

The Complete JavaScript Course 2022: From Zero to Expert!

1. A brief Introduction to Javascript
2. JavaScript Fundamentals part 1
3. JavaScript Fundamentals part 2
4. Developer Skills and Vs console setup
5. JavaScript in the Browser: DOM and Event Fundamentals - Guess my Number
6. JavaScript in the Browser: DOM and Event Fundamentals - pop up modal window
7. JavaScript in the Browser: DOM and Event Fundamentals – Pig Game
8. How JavaScript works behind the Scenes
9. Data Structures, Modern Operators and Strings
10. A closer look at functions
11. Working with Arrays
12. Numbers, Dates, Intl and Timers
13. Advanced DOM and Events
14. Object Orientated Programming (OOP) with JavaScript
15. Mapty App: OOP, Geolocation, External Libraries and more
16. Asynchronous JavaScript: Promises, Async/Await, AJAX
17. Modern JavaScript Development: Modules, Tooling, and Functional
18. Forkify App: Building a Modern Application
19. Setting up GIT and Deployment

A BRIEF INTRODUCTION TO JAVASCRIPT

Javascript is one of three core languages that are used to build websites.

HTML 5 is responsible for the content of the page, text, images, links etc.

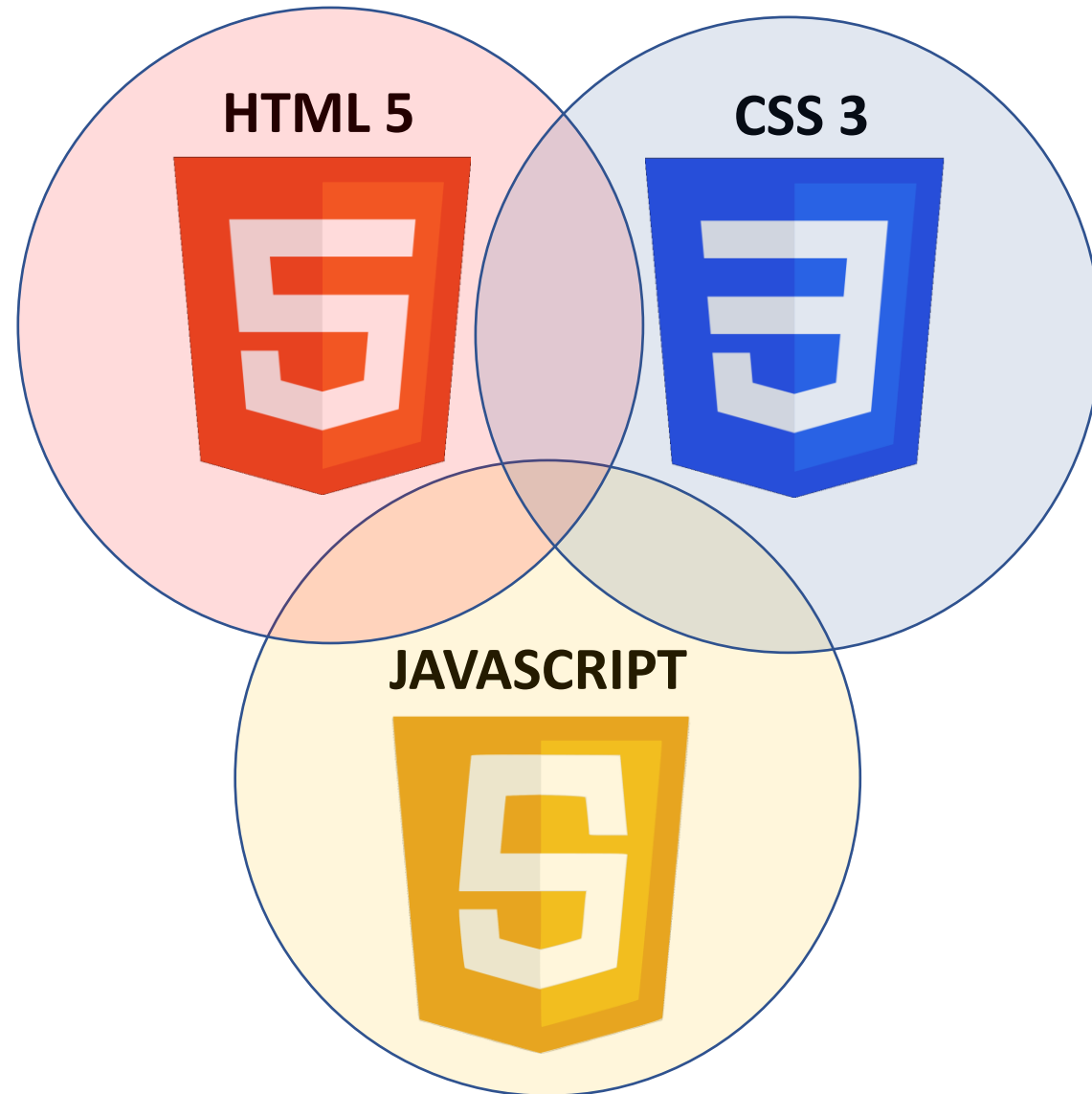
CSS 3 is responsible for the presentation. I.e layout, positioning, font sizes, colours etc.

JS is used to add dynamic and interactive effects to the webpage, manipulate content or CSS and build entire web applications in the browsers.

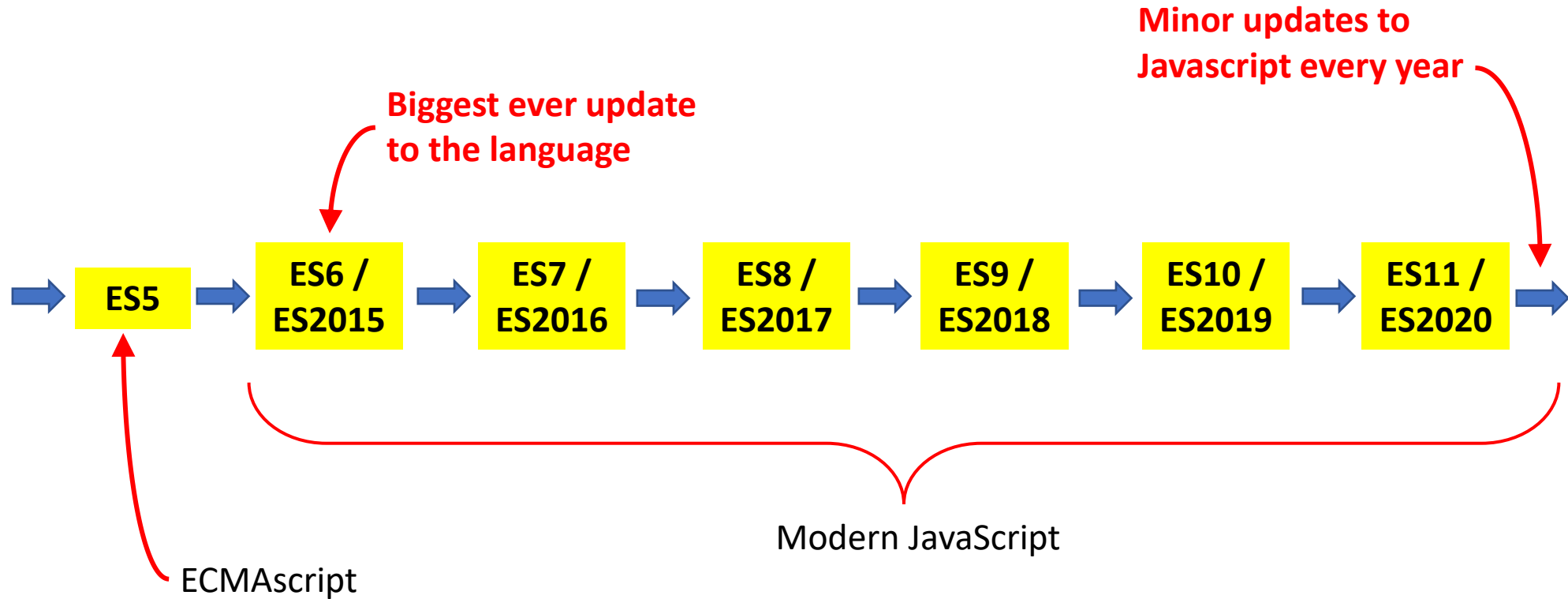
JavaScript has many popular frameworks that allow us to write code easily. Examples are jQuery, angular etc.

Javascript is a front end language that interacts with the user and the backend languages like PHP. JavaScript can also be run in the backend server-side as a browser independent application using tools like Nodejs.

