specify here.

\*\*\*this will then basically create a namespace, for all of the values, exported from that module.

```
// Importing module
import * as ShoppingCart from './shoppingCart.js';
ShoppingCart.addToCart('bread', 5);
console.log(ShoppingCart.totalPrice);
```

shoppingCart.js is now basically exporting a public API, just like a class. So it's as if this object here, was an object created from a class, which now has these methods (like addToCart()), and also, for example totalPrice properties.

Now of course we are not trying to replace classes with modules. I just wanted to turn your attention to the fact, that some things here look pretty similar indeed. And actually we can say that module exports kind of a public API, because everything else of course stays private inside of the module.

### Default Exports:

So usually, we use Default Exports when we only want to export one thing per module, and so that's the reason why they are called default.

for example if we wanted to export the same function, we would simply export the value itself, so not the variable. we want to export a value so no name is involved at all.

And so then when we import it we can basically give it any name that we want.

```
// Exporting module
export default function (product, quantity) {
  cart.push({ product, quantity });
  console.log(` ${quantity} ${product} added to cart`);
}
```

```
// Importing module
import add from './js5_shoppingCart'
add('pizza',2)
```

if we wanted, we could have default and named imports and exports all at the same time.

\*\*\*However in practice, we usually never mix Named and Default Exports in the same module.

```
import add, { addToCart, totalPrice as price, tq } from './js5_shoppingCart'
```

```
add('pizza', 2)
console.log(price);
```

\*\*\*Imports are not copies of the export. They are instead like a live connection, and so what that means is that I point to the same place in memory, this means if we export `const arr = []` and manipulate it later, we have update value(not empty) when we use it in importing module.

## ❖ 17. Modern JavaScript Development Modules, Tooling, and Functional → 6. The Module Pattern

I just wanna quickly show you the module pattern that we used to use before in order to implement modules in JavaScript. And I believe that it's important that you understand this module pattern because you will still see it around, ow, of course, just like in regular modules that we just learned about, the main goal of the module pattern is to encapsulate functionality, to have private data, and to expose a public API. And the best way of achieving all that is by simply using a function, because functions give us private data by default and allow us to return values, which can become our public API. is only created once because the goal of this function is not to reuse code by running it multiple times, the only purpose of this function is to create a new scope and return data just once.

```
(function () {
...
})();
```

That's the implementation of the module pattern. Now, do you understand exactly how and why this works? I mean, how do we, for example, have access to the cart variable here and even are able to manipulate it, so we see that it at changed, indeed. So how are we able to do that, even if this IIFE here, so this function has already returned long ago, right?

So this function, of course, was only executed once in the beginning, and then all it did was to return this object and assigned it to this variable, right? But then we are able to use all of this and to also manipulate the data that is inside of this function, which is the function that returned this object. And the answer to how all of this works like this is one more time, closures. So closures, remember, allow a function to have access to all the variables that were present at its birthplace this is how the module pattern works and it works very well, and it has been working for a long time for developers, so long before ES6 modules even existed in JavaScript. Now, the problem is that if we wanted one module per file, like we have with ES6 modules, then we would have to create different scripts

and link all of them in the HTML file. And that then creates a couple of problems, like we have to be careful with the order in which we declare them in HTML, and we would have all of the variables living in the global scope, and finally, we also couldn't bundle them together using a module bundler. And so as you learned at the beginning of this section, using a module bundler is very important in modern JavaScript. So the module pattern that we just learned about does indeed work quite good, but it has some limitations. And so that's exactly the reason why native modules were added to the language in ES6.

```
const ShoppingCart2 = (function () {
  const cart = [];
  const shippingCost = 10;
  const totalPrice = 237;
  const totalquantity = 23;
  const addToCart = function (product, quantity) {
    cart.push({ product, quantity });
    console.log(`${quantity} ${product} added to cart`);
  };

  const orderStock = function (product, quantity) {
    console.log(`${quantity} ${product} ordered from supplier`);
  };

  return {
    addToCart,
    cart,
    totalPrice,
    quantity,
  };
})();

ShoppingCart2.addToCart('apples', 4);
ShoppingCart2.addToCart('pizza', 2);
console.log(ShoppingCart2);
console.log(ShoppingCart2.shippingCost); // undefined (its private to function)
```

## ❖ 17. Modern JavaScript Development Modules, Tooling, and Functional → 7. CommonJS Modules

Besides native ES Modules and the module pattern, there are also other module systems, that have been used by JavaScript in the past. But again, they were not native JavaScript.

so they relied on some external implementations. And two examples are: AMD Modules, and CommonJS modules. And in fact, CommonJS modules, are worth taking a look at. And so let's do that now. Now CommonJS modules are important for us, because they have

been used in Node.js, for almost all of its existence. So only very recently, ES Modules have actually been implemented, in Node.js. And remember, Node.js is a way of running JavaScript on a web server, outside of a browser. Now the big consequence of this, is that almost all the modules, in the npm repository, that we talked about in the beginning of this section, remember? So all these modules that we can use in our own code, still use the CommonJS module system. And the reason for that, is that npm was originally only intended for node. Which as they said, uses commonJS. Only later npm became the standard repository, for the whole JavaScript world. And so now we are basically stuck, with CommonJS. And so therefore, you will see probably, a lot of CommonJS still around. And so let's take a quick second to see what it looks like.