A brief Introduction to Javascript



Javascript Releases

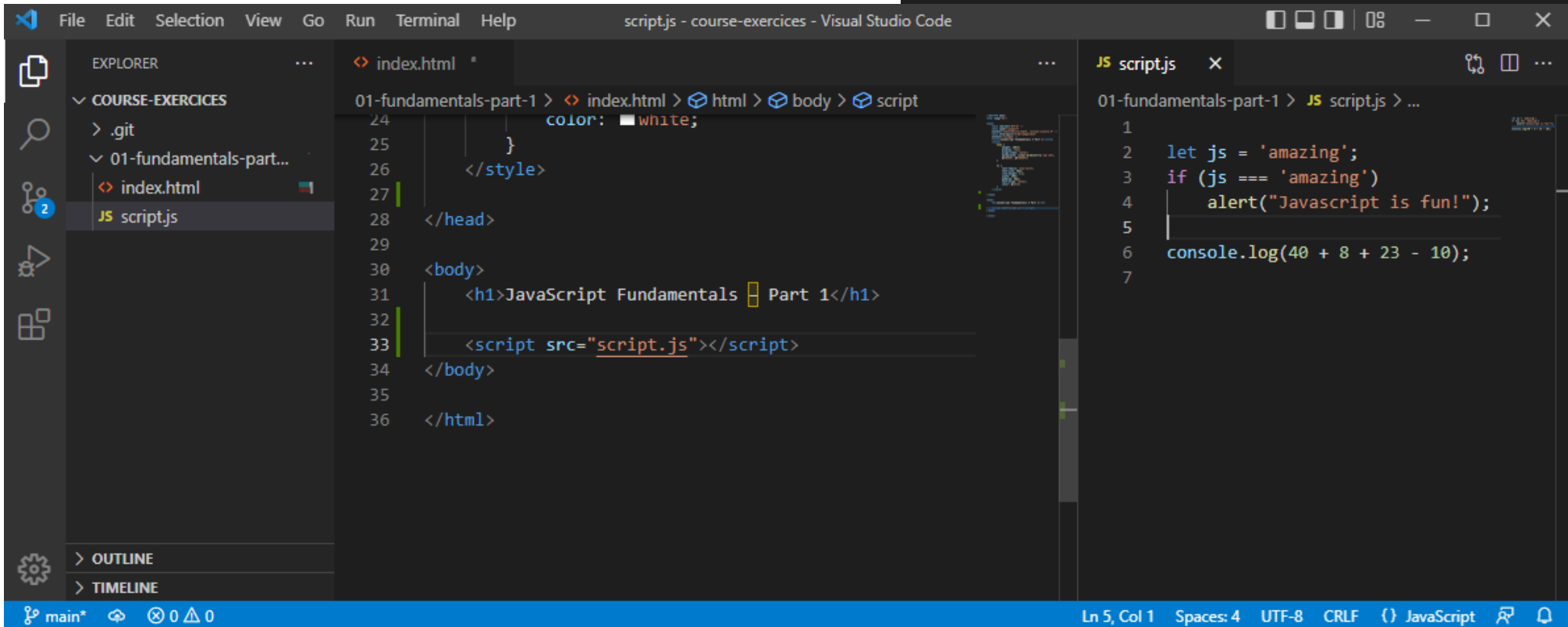


JAVASCRIPT FUNDAMENTALS PART 1

This is an example of inline script because the script code is within the html file within `<script>` tags.

Better to create a separate external Javascript file and link to it in the html file.

```
22     padding: 30px;
23     text-align: center;
24     color: white;
25   }
26 </style>
27
28 <script>
29   let js = 'amazing';
30   if (js === 'amazing')
31     alert("Javascript is fun!");
32
33   console.log(40 + 8 + 23 - 10);
34 </script>
35
36 </head>
```



Javascript 7 primitive data types

- 1 **Number** Floating point numbers. Used for decimals and integers i.e. *let age = 23;*
- 2 **Strings** Used for text. Always put strings in quotes. *Let firstName = "Jonus";*
- 3 **Boolean** Logical values than can be true or false. *Let fullAge = true;*
- 4 **Undefined** Value taken by a variable that is not yet defined. *Let children;*
- 5 **Null** Also means empty value.
- 6 **Symbol** Value that is unique and cannot be canged (Not usefull for now)
 (ES2015)
- 7 **BigInt (ES** For integers that are too large to be represented by the number type.
 2020)

Javascript has **dynamic typing**: We do not have to manually define the data type of a value in a variable. Javascript will **automatically** determine the data type.

Note that it is the value that has datatype and not the variable.

Variable X can be a Number then it can be a string in our code later on. This can cause some tricky bugs.

Let, const & var – 3 ways to define variables.

var let const

Var is the old way to define variables pre ES6.

Var should not be used but may exist in older code. Using var is still valid but it is a bad practice.

Let is used to define variables that may change later on in our code.

For example age is a variable that may change. It mutates yearly increasing by one.

Const is used to define variables that never change.

For example date of birth. This is an immutable variable because the birthday of a person can never change.

Because a const variable is immutable we cannot declare and empty variable.

Const should always be used in preference to let unless you are absolutely sure that the variable will mutate later in the code. This is to avoid possible future bugs.

Javascript basic Operators.

Mathematical or Arithmetic operators

Multiplication, Division addition, subtraction and to the power of.

Note that the plus symbol is also used to concatenate or join together, i.e. firstName and lastName

Assignment operators

Equals (=) , plus Equals (+=) , minus Equals (-=) , divide Equals (/=) , multiply Equals (*=) , plus plus (++) , minus minus (--)

Comparison operators

Used with Boolean. i.e. is age Jonas greater than age Sarah? True or false?

Grater than (>) , Less than (<) , grater than or equal to (=>) , less than or equal to (=>)

Operator Precedence

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence

```
console.log(now - 1991 > now - 2018);
```

Consulting the operator precedence table on the above website we see that Minus has a higher precedence number than > so it is executed first.

Operator precedence also has **Associativity**. I.e. if the code is executed left-to-right or right-to-left.

Strings and Template Literals

```
const firstName = "Jonas";  
const job = "teacher";  
const birthYear = 1991;  
const year = 2037;
```

We use concatenation with strings to produce a the text output of I'm Jonas, a 46 year old teacher!

```
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);
```

A better way to achieve this would be using template literals. Here the variables are placed inside curly braces and preceded by a dollar sign. Note that the start and end of the string is defined by a backtick ` symbol which on a Spanish keyboard is found to the left of the P key.



```
console.log(`Just a regular string....`)
```

We also use backticks to output directly a string.

If Else statements.

```
const age = 17;

if (age >= 18) {
  console.log(`Sarah can start learning to drive🚗`);
} else {
  yearsLeft = 18 - age;
  console.log(`Sarah can't start learning to drive for ${yearsLeft} years 🛵`);
}
```

Note that the else block is optional. If no else statement is present and the condition is false then JavaScript will just move onto the following line of code.

Emoji's are found by using the windows and dot key.




```
const birthyear = 1991;
if (birthyear <= 2000) {
  let century = 20;
} else {
  let century = 21;
}

console.log(century);
}
```

Note that the variable `century` is defined within the if else code block and so will produce an error if we try to call it outside the if else code block.

Uncaught ReferenceError: `century` is not defined.

```
const birthyear = 2001;
let century;

if (birthyear <= 2000) {
  century = 20;
} else {
  century = 21;
}

console.log(century);
```

so we first need to define an empty variable outside of the if else block to be able to call `century` outside of the if else statement.

Type Conversion & Coercion

Type conversion is when we manually convert from one data type to another, I.e. String to number.

`const inputYear = "1991";` The year is a string not a number because it is in quotes.

`console.log(inputYear + 18);` This will concatenate to give output of 199118.

`console.log(Number(inputYear) + 18);` Number will convert the string to a number

`console.log(Number("Jonas"));` When a string cannot be converted to a number we get NaN – Not a Number.

`console.log(String(23), 23);`

`console.log(typeof String(23));` We can convert a number to a string.

Type coercion is when JavaScript automatically does the type conversion behind the scenes.

`console.log('I am ' + 23 + ' years old');` Here JavaScript is automatically treating 23 as a string.

`console.log("23" - "10" - 3);` JavaScript does type coercion here to convert strings to numbers outputting 10.

`console.log("23" * "2");` JavaScript does type coercion here to convert both strings to numbers outputting 46.

What value will be logged to the screen?

```
let n = "1" + 1;  
n = n - 1;  
console.log(n);
```

JavaScript does type coercion to string so "1" + "1" will be 11.
Now n with a value of 11 is treated as a number so the output will be 10.

```
let x = 2 + 3 + 4 + "5";  
console.log(x);
```

JavaScript will treat 2 + 3 + 4 as a number giving a value of 9
then concatenate the 5 onto string 9 -> "95"

```
let y = "10" + "4" - "3" - 2 + "5";  
console.log(y);
```

JavaScript will treat 10 - 4 - 3 as a number giving a
value of 1 then concatenate the 1 onto string 5 -> "15"

Truthy values & Falsy Values

5 Falsy Values: Zero 0, Empty string "", undefined, Null and NaN

```
console.log(Boolean(0)); // False
console.log(Boolean(undefined)); // False
console.log(Boolean("Jonas")); // True - Strings are truthy when converted to Boolean.
console.log(Boolean({})); // True
console.log(Boolean("")); // False
```

JavaScript performs type coercion in a Boolean operation.

```
const money = 0;
if (money) {
  console.log("Don't Spend it all;");
} else {
  console.log("You have no money!");
}
```

What operation will be logged? The If or the Else?