And so just like ES6 modules, in CommonJS, one file, is one module. And we export something from a module, using **export**.

\*\*\*its not work in browser.but going to work in browser.

```
// Export
export.addToCart = function (product, quantity) {
  cart.push({ product, quantity });
  console.log(` ${quantity} ${product} added to cart`);
};

// Import
const {addToCart} = require('./shippingCart.js');
```

❖ **17. Modern JavaScript Development Modules, Tooling, and Functional → 8. A Brief Introduction to the Command Line**

<b>Ls</b>	<b>Show files and folders</b>
<b>Cd \</b>	<b>Go to root</b>
<b>Cd ..</b>	<b>Go to parent dir</b>
<b>Cd ../../</b>	<b>Go 2 level to parent dir</b>
<b>clear</b>	<b>Clear the screen</b>
<b>mkdir</b>	<b>Making directory</b>
<b>Edit filename.ex</b> <b>echo &gt; some-text filename.ex</b>	<b>Create a new file</b>
<b>Del filename.ex</b>	<b>Delete a file</b>
<b>Mv filename.ex ../</b>	<b>Move a file to parent dir</b>

## ❖ 17. Modern JavaScript Development Modules, Tooling, and Functional → 9. Introduction to NPM

NPM stands for Node Package Manager, and it's both a software on our computer and a package repository.

why we actually need something like NPM. So, why do we actually need a way of managing packages or dependencies in our project? Well, back in the day before we had NPM,

we used to include external libraries right into our HTML. So, basically using the script tag. And this would then expose a global variable that we could use and actually that's exactly what we did earlier in our Mapty project. (leaflet by a script tag in html file)

this actually creates a couple of problems, at least in a big project so maybe not in this particular project that is really small, but in a huge project and a huge team, this is just not manageable. First, it doesn't make much sense having the HTML loading all of JavaScript, that is just really messy. Also many times we would actually download a library file to our computer directly, for example, a jQuery JavaScript file. But then whenever a new version would come out, we would have to manually go to the site,

download the new version, change the file in our file system manually,

and then include it here again, maybe with some other name, with some other version number. And that was just a huge pain, believe me. And a third reason is that before NPM, there simply wasn't a single repository that contained all the packages that we might need.

And so this made it even worse and more difficult to manually download libraries and manage them on our computers. So in summary, this all used to be a huge pain and a huge mess. And maybe you even remember this yourself, like the old days of jQuery and dozens of jQuery plugins that you would have to keep updated. But anyway, all of this is just to say that we really need a way to manage our dependencies in a better and more modern way.

And NPM is exactly how we do that. And so, let's start by using the NPM software now.

**1:**

```
npm -v
```

 show you installed npm or no

So, all that matters is that you install Node.js and then as you run this command, you just need to get some number here, and then you're good to go.

**2:**

Now in each project in which we want to use NPM, we need to start by initializing it.

```
npm init
```



And so for that, we write NPM in it, and this will then ask us a couple of questions in order to create a package.json file. So, the questions are down here. So, first the package name, and if we just had enter, then whatever is here in these parentheses will be the default. And then we end up with a special file called package.json. Now, this file here is basically what stores the entire configuration of our project.

3:

And now let's actually install the leaflet library that we used before, but this time using NPM.

```
npm install leaflet (we can use npm l leaflet)
```

and then it will start downloading and installing. And now you'll see that two things happened. So first of all, in our package.json file, a new field here was created for the dependencies. And the dependency that we have here now is leaflet that we just installed.

Okay, so we installed our leaflet library now, but if we wanted to use it, that wouldn't be easy without a module bundler. And that's because this library actually uses

the common JS module system. So for the reasons that I explained to you before, and so therefore we cannot directly import it into our code. We could only do that if later we used a module bundler, but for now we are not doing that. And so, let's just not use leaflet for now. So, this was just to show you how to install it.

So instead, let me show you how we can install and import one of the most popular JavaScript libraries, which is Lodash.

do you remember we use `obj2 = Object.assign({},obj1)` for create a clone of a object, but if change one of nested object in obj1, now its changed in obj2.

```
const state = {
  cart: [
    { product: 'bread', quantity: 5 },
    { product: 'pizza', quantity: 5 },
  ],
  user: { loggedIn: true },
};

const stateClone = Object.assign({}, state);
stateClone.user.loggedIn = false;
console.log(stateClone.user.loggedIn); //false
```

For resolving this problem using `Lodash` would probably be a good idea, instead of using `Object.assign`, because if we wanted to manually create a deep copy or a deep clone, well, that would be a lot of work. So `Lodash`, like `underscore` and `Lodash` is essentially a collection of a ton of useful functions for arrays, objects, functions, dates, and more. So, it's a lot of like functions that could or should be included in JavaScript, but are not. And so people simply implemented them in `Lodash`, and so now we can use them.