The zero will be treated as a falsy Boolean so the else statement will be executed.

```
const money = 100;
if (money) {
  console.log("Don't Spend it all;");
} else {
  console.log("You have no money!");
}
```

100 is a truthy Boolean so the if statement will be executed.

Boolean can be used to check if a variable is defined.

```
let height;  
if (height) {  
    console.log('YAY! Height is defined');  
} else {  
    console.log('Height is NOT defined');  
}
```

Height is UNDEFINED and therefore a falsy value so the else statement will be executed.

```
let height = 0;  
if (height) {  
    console.log('YAY! Height is defined');  
} else {  
    console.log('Height is NOT defined');  
}
```

Height is set to zero and therefore a falsy value so the else statement will be executed, even though height is defined.

```
let height = 10;  
if (height) {  
    console.log('YAY! Height is defined');  
} else {  
    console.log('Height is NOT defined');  
}
```

Height is set to 10 and therefore a truthy value so the if statement will be executed.

Equality Operators == or ===

```
const age = 18;  
if (age === 18) console.log('You Just became an adult');
```

Boolean True

```
const age2 = "18";  
if (age2 == 18) console.log('You Just became an adult');
```

Boolean True

```
const age3 = "18";  
if (age3 === 18) console.log('You Just became an  
adult');
```

Boolean False

=== returns a Boolean value if the results match strictly.

== returns a Boolean value if the results match. Note that this is not strict so JavaScript does type coercion.

=== returns a Boolean value if the results strictly match. Note a string and a number do not match strictly because they are different data types.

The loose equality operator of == can introduce a lot of unusual bugs in JavaScript so it is always better to use strict === and perform manual type conversion if needed.

```
const favorite = prompt("what is your favorite number?");  
console.log(favorite);
```

The prompt asks the user for a number and of course we input 23.

```
if (favorite == 23) {  
    console.log('Cool! 23 is an amazing number!');  
}
```

When we run the if statement using loose equality operator == the Boolean is true.

```
if (favorite === 23) {  
    console.log('Cool! 23 is still an amazing number!');  
}
```

However when we run the if statement with strict equality then the Boolean is false and it returns nothing.

```
console.log(typeof favorite);
```

This is because the user input from the prompt is treated as a string.

```
const favorite2 = Number(prompt("what is your favorite number?"));  
console.log(favorite2);
```

```
if (favorite2 === 23) {  
    console.log('Cool! 23 is still an amazing number!');  
}
```

To get around this perform manual type conversion of the prompt into a number.

```
const favorite3 = Number(prompt("what is your favorite number?"));
console.log(favorite3);
```

```
if (favorite3 === 23) {
    console.log('Cool! 23 is still an amazing number!');
} else if (favorite3 === 7) {
    console.log('Cool! 7 is a lucky number!');
} else if (favorite3 === 9) {
    console.log('Cool! 9 is the square of three!');
} else {
    console.log('Number is not 7 or 23');
}
```

```
if (favorite3 !== 23) console.log('Why not 23?');
```

Else if can be used to add more options to the statement.

!== means not equal to (strict mode)

!= Means not equal to (loose mode)

An input of 9 Cool! 9 is the square of three!
 Why not 23?

An input of 7 Cool! 7 is a lucky number!
 Why not 23?

An input of 23 Cool! 23 is still an amazing number!

All other numbers Number is not 7 or 23!
 Why not 23?

Boolean Logic – And // Or // Not

A: Sarah has a driving license

B: Sarah has good Vision

 Boolean values that can be TRUE or FALSE

A and B: Sarah has a driving license **AND** good vision

AND	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

Truth table for AND

True when ALL are true

A or B: Sarah has a driving license **OR** good vision

OR	TRUE	FALSE
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

Truth table for OR

True when ONE is true

 No matter how many variables

Not A, Not B

Inverts true/false value.

Boolean Variables that can be TRUE or FALSE

A: Age is greater than or equal to 20 **FALSE**

B: Age is less than 30 **TRUE**

Age = 16

A:

AND	TRUE	FALSE
B: TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

JavaScript Operators

(Not A) !A FALSE = TRUE	A Or !B FALSE FALSE = FALSE
A and B FALSE TRUE = FALSE	!A and B TRUE TRUE = TRUE
A or B FALSE TRUE = TRUE	

A:

OR	TRUE	FALSE
B: TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

Logical Operators

```
const hasDriversLicense = true; Variable A  
const hasGoodVision = true;    Variable B
```

&& = And

|| = Or

!= Not

```
console.log(hasDriversLicense && hasGoodVision);  
console.log(hasDriversLicense || hasGoodVision);  
console.log(!hasDriversLicense);
```

```
const shouldDrive = hasDriversLicense && hasGoodVision;
```

```
if (hasDriversLicense && hasGoodVision) {  
    console.log("Sarah is able to drive!")  
} else {  
    console.log("someone else should drive....")  
}
```

```
const isTired = true; Variable C
```

```
console.log(hasDriversLicense && hasGoodVision && isTired);
```

```
if (hasDriversLicense && hasGoodVision && !isTired) {  
    console.log("Sarah is able to drive!")  
} else {  
    console.log("someone else should drive....")  
}
```

The Switch Statement

```
const day = "monday";

switch (day) {
  case 'monday':
    console.log('Plan Course Structure');
    console.log('Go to coding meetup');
    break;
  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');
    break;
  default:
    console.log('not a valid day!');
}
```

The switch statement is like a string of If else statements.

We define the true condition with a case, followed by the code we want to operate then a break.

Default is used at the end to cover a scenario where none of the cases match the true clause.

```
const dayOfWeek = 'Saturday';
```

```
if (dayOfWeek === 'Monday') {  
    console.log('Plan Course Structure');  
    console.log('Go to coding meetup');  
} else if (dayOfWeek === 'Tuesday') {  
    console.log('Prepare theory videos');  
} else if ((dayOfWeek === 'Wednesday') || (dayOfWeek === 'Thursday')) {  
    console.log('write code examples');  
} else if (dayOfWeek === 'Friday') {  
    console.log('Record Videos');  
} else if ((dayOfWeek === 'Saturday') || (dayOfWeek === 'Sunday')) {  
    console.log('Enjoy the weekend');  
} else {  
    console.log('not a valid day!');  
}
```

The switch statement written as If else would be something like this.

Statements & Expressions

An Expression is a line of code that produces a value. For example:

`3 + 4` The expression is `3 + 4`.

`True && false && !true`

`1991`

A Statement is an action to be performed, like a complete sentence.

```
if (23 > 10) {  
    const str = "23 is bigger";  
}
```

This is important because JavaScript expects statements and expressions in certain places. For example we cannot insert a statement in a template literal.

```
const me = 'Jonas'  
console.log(`i'm ${2037 - 1991} years old and my name is ${me}`)
```

We can insert variables and values into the template literal because they are expressions.

```
console.log(`i'm ${if (23 > 10) { const str = "23 is bigger"; }}`)
```

But we cannot insert an if statement into a template literal.

The Conditional Ternary Operator

The ternary operator can be used to replace an if else with one line of code. It is called a ternary operator because it has 3 parts.

```
const age = 19;
```

Condition

(age is greater than 18) True expression

false expression

```
age > 18 ? console.log('I drink wine🍷') : console.log('I drink water🥛');
```

Condition ? True expression : false expression

Conditional ternary operators are really useful for declaring a variable and a lot shorter than writing a full if else statement.

```
const drink = age > 18 ? 'wine🍷' : 'water🥛';  
console.log('I like to drink ' + drink)
```

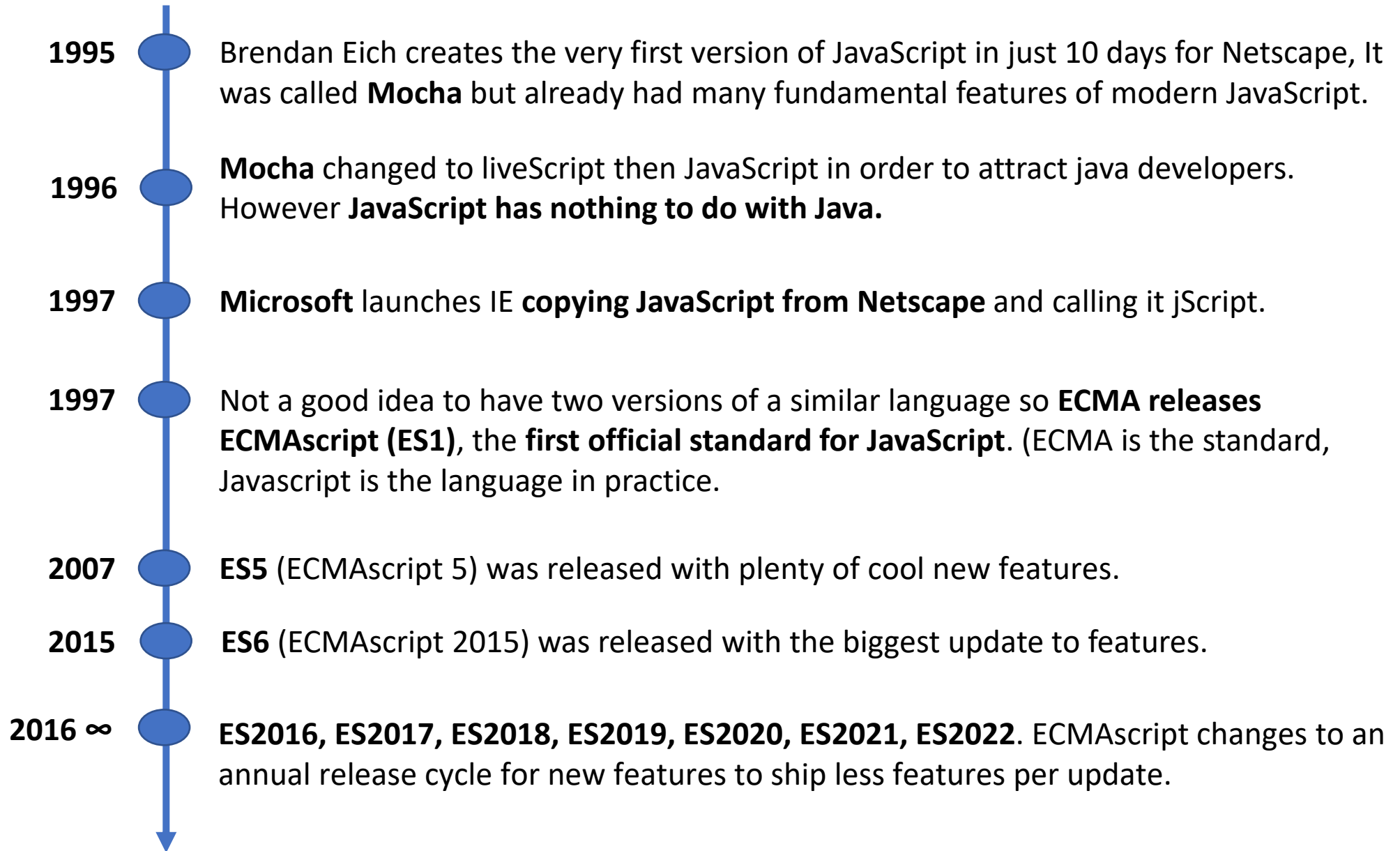
```
let drink2;  
if (age <= 18) {  
    drink2 = 'wine🍷';  
} else {  
    drink2 = 'water🥛';  
}  
console.log('I like to drink ' + drink2)
```

Additionally, because a ternary operator is an expression it can be inserted into a template literal.

```
console.log(`I like to drink ${age > 18 ? 'wine🍷' : 'water🥛'} `);
```

Ternary operator is not a replacement for if else but a substitution for a quick calculation for two options.

JavaScript Releases: ES5, ES6+ and ESNext



JavaScript has full backwards compatibility. If we take code written in 1997 then it will still work today in the browser.

The guiding principle behind this is to not break the web. Websites keep working forever.

Forwards compatibility: If we take code written in 2009 and try and run it in a browser from 1997 then it will not work!

How can we use JavaScript today when coding for possible users that have an outdated browser?

Well as a developer all we have to do is use the most up to date version of a browser with dev tools.

When we finish the code we can use a tool like babel to transpile and polyfill the code converting it back to an ES5 version to ensure most browser compatibility for most users.

ES5

Fully supported in all browsers back to IE9 from 2011 → **Ready to be used today.**

ES6



ES2022

ES6+ Fully supported in most modern browsers

No support in older browsers

→ **Can use most features with transpiling and polyfilling**

ES2022

∞

Esnext New versions of the language (new feature proposals that reach stage 4)

→ **Can already use SOME features in production with transpiling and polyfilling**

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

<https://kangax.github.io/compat-table/es2016plus/>

JAVASCRIPT FUNDAMENTALS PART 2

Activating Strict Mode in JavaScript

Strict Mode allows us to write better, more secure, less buggy JavaScript code.

`'use strict';` Enable strict mode with use strict command.

This command takes effect for all lines of JavaScript code below the use strict line so it is important that this appears at the very top of our JavaScript file.

Strict mode prohibits us from doing certain things and also shows errors in our code that may otherwise be silent errors.

```
let hasDriversLicense = false;
const passTest = true;

if (passTest) hasDriverLicense = true;
if (hasDriversLicense) console.log('I can drive!');

'use strict';

let hasDriversLicense = false;
const passTest = true;

if (passTest) hasDriverLicense = true;
if (hasDriversLicense) console.log('I can drive!');
```

Note in the first if statement that the variable `hasDriversLicense` is spelled wrong. Without strict mode enabled this will produce a silent error because the second if statement will not be executed. Console log will be blank.

But with strict mode it throws an error. “Uncaught ReferenceError: `hasDriverLicense` is not defined”

Strict Mode also protects us from using possible future protected words or actual protected words.

```
'use strict';
```

```
const interface = 'Audio'; Uncaught SyntaxError: Unexpected strict mode reserved word
```

```
const private = 1; Uncaught SyntaxError: Unexpected strict mode reserved word
```

```
const if = 10; Uncaught SyntaxError: Unexpected token 'if'
```

Interface and private are protected words that are reserved for future JavaScript features and If is a protected word that cannot be used for a variable name.

JavaScript Functions

A function is a block of code that we can reuse multiple times in our project to keep the code DRY – Don't Repeat yourself!

```
function logger() {  
  console.log('my name is Jonas');  
}
```

```
logger();  
logger();  
logger();
```

We have a function called logger which performs a console log of a text string. We can then call or run or invoke that function multiple times, in this case three times

```
my name is Jonas    script.js:19  
my name is Jonas    script.js:19  
my name is Jonas    script.js:19
```

We have a fruit processor function that returns apple and orange juice.

```
function fruitProcessor(apples, oranges) {  
  const juice = `juice with ${apples} apples and ${oranges} oranges.`;  
  return juice;  
}
```

```
const appleJuice = fruitProcessor(5, 0);  
console.log(appleJuice);
```

We can use that function to call an apple juice from 5 apples!

```
const orangeJuice = fruitProcessor(0, 2);  
console.log(orangeJuice);
```

We can use that function to run an orange juice from 2 oranges!

```
const mixedJuice = fruitProcessor(1, 1);  
console.log(mixedJuice);
```

We can use that function to invoke an orange and apple juice from 1 apple and 1 orange!

JavaScript Function Declarations vs Expressions

The food processor was a function declaration where we simply declare it as a function

```
function calcAge1(birthYear) {  
    return 2037 - birthYear;  
}
```

CalcAge1 is also a function declaration where we have a placeholder called birthyear that we can insert values into when we call that function.

```
const age1 = calcAge1(1991);  
console.log(age1);
```

```
const calcAge2 = function  
(birthYear) {  
    return 2037 - birthYear;  
}
```

CalcAge2 is also function expression or anonymous function where we define a variable as a function then invoke that variable latter.

```
const age2 = calcAge2(1991);  
console.log(age2);
```

Remember that expressions produce values to the variable calcAge2 just holds a value such as a number, string or Boolean.

We can call function declarations BEFORE they are defined in the code whereas function expressions cannot be called before they are written in the code. This is due to Hoisting which will be covered in a future lesson.

```
const age3 = calcAge1(1991);  
console.log(age3);
```

```
// function declaration  
function calcAge1(birthYear) {  
    return 2037 - birthYear;  
}
```

```
const age4 = calcAge2(1991);  
console.log(age4);
```

```
// function expression  
const calcAge2 = function (birthYear) {  
    return 2037 - birthYear;  
}
```

Uncaught ReferenceError: Cannot access 'calcAge2' before initialization

What type of function should be used? Function declaration or Function expression? Well function expression forces you to define all the functions at the top of the code with clearly defined variables but it is really down to personal choice.