\*\*\* we can't use common JS modules without a module bundler.\*\*\*

And so I'm looking for a special version is called ***Lodash-es***. (ES is because of ES modules)

```
npm i lodash-es
```

now if we take look at `Lodash` directory on `node_modules` see now basically we have one file for each of the methods that are available in `Lodash`. And the one that I want to include now is the one for cloning objects. So, that's called `cloneDeep.js`.

```
import cloneDeep from './node_modules/lodash-es/cloneDeep.js';

const state = {
  cart: [
    { product: 'bread', quantity: 5 },
    { product: 'pizza', quantity: 5 },
  ],
  user: { loggedIn: true },
};

const stateDeepClone = cloneDeep(state); //using Lodash to deepClone

state.user.loggedIn = false; // just user.loggedIn of "state" will be false

console.log(stateDeepClone.user.loggedIn); //true
```

\*\*\*

let's say that you want to move your project to another computer, or also share it with another developer or even check it into version control like `Git`. Now in all of these scenarios, you should never ever include the `node_modules` folder.

So again, when you copy your project to somewhere else, there is no reason to include this huge `node_modules` folder, because in a real project, it will actually be really, really huge. So, I have had a folders here with tens of thousands of files, and so that will just slow you down. And it doesn't make much sense either because all of these files, they are already at `NPM`. And so, you can always get them back from there. But now you might ask, well, if I copy my project without the `node_modules` folder, so without the dependencies, will I have

to install all of them, one by one? What if I have 20 or 200 dependencies? Well, that's again where this important package.json file comes into play. So, let's actually simulate that by deleting this folder. So, move to trash.

And so now of course this does not work anymore, but there is fortunately a very easy way to get it back. All we have to do is NPM and then install or I, but just without any package name. And so then, NPM will reach into your package.json file, look at all the dependencies and install them back.

importing packages like we did here, for example:

```
import cloneDeep from '../node_modules/lodash-es/cloneDeep.js';
```

by specifying this entire path is not practical at all. And so in the session, it's time to finally use **Parcel** to fix this.

## ❖ 17. Modern JavaScript Development Modules, Tooling, and Functional → 10. Bundling With Parcel and NPM Scripts

So the module bundler that we're gonna use in this course is called Parcel. And it's super fast and easy to use, and, even more importantly, it works out of the box without any configuration. Now you might've heard of Webpack as well which is probably the most popular bundler and especially in react world. However, it's way too complex to use in a course like this. And so let's now learn how to use Parcel.

\*\*\* So a devDependency is basically like a tool that we need to build our application.

But it's not a dependency that we actually include in our code. so that's why it's called a devDependency because we can use it to develop our project.

And so therefore it appears here in a new field, in our package.json file.

these libraries, that we actually include in our code, are the regular dependencies and Parcel is a devDependency

```
npm i parcel --save-dev
```

let's use Parcel in console. we do it here in the terminal

## parcel index.html

So “parcel index.html” is not going to work the reason for that is simply that this doesn't work with locally installed packages. And Parcel was indeed installed locally. So basically only on this project and that's why it showed up in the package.json file

of this exact project. So there are also global installations but more about that by the end of this session. Now, in order to still be able to use Parcel here in the console, we have two options. So we can use something called NPX or we can use NPM scripts. So let's start with NPX, which is basically an application built into a NPM.

the option that we pass into Parcel basically is “index.html” as an entry point.

because that is where we include our script.js. So basically the file that we want to bundle up. so basically in this example, the goal of using Parcel is to bundle shoppingCart.js, cloneDeep.js and script.js together.

## npx parcel index.html

So basically besides only bundling, it also does exactly the same job as our live server.

\*\*\* if you had any error installing it, you can try installing it with sudo which will give you basically more permissions.

## sudo npm i parcel

\*\*\*

\*\*\* to installing a specific ver use: **npm i parcel@1.1.1.1**

\*\*\* to uninstall use: **npm uninstall parcel**

what Parcel does is that it basically, simply, creates a script. And so now we are actually no longer using a module but we are back to using a regular script. And that is important

because modules do not work in older browsers. But now let's actually take a look at what Parcel did here. So what it did was to create this dist folder, which stands for distribution, because it is essentially gonna be this folder that we will send for production. So basically it's the code in this folder that we will send to our final users.

\*\*\* in Parcel, we can activate something even better, which is called hot module replacement.

```
if (module.hot) {  
  module.hot.accept();  
}
```

Now this code here is code that only Parcel understands. And so of course it will not make it into our final bundle because the browser is not going to understand any of it. But anyway, what hot module reloading means is that whenever we change one of the modules, it will then, of course, trigger a rebuild, like this, but that new modified bundle will then automatically, like magic, get injected into the browser without triggering a whole page reload. All right. So again, whenever we change something here, this will then not reload this part of the page. And so that's going to be amazing for maintaining state on our page whenever we are testing out something.

So this used to be something quite annoying in the past. For example, in our Bankist the application, where whenever we reloaded the page, we needed to log in again into the application. Remember that? But with Parcel and hot module replacement,

that's not going to happen, because the page will not reload. So if I save this now, then probably it's going to look the same. But again, if we had some state here on the page, then that would be maintained.

\*\*\* when we first included this cloneDeep here, from lodash, this is quite cumbersome doing it like this.:

```
import cloneDeep from "./node_modules/lodash-es/cloneDeep.js";
```

there's no need for specifying the entire path to any module. So instead we can simply do this:

```
import cloneDeep from "lodash-es";
```

is that we want to include the lodash library. And so Parcel will then automatically find the path to this module, And in fact, this works with all kinds of assets. So even with HTML or CSS or SAS files, or even images, and of course also all kinds of modules. this is also going to work with CommonJS modules. So instead of importing the ES version of lodash, we can simply import lodash like this. So just a regular version of lodash So Parcel can indeed work with all the CommonJS modules as well. And so this way we can then simply use all the modules that are available on NPM and which still use this older module format.