Arrow Function (new in Javascript ES6)

```
const calcAge3 = birthYear => 2037 - birthYear;
```

```
const age5 = calcAge3(1991);  
console.log(age5);
```

An arrow function is a shorthand way of writing a function expression with the return implicitly running.

```
const yearsUntilRetirement = birthYear => {  
  const age = 2037 - birthYear;  
  const retirement = 65 - age;  
  return retirement;  
}
```

If we want the arrow to perform multiple calculations then we need the curly braces.

```
console.log(yearsUntilRetirement(1991));
```

```
const yearsUntilRetirement = (birthYear, firstName) => {  
  const age = 2037 - birthYear;  
  const retirement = 65 - age;  
  return `${firstName} retires in ${retirement}  
years`;  
}
```

We can pass multiple parameters into an arrow function but we must wrap them in brackets and separated by a coma.

```
console.log(yearsUntilRetirement(1991, 'Jonas'));  
console.log(yearsUntilRetirement(1980, 'Bob'));
```


Functions calling other functions

```
// The fruit dicer chops fruit into four chunks
```

```
function diceFruit(fruit) {  
  return fruit * 4;  
}
```

3) The diceFruit function dices the fruit into 4 chunks

```
// The fruit processor can only make juice from diced fruit
```

```
function fruitProcessor(apples, oranges) {
```

```
  const dicedApple = diceFruit(apples);
```

2) 2 apples get diced by the external diceFruit function.

```
  const dicedOrange = diceFruit(oranges);
```

4) 5 oranges get diced by the external diceFruit function.

```
  const juice = `juice of ${dicedApple} apple chunks and ${dicedOrange} orange  
chunks.`;
```

```
  return juice;
```

5) The fruitProcessor returns juice containing many chunks of fruit.

```
}
```

```
console.log(fruitProcessor(2, 5));
```

1) We pass into the fruitProcessor 2 apples and 5 oranges.

This is a good example of DIY. Don't repeat yourself because we are dicing two fruit into 4. if we were fruitProcessing 10 items of fruit then we would still only need one diceFruit Function.

Review Functions

To convert an arrow function into a function Expression we just remove the arrow and add function before the input params.

```
const yearsUntilRetirement = function (birthYear, firstName) {  
  const age = 2037 - birthYear;  
  const retirement = 65 - age;  
  return `${firstName} retires in ${retirement} years`;  
}
```

Now the age is calculated in a separate function called calc age. We call the calcAge function from within the yearsUntilRetirement function.

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

```
const yearsUntilRetirement = function (birthYear, firstName) {  
  const age = calcAge(birthYear);  
  const retirement = 65 - age;  
  return `${firstName} retires in ${retirement} years`;  
}  
  
console.log(yearsUntilRetirement(1991, 'Jonas'));  
console.log(yearsUntilRetirement(1970, 'Bob'));
```

Now with a birth year of 1970, bob is already retired by -2 years. We can account for that with an if else statement to return either retirement variable or -1 if retirement is not greater than zero.

Note that after the return there is an explicit exit. i.e. it does not run any more of the function code.

```
const calcAge = function (birthYear) {  
  return 2037 - birthYear;  
}  
  
const yearsUntilRetirement = function (birthYear, firstName) {  
  const age = calcAge(birthYear);  
  const retirement = 65 - age;  
  if (retirement > 0) {  
    return retirement;  
  } else {  
    return -1;  
  }  
}  
  
console.log(yearsUntilRetirement(1991, 'Jonas'));  
console.log(yearsUntilRetirement(1970, 'Bob'));
```

If I now add a console log after each of the return lines will it show in the browser? No – because it is after the return which has an explicit exit so the console log line is simply ignored.

```
const yearsUntilRetirement = function (birthYear, firstName) {  
  const age = calcAge(birthYear);  
  const retirement = 65 - age;  
  if (retirement > 0) {  
    return retirement;  
    console.log(`${firstName} retires in ${retirement} years`);  
  } else {  
    return -1;  
    console.log(`${firstName} is already retired`);  
  }  
}  
  
console.log(yearsUntilRetirement(1991, 'Jonas'));  
console.log(yearsUntilRetirement(1970, 'Bob'));
```

In Visual studio code you can highlight the line and use alt and up arrow to move the highlighted line up 1. Now we see the console.log is before the return so it will be executed.

```
const calcAge = function (birthYear) {  
    return 2037 - birthYear;  
}  
  
const yearsUntilRetirement = function (birthYear, firstName) {  
    const age = calcAge(birthYear);  
    const retirement = 65 - age;  
    if (retirement > 0) {  
        console.log(`${firstName} retires in ${retirement} years`);  
        return retirement;  
    } else {  
        console.log(`${firstName} is already Retured`);  
        return -1;  
    }  
}  
  
console.log(yearsUntilRetirement(1991, 'Jonas'));  
console.log(yearsUntilRetirement(1970, 'Bob'));
```

Function Declaration, can be used before it is declared.

```
// Function Declaration
function calcAge (birthYear) {
  return 2037 - birthYear;
}
```

Function Expression, essentially a function stored inside a variable

```
// Function Expression
const calcAge = function (birthYear) {
  return 2037 - birthYear;
}
```

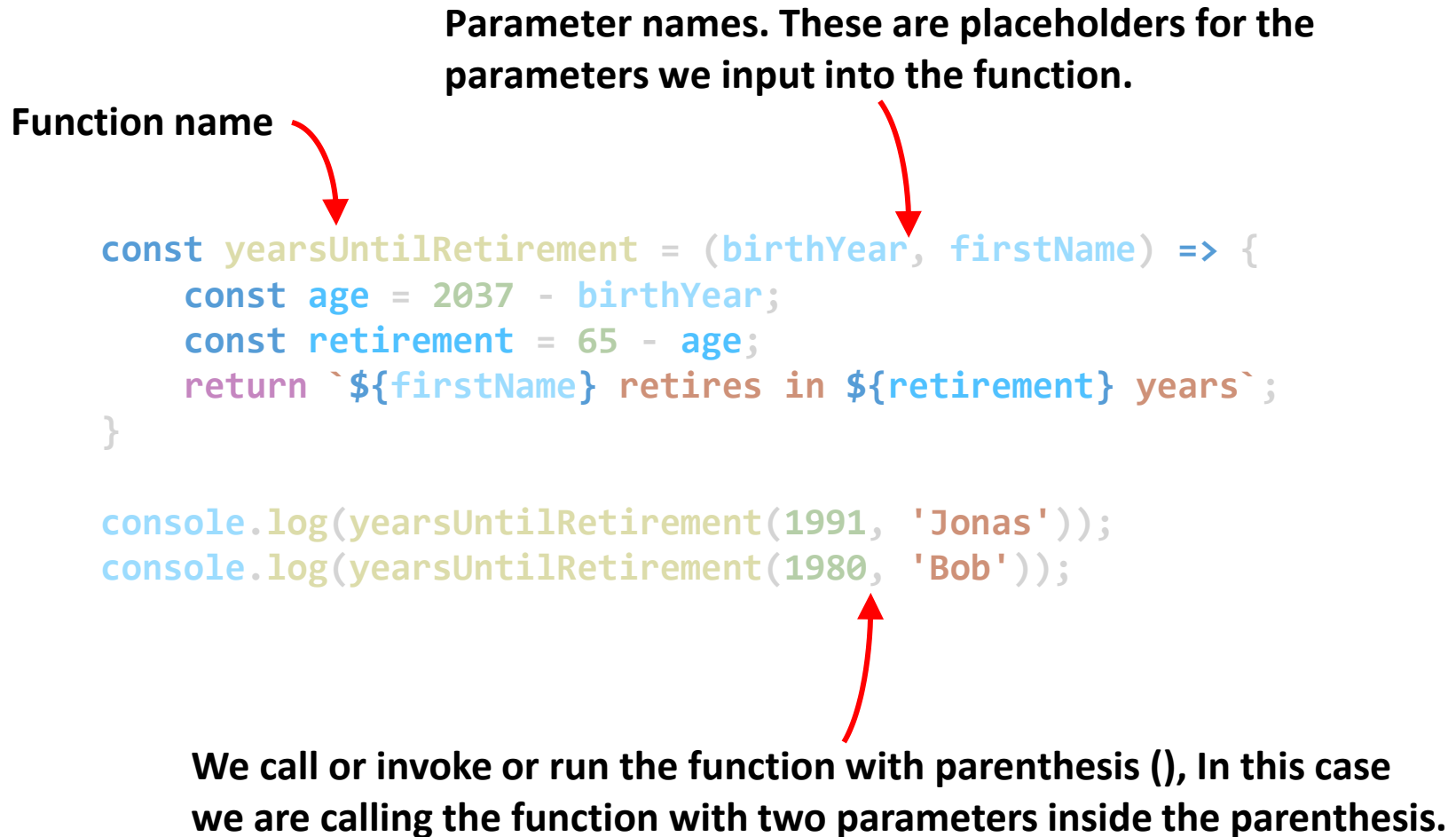
Arrow Function, good for quick one line functions. Has no This keyword. More about this later.

```
// Arrow Function
const calcAge = birthYear => 2037 - birthYear;
```

All three types of function work in the same way, they receive input data, transform the data and then output the data.

Parameter names. These are placeholders for the parameters we input into the function.