\*\*\* So notice the state is maintained whenever we reload the page.

We did npx, there is a second way, which is to use NPM script. So NPM scripts are basically another way of running a locally installed packages in the command line.

They also allow us to basically automate repetitive tasks. And so therefore we then don't have to write NPX Parcel and all of that, every time that we want to use a command.

```
"scripts": {  
  "start": "parcel index.html"  
3  "build": "parcel build index.html"  
},
```

Now just write : **npm run start** to run the command.

whenever we are done developing our project, it is time to build the final bundle.

**npm parcel build index.html** (or npm run build (because of created npx script in line 3))

the bundle is compressed, and has dead code elimination. So the script that we can then ship to the browser and ship to our users is this one.

we can also install packages globally. And so that would work like this:

**npm i parcel -g**

And this is actually the way that we installed the live server package before. And so, because of that, we were then simply able to use live server in every directory on our computer. So basically the big difference between globally and locally installed packages and especially these tools like Parcel or live server, is that we can use the global tools directly in the command line without the intermediate step of an NPM script. However, most of these tools actually advise developers to always install the tools locally so that they can always stay on the latest version. And so usually I follow that approach as well. And so I'm not going to install Parcel globally like this.

## ❖ 17. Modern JavaScript Development Modules, Tooling, and Functional → 11. Configuring Babel and Polyfilling

Now that we activated bundling it's time to configure Babel to transpile our super modern code back to ES5 code.

And this is still important right now even many years after the new ES6 standard has been introduced. And the reason for that is simply that there are still many people out there

who are stuck on like a windows XP or windows seven computer and two cannot upgrade their old internet explorers but we want our applications to work for everyone.

Now, the good news is that parcel actually automatically uses Babel to transpile or code.

And we can configure Babel a lot if we want to for example defining exactly what browsers should be supported but as always, that's a ton of work. And so we don't want that. And instead parcel makes some very good default decisions for us.

And so we will simply mainly just go with these defaults. Now just so you get a very broad and very general overview of how Babel works. Let's just take a look at their website. So that's Babeljs.io and then here in the documentation let's take a look here at plugins. Okay. So basically Babel works with plugins and presets that can both be configured.

Now a plugin is basically a specific JavaScript feature that we want to transpile. So to convert. So for example let's say we only wanted to convert arrow functions back to ES5 but leave everything else in ES6 for example, const and var declarations. Now usually that doesn't make a lot of sense because basically we will want to transpile everything at the same time. And so usually instead of using single plugins for each of these features,

Babel actually uses presets. And so a preset is basically a bunch of plugins bundled together. And its by default parcel is going to use this preset and preset here. And this preset will automatically select which JavaScript features should be compiled based on browser support. And again that will all happen automatically and out of the box Babel will convert all features So only browsers that are barely used anymore with the market share of less than 0.25% are not going to be supported by the transpiling with this preset here.

this preset-env does actually only include final features. So features that are already part of the language after passing the four stages of the AGMA process.

then promise is also not converted to ES5. So remember promise is an ES6 feature but again, it was not converted back to ES5 and the reason for that is that Babel can actually only transpile ES6 Syntax. So data things like arrow functions, classes, const, or the spread operator. So these are basically things that have an equivalent way of writing them in ES5.

So for example the arrow function, it is simply a different syntax. And so Babel can simply write function instead of data. And the same goes with const. So it's very easy to simply convert that to VAR but the same is not true for real new features that were added to the language like find and promise. So these new additions to the language

so these new features, they can simply not be transpiled. It's simply not possible.

Only syntax is easy to convert, so easy to compile. However, all hope is not lost. So for these added features again, such as promises or all the array methods like finds and really a bunch of other stuff, we can polyfill them. Now Babel used to do polyfilling out of the box some time ago but recently they started to simply recommending another library called **core.js**

```
npm i core.js
```

and use in js file:

```
Import core-js/stable
```

after install and importing core.js packages, if you check js file in dist folder, it's still the same and promise and find have not been replaced.

And that's because that's simply not the way it's supposed to work. So instead what polyfilling does is to basically recreate defined function and to make it available in this bundle so that the codes can then use it. `So let's search for array.prototype which remember, is where all the array methods are put. So array.prototype.find. And so, well here it is. as you just saw before the polyfilling is going to polyfill everything even if we don't need it. So we had to D find index method there as well.