Function name



```
const yearsUntilRetirement = (birthYear, firstName) => {  
  const age = 2037 - birthYear;  
  const retirement = 65 - age;  
  return `${firstName} retires in ${retirement} years`;  
}
```

The diagram shows a function definition and two calls. A red arrow points from the text 'Function name' to the function name 'yearsUntilRetirement' in the definition. Another red arrow points from the text 'Parameter names. These are placeholders for the parameters we input into the function.' to the parameter names 'birthYear' and 'firstName' in the definition. A third red arrow points from the text 'We call or invoke or run the function with parenthesis (), In this case we are calling the function with two parameters inside the parenthesis.' to the opening parenthesis of the second function call 'yearsUntilRetirement(1980, 'Bob')'.

```
console.log(yearsUntilRetirement(1991, 'Jonas'));  
console.log(yearsUntilRetirement(1980, 'Bob'));
```

We call or invoke or run the function with parenthesis (), In this case we are calling the function with two parameters inside the parenthesis.

Introduction to Arrays

```
const friend1 = 'Michael';  
const friend2 = 'Steven';  
const friend3 = 'Peter';
```

```
// Create array using littoral syntax  
const friendArr = ['Michael', 'Steven', 'Peter'];  
console.log(friendArr);
```

```
// Create array using the inbuilt array function  
const yearsArr = new Array(1991, 1984, 2008, 2020);  
console.log(yearsArr);
```

```
// retrieve a value from within the array  
console.log(friendArr[0], yearsArr[2]);
```

```
// To determine array length  
console.log(friendArr.length);
```

```
// To get the last value from the array  
console.log(friendArr[friendArr.length-1]);
```

An array is a means to store multiple values of the same thing i.e. friends or years, rather than a variable for each one.

An array can be defined by using the littoral method or the inbuilt array function.

The position of values in an array start from zero. i.e. the first value in the array is [0], the second position in the array is [1], third position [2], etc....

Array.length will always output the length counting from 1, i.e. the quantity of values inside the array.

Because the values in the array are numbered from zero the last value in the array will be length minus 1.


```
// Mutate array by replacing Peter with Jay
friendArr[2] = 'Jay';
console.log(friendArr);
```

We can overwrite a position in an array with a new value by specifying its position number starting from zero.

```
// We cannot overwrite the contents of the entire array
friendArr = ['Bob', 'Alice'];
```

But we cannot overwrite the entire contents of the array. This is illegal.

Uncaught TypeError: Assignment to constant variable.

```
// We can populate an array with any data type. string, number etc..
const personArr = ['Jonas', 'Schmedtmann', 2037-1991];
console.log(personArr);
```

An array can be populated with more than one data type.

```
// We can insert variables into an array
const firstName = 'Jonas';
const lastname = 'Schmedtmann';
const age = 2037-1991;
const varArr = [firstName, lastname, age];
console.log(varArr);
```

Variables can even be inserted into an array.

```
// An array can be inserted into an array
const profession = 'teacher';
const multiArr = [firstName, lastname, age,
profession, friendArr];
console.log(multiArr);
```

```
(5) ['Jonas', 'Schmedtmann', 46, 'teacher', Array(3)] ⓘ
0: "Jonas"
1: "Schmedtmann"
2: 46
3: "teacher"
4: (3) ['Michael', 'Steven', 'Jay']
length: 5
[[Prototype]]: Array(0)
```

Note how the array length is 5. [0] is firstName, [1] is lastName, [2] is age, [3] is profession and [4] is the friendArr. This array contains a total of five values with the fifth value being an array.

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

```
const yearArr = [1990, 1967, 2002, 2010, 2018];
```

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

```
const age1 = calcAge(yearArr[0]);
const age2 = calcAge(yearArr[1]);
const age3 = calcAge(yearArr[yearArr.length-1]);
console.log(age1, age2, age3);
```

```
const agesArr = [calcAge(yearArr[0]),
calcAge(yearArr[2]), (yearArr[yearArr.length-1])];
console.log(agesArr);
```

If I try and run an array through a function I get a NaN error, Not a number.

But I can run individual values from the array through the function.

I can create an array based on values created from a function by calling that function from within the array.

Basic Array Operations (Methods)

Push adds an element onto the end of the array

```
const friendArr = ['Michael', 'Steven', 'Peter'];
friendArr.push('Jay');
console.log(friendArr);
```

Push is also a function which returns array length by default so we can put this in a variable.

```
const friendArr = ['Michael', 'Steven', 'Peter'];
const newLength = friendArr.push('Jay');
console.log(friendArr);
console.log(newLength);
```

Unshift method adds elements to the beginning of the array

```
friendArr.unshift('John');
console.log(friendArr);
```

Pop will remove the LAST element of the array

```
friendArr.pop();
console.log(friendArr);
```

It does not return the length of the array but returns what was popped

```
const popped = friendArr.pop();
console.log(friendArr);
console.log(popped);
```

```
▶ (4) ['Michael', 'Steven', 'Peter', 'Jay']  
>
```

```
(4) ['Michael', 'Steven', 'Peter', 'Jay'] ⓘ  
  0: "Michael"  
  1: "Steven"  
  2: "Peter"  
  3: "Jay"  
  length: 4  
  [[Prototype]]: Array(0)  
4
```

```
(5) ['John', 'Michael', 'Steven', 'Peter', 'Jay'] ⓘ  
  0: "John"  
  1: "Michael"  
  2: "Steven"  
  3: "Peter"  
  4: "Jay"  
  length: 5  
  [[Prototype]]: Array(0)
```

```
(4) ['John', 'Michael', 'Steven', 'Peter']  
(3) ['John', 'Michael', 'Steven']  
Peter
```

SHIFT Removes First element from the array

```
friendArr.shift();  
console.log(friendArr);
```

It does not return the length of the array but returns what was shifted

```
const shifted = friendArr.shift();  
console.log(friendArr);  
console.log(shifted);
```

```
(2) ['Michael', 'Steven']  
['Steven']  
Michael
```

Find index of an element in array (The position starting from zero)

```
console.log(friendArr.indexOf('Steven'));
```

If we try and look for an element which is not in the array we will get -1 as a result.

```
console.log(friendArr.indexOf('Micahel'));
```

```
0
```

```
-1
```

Modern Es6 method for indexOf is includes which returns a Boolean if element is in array

```
console.log(friendArr.includes('Steven'));  
console.log(friendArr.includes('Micahel'));
```

Includes uses strict mode so does not do type coercion

If I add 23 as a number then try and find it as a string

```
friendArr.push(23);  
console.log(friendArr.includes('23'));  
console.log(friendArr.includes(23));
```

```
true
```

```
false
```

```
false
```

```
true
```

We can use includes to write conditionals

```
if (friendArr.includes('Peter')) {  
    console.log('you have a friend called Peter');  
} else {  
    console.log('Peter is not your friend');  
}
```

Peter is not your friend

Introduction to Objects

Here we have a person array with elements in it, but we can only reference those elements by calling their position (starting from zero).

```
const personArr = [  
  'Jonas',  
  'Schmedtmann',  
  2037-1991,  
  'teacher',  
  ['Michael', 'Steven', 'Peter']  
];  
console.log(personArr);
```

```
script.js:346  
(5) ['Jonas', 'Schmedtmann', 46, 'teacher', Array(3)] ⓘ  
  0: "Jonas"  
  1: "Schmedtmann"  
  2: 46  
  3: "teacher"  
  4: (3) ['Michael', 'Steven', 'Peter']  
  length: 5  
  [[Prototype]]: Array(0)
```

But we can assign a name to each element in the object array using curly braces to make key value pairs.

```
const personObj = {  
  firstName: 'Jonas',  
  lastName: 'Schmedtmann',  
  age: 2037-1991,  
  job: 'teacher',  
  friends: ['Michael', 'Steven', 'Peter']  
}  
console.log(personObj);
```

```
script.js:358  
{firstName: 'Jonas', lastName: 'Schmedtmann', age: 46, job: 'teacher', friends: Array(3)} ⓘ  
  age: 46  
  firstName: "Jonas"  
  friends: (3) ['Michael', 'Steven', 'Peter']  
  job: "teacher"  
  lastName: "Schmedtmann"  
  [[Prototype]]: Object
```

The object of PersonObj has 5 properties - firstName, lastName etc... Using curly braces to write an object is called the object literal. Note how the object is ordered alphabetically in the console.