Let's actually go back there. So array.prototype.find So it also created find index and every, and some and so we are actually not even using these. And so if we wanted we could cherry pick basically just the features that we actually want to polyfill. Now that of course is a lot of work but it will also greatly reduce the bundle size. And so it might be worth it if that's really a concern. So we could instead of this:

```
Import core-js/stable
```

So instead of basically importing everything there is we can just say:

```
Import 'core-js/stable/array/find'
```

And so probably only "find" was not polyfilled

the same you could do for a promise like this :

```
Import 'core-js/stable/array/promise'
```

So that's going to be a lot of work which we usually don't do but it is possible again if you are really worried about your bundle size.

there is still one feature that is not polyfilled by 'core-js/stable'. And so we always need to install just one more package

for polyfilling async functions which is called regenerator-runtime.

```
Install: npm i regenerator-runtime
```



Usage: `import "regenerator-runtime/runtime"`

## ❖ 17. Modern JavaScript Development Modules, Tooling, and Functional → 12. Review Writing Clean and Modern JavaScript

### REVIEW: MODERN AND CLEAN CODE

#### READABLE CODE

- ✚ Write code so that **others** can understand it
- ✚ Write code so that **you** can understand it in 1 year
- ✚ Avoid too "clever" and overcomplicated solutions
- ✚ Use descriptive variable names: **what they contain**
- ✚ Use descriptive function names: **what they do**

#### GENERAL

- ✚ Use DRY principle (refactor your code)
- ✚ Don't pollute global namespace, encapsulate instead
- ✚ Don't use `var`
- ✚ Use strong type checks (`===` and `!==`)

#### FUNCTIONS

- ✚ Generally, functions should do **only one thing**
- ✚ Don't use more than 3 function parameters
- ✚ Use default parameters whenever possible
- ✚ Generally, return same data type as received
- ✚ Use arrow functions when they make code more readable

#### OOP

- ✚ Use ES6 classes
- ✚ Encapsulate data and **don't mutate** it from outside the class
- ✚ Implement method chaining
- ✚ Do **not** use arrow functions as methods (in regular objects)

 Udacity

### REVIEW: MODERN AND CLEAN CODE

#### AVOID NESTED CODE

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

#### ASYNCHRONOUS CODE

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

 Udacity

## ❖ 17. Modern JavaScript Development Modules, Tooling, and Functional → 13. Let's Fix Some Bad Code Part 1

Optional chaining: ?.[user]

Nullish coalescing operator: ??

```
const limit = spendingLimits?.[user] ?? 0;
```

enhanced object literal syntax:

if the property name is the same as the variable name, we don't need to repeat that.

```
if (value <= limit) {  
  budget.push({ value: -value, description: description, user: user });  
}
```

Same as :

```
if (value <= limit) {  
  budget.push({ value: -value, description, user });  
}
```

## ❖ 17. Modern JavaScript Development Modules, Tooling, and Functional → 14. Declarative and Functional JavaScript Principles

So, we just reviewed and also implemented some clean and modern JavaScript practices. However, there is currently a major trend and shift to something called declarative code and functional programming in JavaScript. And so, let's now take some time to look at what declarative and functional programming actually are. So, there are two fundamentally different ways of writing code in programming, which we also call paradigms.

And these two paradigms, are imperative code and declarative code. Now, whenever we write imperative code, we basically need to explain to the computer how to do a certain things. So, basically, we need to explain every single step that the computer needs to follow in order to achieve a certain result. But, this might sound a little bit abstract, so let's try a more real world example. So, let's say that we want someone to bake a cake for us. And so, if we would do that in an imperative way, we would tell the person exactly the step by step recipe that they would have to follow in order to bake that cake, am I right?

So again, it is telling every single step that the person has to follow in order to achieve a result. And now bringing that back into code,

here in this code example, we are trying to double the R array. And so, this loop that I have here, is a purely imperative way of writing that. So, here we are telling the computer step by step, to create an empty array to create a counter that starts at zero, then to increase

that counter until we reach the length of the original array, and then how exactly to store the new result in each new position of the array. So, there's a lot of steps that we really give the computer here, in order for us to achieve the result of doubling that R array. Okay, so that's imperative programming, but on the other hand, we also have declarative programming, where the programmer tells the computer only what to do. And so, when we write declarative code, we simply describe the way that the computer should achieve a certain result. But the how it should do it, so basically, the step by step instructions, they get abstracted away, so we do not care about them.

And going back to our cake example here, the declarative way of instructing someone to bake the cake would be to simply describe that cake to the person, and then the person would have to come up with the step by step recipe on their own. So, simply describing the task, and the result that should be achieved is the declarative way of doing it, all right?

And now coming back to the code example of duplicating the values in an array, this is how we do it in the declarative way. So, we have R array, and then we simply tell JavaScript, that it should map the values in the R array to a new array, and each of these values should be multiplied by two. And so, if you compare this code example, with the one on the left, then you will really see that in this example, all we are doing is describing the way that the computer should achieve the result that we are looking for.

We are simply telling it what to do, which in this case, is to simply map the original array onto a new array and doubling all the elements. But, all these super detailed steps that we have on the left side, like creating an empty array and initializing a counter, all of these steps have been abstracted away, because we don't really care about them, all right? And this is pretty important to understand, because more and more this is how modern JavaScript code is actually written. So, the difference between imperative and declarative is not just some theoretical difference. So, the declarative paradigm is actually a really big and popular programming paradigm, which has even given rise to a sub paradigm called, functional programming. And functional programming, is basically a declarative paradigm, which is based on the idea of writing software, simply by combining multiple so called pure functions, while avoiding side effects and mutating data. And actually, functional programming and writing declarative code, has now basically become the modern way of writing code in the JavaScript world. So, you will see declarative and functional code everywhere. And, in fact, we have even been using it all along, but without really knowing that this style was called declarative, and functional, all right. But let's quickly go back to the definition of functional programming, and talk about what side effects and pure functions are. So, a side effect is basically simply a modification of any data that's outside of a function. So, for example, mutating any variable that is external to the function is causing a side effect. So basically, any variable that is outside of the scope of the function, all right? Now, data does not only refer to variables, so for example, logging stuff to the console, or also changing something in the DOM, is also causing side effects. Now next up, a pure function, is a function without side effects. So, basically a function that does not mutate any external variables, and that does also not depend on any external variables. So basically, if we give the same inputs to a pure function, it will always return the same output and again, that's because it does not depend

on any external variables, and it also does not manipulate them. And finally, if we look again, at our definition here, we also see that functional programming is about avoiding mutating data, and we do that by using something called immutability.

So, in functional programming state, which also means basically data is never modified. So, let's say that we have some application, and we have an object there to keep track of all the data that we need in an application.

And so that we usually called state, and so again, in functional programming, that state is never modified. Instead, what we will do is to copy that object, so that state, and then it is that copy that is mutated, and can then be returned, but the original state is never touched, okay? So, that's what it means for the state being immutable, and the big upside of immutability is that, it makes it so much easier to keep track of how the data flows through our entire application. And so ultimately, that will allow us to write better code with less bugs, and code that is also more readable, which overall, is the entire goal of using functional programming in the first place.

Now, I'm telling you all this, not with the goal of turning you into a functional programmer, because that would actually be a very hard task, because this is really just a very high level introduction to what functional programming actually is. But behind the surface, functional programming is a huge paradigm, which is really difficult to implement in practice. But it is still very important that you know, some of these principles, such as side effects, pure functions, and immutability, because many of the popular libraries, such as React or Redux, are actually built around all of these principles. So for example, in React, the state is also completely immutable, and so if you ever want to learn something like React, you will need to know about these concepts in order to use it properly. However, some principles such as pure functions, or side effects, can actually be a bit easier to implement into our own code. So, what I'm trying to say is that, we can actually mix imperative and declarative programming in our own codes, we don't have to go 100% declarative. Or in other words, we don't have to go 100% in the direction of making our code completely functional. And so again, we can already start using, some of the functional programming techniques in our own code base. So, for example, you can try to avoid data mutations as often as possible. And of course, this will not always be possible, but it's also not really necessary. So, these are mainly and are just suggestions, but which will still create more readable and overall better and cleaner code. So, another thing that you can do is to always prefer, built in methods or functions that do not produce side effects over the ones that do, and this is really important for data transformations. So, whenever you want to do that, you should use a method such as Map, Filter and Reduce. So, this is the functional and modern way of doing data transformations, and many times, this is actually the first contact that many people have, with functional programming. So, Map, Filter and Reduce are actually present in all functional programming languages, and they are very important to implement a functional code into more declarative code in our code.

And finally, you can also try to avoid side effects into functions that you write yourself. And again, this is of course, not always possible, and also not always necessary. So, we will never be able to avoid all side effects in applications, because of course, at some point, the application needs to do something.

So, it needs to display something on the DOM, or log something to the console, or really create some side effect, okay? But you can still try to think about this, and to start incorporating side effects more into your own code. And now to finish, let's come back to declarative syntax, because functional programming is only a part of using and writing declarative code. So, in order to write code that is more declarative, you should use array and object destructuring whenever that's possible. You should also use the spread operator, the ternary operator, and also template literals whenever that is possible, because if you think about it, then all of these four ways of writing code, actually makes the code more declarative. So, these operators are more about telling the code what to do, and not exactly the steps that it should take, right?

And that's, again, true for all these four pieces of syntax.

All right, so let's now actually continue working on the code example from last lecture and implement, some of these functional programming principles in practice.

## IMPERATIVE VS. DECLARATIVE CODE

Two fundamentally different ways of writing code (paradigms)

### IMPERATIVE

- Programmer explains "HOW to do things"
- We explain the computer *every single step* it has to follow to achieve a result
- Example:** Step-by-step recipe of a cake

```
const arr = [2, 4, 6, 8];
const doubled = [];
for (let i = 0; i < arr.length; i++)
  doubled[i] = arr[i] * 2;
```

### DECLARATIVE

- Programmer tells "WHAT do do"
- We simply *describe* the way the computer should achieve the result
- The **HOW** (step-by-step instructions) gets abstracted away
- Example:** Description of a cake

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

the declarative paradigm is actually a really big and popular programming paradigm, which has even given rise to a sub paradigm called, functional programming. And functional programming, is basically a declarative paradigm, which is based on the idea of writing software, simply by combining multiple so called pure functions, while avoiding side effects and mutating data. And actually, functional programming and writing declarative code, has now basically become the modern way of writing code in the JavaScript world.

# FUNCTIONAL PROGRAMMING PRINCIPLES

## FUNCTIONAL PROGRAMMING

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

➤ **Examples:**  **React**  **Redux**

## FUNCTIONAL PROGRAMMING TECHNIQUES

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

## DECLARATIVE SYNTAX

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

## ❖ 17. Modern JavaScript Development Modules, Tooling, and Functional → 15. Let's Fix Some Bad Code Part 2

that we started to work on a bit earlier. And so in this lecture, we're gonna focus on some of the, functional and declarative principles So that's first immutability, second, side effects and pure functions, and third making data transformations using pure functions, such as map filter and reduce.