Dot vs Bracket notation

Retrieving a value by property name from an object using dot notation

```
console.log(personObj.firstName);
```

Jonas

Retrieving a value by property name from an object using brackets

```
console.log(personObj['lastname']);
```

Schmedtmann

Using square brackets we can concatenate

```
const nameKey = 'Name';  
console.log(personObj['first' + nameKey]);  
console.log(personObj['last' + nameKey]);
```

Jonas

Schmedtmann

But using dot notation to compute using concatenation does not work

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

Uncaught SyntaxError: Unexpected string

We can use the function prompt to define a variable of what property we want to find.

```
const tellMe = prompt('What do you want to know about personObj? Choose between  
firstName, LastName, age, job and friends');
```

Then use the bracket notation to output the contents of the object property.

```
console.log(personObj[tellMe]);
```

But what if the user enters a falsy value that is not a property?

```
const tellMe = prompt('What do you want to know about personObj? Choose between  
firstName, LastName, age, job and friends');
```

```
if (personObj[tellMe]) {  
    console.log(personObj[tellMe]);  
} else {  
    console.log('Wrong Request! Choose between firstName, LastName, age, job and  
friends');  
}
```

We can add new properties to the object using the dot notation or the bracket notation.

```
personObj.location = 'Portugal';  
personObj['twitter'] = '@jonasschmedtmann';  
console.log(personObj);
```

```
script.js:394  
{firstName: 'Jonas', LastName: 'Schmedtmann', age: 46, job: 'teacher', friends:  
Array(3), ...} ⓘ  
  age: 46  
  firstName: "Jonas"  
  friends: (3) ['Michael', 'Steven', 'Peter']  
  job: "teacher"  
  lastName: "Schmedtmann"  
  location: "Portugal"  
  twitter: "@jonasschmedtmann"  
  [[Prototype]]: Object
```


Challenge: Compute string 'Jonas has 3 friends and his best friend is called michael' using arrays.

```
const personObj = {  
  firstName: 'Jonas',  
  lastName: 'Schmedtmann',  
  age: 2037-1991,  
  job: 'teacher',  
  friends: ['Michael', 'Steven', 'Peter']  
}  
console.log(personObj);
```

Option 1

```
const FirstName = personObj.firstName;  
const bestFriend = personObj.friends[0];  
const totalFriends = personObj.friends.length;  
console.log(FirstName + ' Has ' + totalFriends + ' friends and his best friend is  
called ' + bestFriend);
```

Option 2

```
console.log(`${personObj.firstName} has ${personObj.friends.length} friends and his  
best friend is called ${personObj.friends[0]}`);
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence

In the operator precedence 'member access' has a high precedence and associativity of left to right which is why we are able to go into the friends array of the personObj.

Object Methods

Objects like arrays can hold different types of data, they can even hold arrays. They can even hold objects inside of objects. Functions are just like another type of property because they hold values.

Any function attached inside an object is called a method. But the function must be an expression.

```
const personObj = {  
  firstName: 'Jonas',  
  lastName: 'Schmedtmann',  
  birthYear: 1991,  
  job: 'teacher',  
  friends: ['Michael', 'Steven', 'Peter'],  
  hasDriversLicense: true,  
  
  calcAge: function(birthYear) {  
    return 2037 - birthYear;  
  }  
};
```

We can call the method in the object using dot or bracket notation

```
console.log(personObj.calcAge(1991));  
console.log(personObj['calcAge'](1991));
```

But we are inputting the birthYear when it is already in the object.

```
const personObj = {  
  firstName: 'Jonas',  
  lastName: 'Schmedtmann',  
  birthYear: 1991,  
  job: 'teacher',  
  friends: ['Michael', 'Steven', 'Peter'],  
  hasDriversLicense: true,  
  
  calcAge: function() {  
    console.log(this);  
    return 2037 - this.birthYear;  
  }  
};
```

We can call the method in the object using dot notation

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

The THIS keyword refers to this object so by using it we are saying console.log this object or birthyear from this object.

Instead of the this keyword we could use personObj but what if we latter rename the object to something like personObj2? Well it would not work anymore so always use this keyword!

But what if we need to call the age more than once in our code?

We would be running the method within the object multiple times which could slow down our code if the method is doing complex calculations.

Better to calculate the age once and store it as a new property so that it only gets calculated once.

```
const personObj = {  
  firstName: 'Jonas',  
  lastName: 'Schmedtmann',  
  birthYear: 1991,  
  job: 'teacher',  
  friends: ['Michael', 'Steven', 'Peter'],  
  hasDriversLicense: true,  
  
  calcAge: function() {  
    this.age = 2037 - this.birthYear;  
    return this.age;  
  }  
};
```

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

```
console.log(personObj.age);  
console.log(personObj.age);  
console.log(personObj.age);
```

Challenge: "Jonas is a 46 year old teacher and he has a drivers License OR has no drivers License"

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

  calcAge: function() {
    this.age = 2037 - this.birthYear;
    return this.age;
  },

  getSummary: function () {
    return `${this.firstName} is a ${this.calcAge()} year old ${this.job} and
    ${this.hasDriversLicense ? "has a drivers license." : "doesn't have a drivers
    license."}`;
  }
};

console.log(personObj.calcAge());

console.log(personObj.getSummary());
```

Iteration – For loop

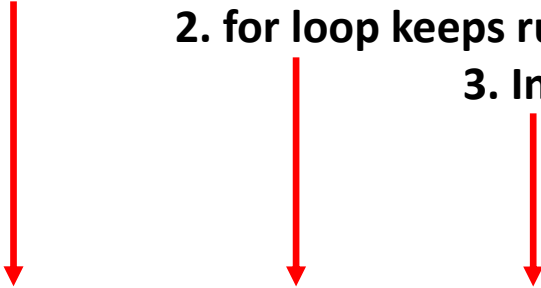
Loops are fundamental to coding. They are used for when we have to repeat something multiple times.

For loop has three components:

1. define variable and set initial value

2. for loop keeps running while condition is true

3. Increment counter



```
for (let rep = 1; rep <= 10; rep++) {  
  console.log(`lifting weights repetition ${rep}`);  
}
```

```
for (let rep = 5; rep <= 10; rep++) {  
  console.log(`swimming repetition ${rep}`);  
}
```

```
for (let rep = 5; rep <= 15; rep++) {  
  console.log(`walking repetition ${rep}`);  
}
```

lifting weights repetition 1
lifting weights repetition 2
lifting weights repetition 3
lifting weights repetition 4
lifting weights repetition 5
lifting weights repetition 6
lifting weights repetition 7
lifting weights repetition 8
lifting weights repetition 9
lifting weights repetition 10
swimming repetition 5
swimming repetition 6
swimming repetition 7
swimming repetition 8
swimming repetition 9
swimming repetition 10
walking repetition 5
walking repetition 6
walking repetition 7
walking repetition 8
walking repetition 9
walking repetition 10
walking repetition 11
walking repetition 12
walking repetition 13
walking repetition 14
walking repetition 15

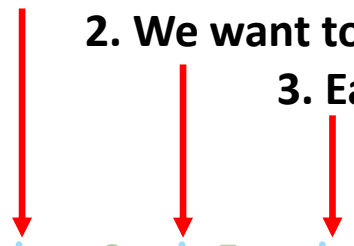
Looping arrays, breaking and Continuing

We have an array with five elements in it.

```
const personObj = [  
  'Jonas',  
  'Schmedtmann',  
  1991,  
  'teacher',  
  ['Michael', 'Steven', 'Peter']  
];
```

We want to loop through all the elements one by one.

1. We start the array at 0 because that is the first element in the array
2. We want to stop looping at 5 (because the array contains five elements)
3. Each time through the loop we increment by 1.



```
for (let i = 0; i < 5; i++) {  
  console.log(personObj[i]);  
}
```

We have an array with six elements in it.

```
const personObj2 = [  
  'Jonas',  
  'Schmedtmann',  
  1991,  
  'teacher',  
  ['Michael', 'Steven', 'Peter'],  
  true  
];
```

But because we have manually defined the condition to be less than 5 it will not iterate through element 6 in the array.

We can dynamically calculate the array length by using built in length function!

```
for (let i = 0; i < personObj2.length ; i++) {  
  console.log(personObj2[i]);  
}
```

```
for (let i = 0; i < personObj2.length ; i++) {  
  console.log(personObj2[i], typeof personObj2[i]);  
}
```

Jonas
Schmedtmann
1991
teacher
(3) ['Michael', 'Steven', 'Peter']
true
true
Jonas string
Schmedtmann string
1991 'number'
teacher string
(3) ['Michael', 'Steven', 'Peter'] 'object'
true 'boolean'


```
const personObj2 = [
  'Jonas',
  'Schmedtmann',
  1991,
  'teacher',
  ['Michael', 'Steven', 'Peter'],
  true
];
```

We can create a new empty array outside of the loop

```
const types = [];
```

```
for (let i = 0; i < personObj2.length ; i++) {
```

Here we are reading the array for each element and getting the typeof

```
  console.log(personObj2[i], typeof personObj2[i]);
```

Here we are filling the types array with typeof for each i.

```
  types[i] = typeof personObj2[i];
```

```
}
```

```
console.log(types);
```

```
Jonas string
```

```
Schmedtmann string
```

```
1991 'number'
```

```
teacher string
```

```
(3) ['Michael', 'Steven', 'Peter'] 'object'
```

```
true 'boolean'
```

```
(6) ['string', 'string', 'number', 'string', 'object', 'boolean']
```

The same operation can be done with an array push

```
const types = [];  
  
for (let i = 0; i < personObj2.length ; i++) {  
    console.log(personObj2[i], typeof personObj2[i]);  
  
    types.push(typeof personObj2[i]);  
}  
  
console.log(types);
```

We have an array of birthYears and we want to calculate ages and store it in a new array.

```
const birthYearArr = [1991, 2007, 1969, 2020];  
const ageArr = [];  
  
for (let i=0; i < birthYearArr.length; i++) {  
    ageArr.push(2037 - birthYearArr[i]);  
}  
  
console.log(ageArr);
```

Continue and break statements

Continue is used to skip an iteration in the loop

Break is used to completely stop the loop

```
const personObj2 = [  
  'Jonas',  
  'Schmedtmann',  
  1991,  
  'teacher',  
  ['Michael', 'Steven', 'Peter'],  
  true  
];  
  
for (let i=0; i<personObj2.length; i++) {  
  if (typeof personObj2[i] !== 'string') continue;  
  
  console.log(personObj2[i], typeof personObj2[i]);  
}
```

Jonas string

Schmedtmann string

teacher string

Here we use an if statement to check if type of iteration is not a string and we skip it (continue)

Continue and break statements

Continue is used to skip an iteration in the loop

Break is used to completely stop the loop

```
const personObj2 = [  
  'Jonas',  
  'Schmedtmann',  
  1991,  
  'teacher',  
  ['Michael', 'Steven', 'Peter'],  
  true  
];  
  
for (let i=0; i<personObj2.length; i++) {  
  if (typeof personObj2[i] === 'number') break;  
  
  console.log(personObj2[i], typeof personObj2[i]);  
}
```

Jonas string
Schmedtmann string

Here we use an if statement to check if type of iteration a number and we stop the loop (break) at that iteration.