And let's actually start with immutability

Before:

```
const spendingLimits = {
  jonas: 1500,
  matilda: 100,
};
```

After:

by adding `Object.freeze()`, spending limits is now immutable, which means that we can no longer put any new properties into it.

```
const spendingLimits = Object.freeze({
  jonas: 1500,
  matilda: 100,
});
```

```
spendingLimits.julia = 300
console.log(spendingLimits); // julia not added
```

\*\*\* we can also do object dot freeze on arrays. because in the end, an array is also just an object:

```
const budget = Object.freeze([
  { value: 250, description: 'Sold old TV 📺', user: 'jonas' },
  { value: -45, description: 'Groceries 🛒', user: 'jonas' },
  { value: 3500, description: 'Monthly salary 💰', user: 'jonas' },
  { value: 300, description: 'Freelancing 💻', user: 'jonas' },
  { value: -1100, description: 'New iPhone 📱', user: 'jonas' },
  { value: -20, description: 'Candy 🍬', user: 'matilda' },
  { value: -125, description: 'Toys 🧸', user: 'matilda' },
  { value: -1800, description: 'New Laptop 💻', user: 'jonas' },
]);
```

\*\*\*Object dot freeze here basically only freezes the first level of the object. So it's not a so-called deep freeze because we can still change objects inside of the object.

```
budget[0].value = 10000; // value changed
budget[9].value = 'alex' // not added (for adding new elem on first level)
```

\*\*\* a side effect basically means that something outside of a function is manipulated or that the function does something other than simply returning a value,

```
const budget = Object.freeze([
  { value: 250, description: 'Sold old TV 📺', user: 'jonas' },
  { value: -45, description: 'Groceries 🛒', user: 'jonas' },
  { value: 3500, description: 'Monthly salary 💰', user: 'jonas' },
  { value: 300, description: 'Freelancing 💻', user: 'jonas' },
  { value: -1100, description: 'New iPhone 📱', user: 'jonas' },
  { value: -20, description: 'Candy 🍬', user: 'matilda' },
  { value: -125, description: 'Toys 🧸', user: 'matilda' },
  { value: -1800, description: 'New Laptop 💻', user: 'jonas' },
]);

budget[0].value = 10000; // value changed
budget[9].value = 'alex' // not added (for adding new elem on first level)
const spendingLimits = Object.freeze({
  jonas: 1500,
  matilda: 100,
});

spendingLimits.julia = 300
console.log(spendingLimits); // julia not added

const getLimit = user => spendingLimits?.[user] ?? 0;

const addExpenditure = function (value, description, user = 'Jonas') {
```

```

21 user = user.toLowerCase();

    // const limit = spendingLimits[user] ? spendingLimits[user] : 0;

22 if (value <= getLimit(user)) {
23     budget.push({ value: -value, description, user });
    }
};
addExpende(10, 'Pizza 🍕');
addExpende(100, 'Going to movies 🎬', 'Matilda');
addExpende(200, 'Stuff', 'Jay');

const checkExpenses = function () {
    for (const entry of budget)
        if (entry.value < -getLimit(entry.user)) entry.flag = 'limit';
};
checkExpenses();

const logBigExpenses = function (bigLimit) {
    let output = '';
    for (const entry of budget)
        output += entry.value <= -bigLimit ? `${entry.description.slice(-2)} ` : '';

    output = output.slice(0, -2); // Remove Last ' / '
    console.log(output);
};

console.log(budget);
logBigExpenses(500);

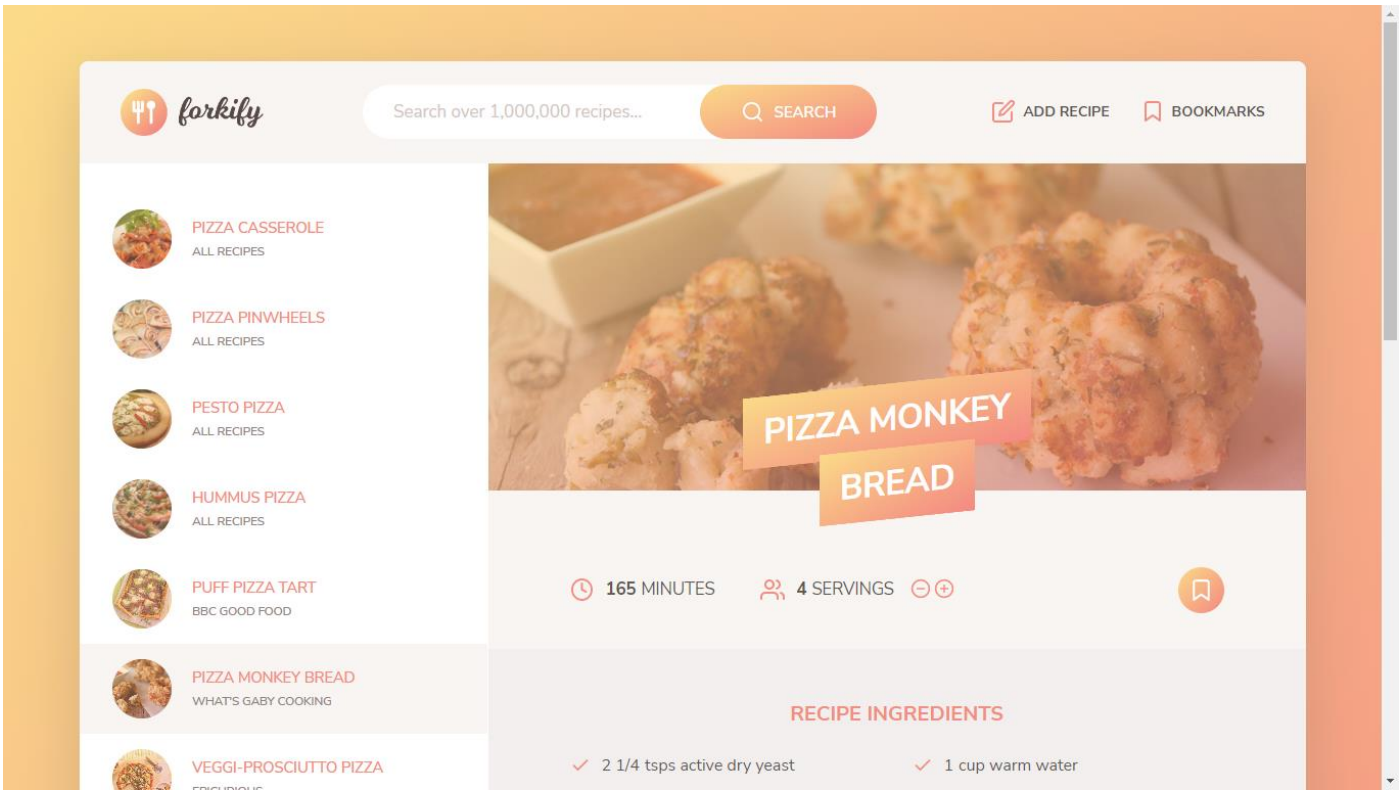
```

And so a function that has, or that produces side effects is called an impure function. So this function at expense right now is an impure function because it does attempt to manipulate (line 23) and to mutate this object that is located outside of it.

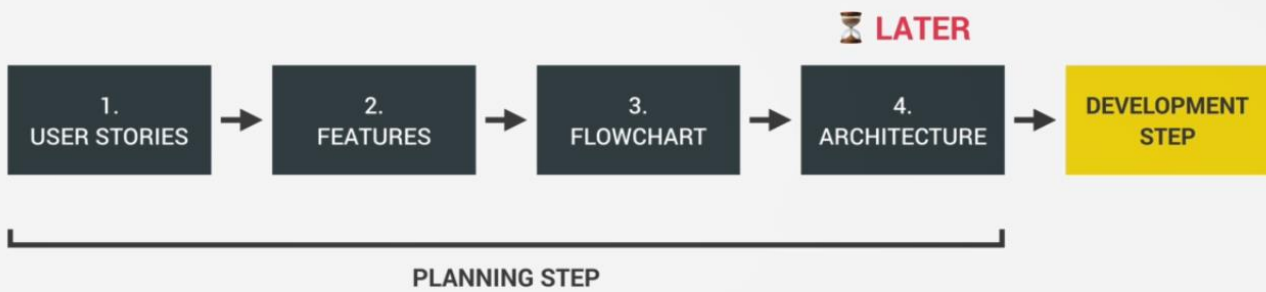
So remember that the solution to that is to create a copy and then return that copy of the state. So of the data,

## ❖ 18. Forkify App Building a Modern Application → 3. Project Overview and Planning (I)





## PROJECT PLANNING



- **User story:** Description of the application's functionality from the user's perspective.
- **Common format:** As a *[type of user]*, I want *[an action]* so that *[a benefit]*

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

## USER STORIES

## FEATURES

1 Search for recipes

2 Update the number of servings

3 Bookmark recipes

4 Create my own recipes

5 See my bookmarks and own recipes when I leave the app and come back later

- Search functionality: input field to send request to API with searched keywords
- Display results with pagination
- Display recipe with cooking time, servings and ingredients

- Change servings functionality: update all ingredients according to current number of servings

- Bookmarking functionality: display list of all bookmarked recipes

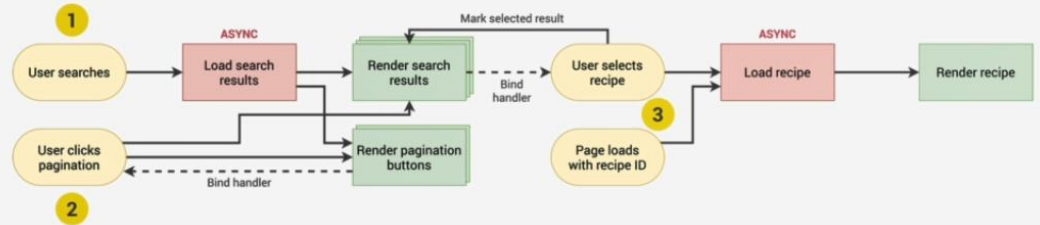
- User can upload own recipes
- User recipes will automatically be bookmarked
- User can only see their own recipes, not recipes from other users

- Store bookmark data in the browser using local storage
- On page load, read saved bookmarks from local storage and display

## FEATURES

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

Other features later



## ❖ 18. Forkify App Building a Modern Application → 3. Project Overview and Planning (I)

Development practice \_ just review the video if you need