Looping backwards

To loop through an array backwards we need to modify the three parts to the for loop.

START: In this case the array has indexes of 0,1,2,3,4,5 so we want to start the loop at index five which is the length of the array minus 1.

CONDITION: We want to stop the loop when it gets to zero so the condition is going to be that *i* is equal to or greater than 0.

ITERATION: Finally, we want the loop to decrement by 1 the counter each iteration.

```
const personObj2 = [  
  'Jonas',  
  'Schmedtmann',  
  1991,  
  'teacher',  
  ['Michael', 'Steven', 'Peter'],  
  true  
];  
  
for (let i=personObj2.length-1; i>=0; i--) {  
  console.log(personObj2[i], typeof personObj2[i]);  
}
```

START

CONDITION

ITERATION



true 'boolean'

Array(3) 'object'

teacher string

1991 'number'

Schmedtmann string

Jonas string

Loops within Loops

We have a routine of three exercises with each exercise being repeated (or looped 5 times)

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

  2)
  for (let rep = 1; rep < 6; rep++) {
    console.log(`===== Exercise ${exercise} weights repetition ${rep}`)
  }
} 4)
```

1) We first define the outer loop with the three parameters of STARTING, CONDITION and COUNTER.

2) Within the outer loop we define the inner loop, also with starting, condition and counter.

3) On each go around of the outer loop the inner loop is executed.

4) Pay particular attention to the placement of opening and closing brackets for the outer loop.

3)

```
===== Starting exercise 1
===== Exercise 1 Lifting weights 1
===== Exercise 1 Lifting weights 2
===== Exercise 1 Lifting weights 3
===== Exercise 1 Lifting weights 4
===== Exercise 1 Lifting weights 5
===== Starting exercise 2
===== Exercise 2 Lifting weights 1
===== Exercise 2 Lifting weights 2
===== Exercise 2 Lifting weights 3
===== Exercise 2 Lifting weights 4
===== Exercise 2 Lifting weights 5
===== Starting exercise 3
===== Exercise 3 Lifting weights 1
===== Exercise 3 Lifting weights 2
===== Exercise 3 Lifting weights 3
===== Exercise 3 Lifting weights 4
===== Exercise 3 Lifting weights 5
```

The While Loop

A for loop has 3 components: Start, Condition and counter.

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

A while loop only has one component, a condition and will run until the condition is not met. However we still need to define start of the loop and counter. We define the start of the loop outside and before the loop. The counter we increment after the loop has performed its action but within the loop curly braces.

```
let rep = 1;  
while (rep <=10) {  
    console.log(`swimming repetition ${rep}`);  
    rep ++;  
}
```

A while loop is more versatile than a for loop because it only needs a condition. It can be used in a larger variety of situations because it does not need a counter. The condition is essential to keep it running but the counter is not always needed.

DICE ROLL WHILE LOOP: the dice will continue to roll until it lands on 6.

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

while (dice !==6) {
  console.log(`You Rolled a ${dice}`);
}
```

WHILE LOOPS CAN BE DANGEROUS: the current code will crash the browser because the while loop will run forever.

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

while (dice !==6) {
  console.log(`You Rolled a ${dice}`);
  dice = Math.trunc(Math.random() * 6) + 1;
}
```

You Rolled a 2
You Rolled a 3
You Rolled a 1

We need to reassign the dice value after each iteration of the loop to prevent infinite looping.

Lets analyse what is happening. The dice is rolled and value 2 is randomly calculated. This is not a six so we reassign the dice value and roll the dice again getting 3. We randomly reassign the dice number and roll again getting 1. Next roll we land on 6 so the loop is stopped. Note that we need to reassign the dice number after each roll because we did not specify a starting value outside of the loop.


```
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...you rolled a 6');  
}
```

```
You Rolled a 3  
You Rolled a 1  
You Rolled a 3  
You Rolled a 1  
Loop is about to end...you rolled a 6
```

```
You Rolled a 5  
You Rolled a 2  
You Rolled a 3  
You Rolled a 1  
You Rolled a 4  
You Rolled a 5  
You Rolled a 3  
② You Rolled a 2  
You Rolled a 3  
② You Rolled a 1  
Loop is about to end...you rolled a 6
```

```
You Rolled a 1  
Loop is about to end...you rolled a 6  
  
You Rolled a 1  
You Rolled a 5  
You Rolled a 4  
② You Rolled a 1  
You Rolled a 4  
③ You Rolled a 5  
Loop is about to end...you rolled a 6  
-
```

```
You Rolled a 1  
④ You Rolled a 4  
You Rolled a 1  
② You Rolled a 2  
You Rolled a 4  
You Rolled a 2  
② You Rolled a 4  
You Rolled a 1  
You Rolled a 5  
You Rolled a 3  
You Rolled a 2  
You Rolled a 3  
You Rolled a 5  
You Rolled a 2  
You Rolled a 1  
You Rolled a 2  
② You Rolled a 4  
Loop is about to end...you rolled a 6
```

Now each time we run this it will randomly roll the dice each time in the while loop until a 6 is found.

DEVELOPER SKILLS & VS CODE SETUP

We can install prettier into vs code and restart it. This automatically format our code according to coding conventions.

The screenshot shows the Visual Studio Code interface with the Extensions Marketplace open. The search bar contains 'prettier'. The left sidebar lists several extensions, with 'Prettier - Code formatter' at the top. The main panel displays the details for 'Prettier - Code formatter' by Prettier, showing a download count of 20,188,494 and a 5-star rating from 321 reviews. The 'Install' button is visible. Below the main title, there are tabs for 'Details', 'Feature Contributions', and 'Changelog'. The 'Details' tab is active, showing a description of Prettier as an opinionated code formatter. On the right side, there are sections for 'Categories' (Formatters), 'Resources' (Marketplace, Repository, License), and 'Marketplace Info' (Released on 10/01/2017, Last updated 29/03/2022, Identifier esbenp.prettier-vscode). The bottom status bar shows 'main*' and '0 0 0'.

EXTENSIONS: MARKETPLACE

prettier

Prettier - Code f... 20.1M ★ 3.5
Code formatter using prettier
Prettier **Install**

Prettier ESLint 551K ★ 4
A Visual Studio Extension to form...
Rebecca Vest **Install**

Prettier Now 369K ★ 4
VS Code plugin for Prettier Misce...
Remi Marsal **Install**

Prettier - Code fo... 233K ★ 3.5
Code formatter using prettier
Simon Siefke **Install**

Prettier-Standard -... 150K ★ 5
VS Code plugin for prettier + sta...
numso **Install**

Prettier - JavaScript... 86K ★ 5
VS Code plugin for jlongster/pre...
Bastian Kistner **Install**

Prettier - JavaScript... 58K ★ 5
Fork of prettier-vscode: VS Code ...
Mathieu SCHROETER **Install**

Prettier Java 56K ★ 3
Format Java with Prettier
dotdevru **Install**

Java prettier format... 44K ★ 5

Prettier - Code formatter
Prettier | 20,188,494 | ★★★★★ (321)
Code formatter using prettier
Install ⚙️

Prettier - Code formatter
Prettier | 20,188,494 | ★★★★★ (321)
Code formatter using prettier
Install ⚙️

Details Feature Contributions Changelog

Prettier Formatter for Visual Studio Code

Prettier is an opinionated code formatter. It enforces a consistent style by parsing your code and re-printing it with its own rules that take the maximum line length into account, wrapping code when necessary.

JavaScript · TypeScript · Flow · JSX · JSON
CSS · SCSS · Less
HTML · Vue · Angular HANDLEBARS · Ember · Glimmer
GraphQL · Markdown · YAML

Categories
Formatters

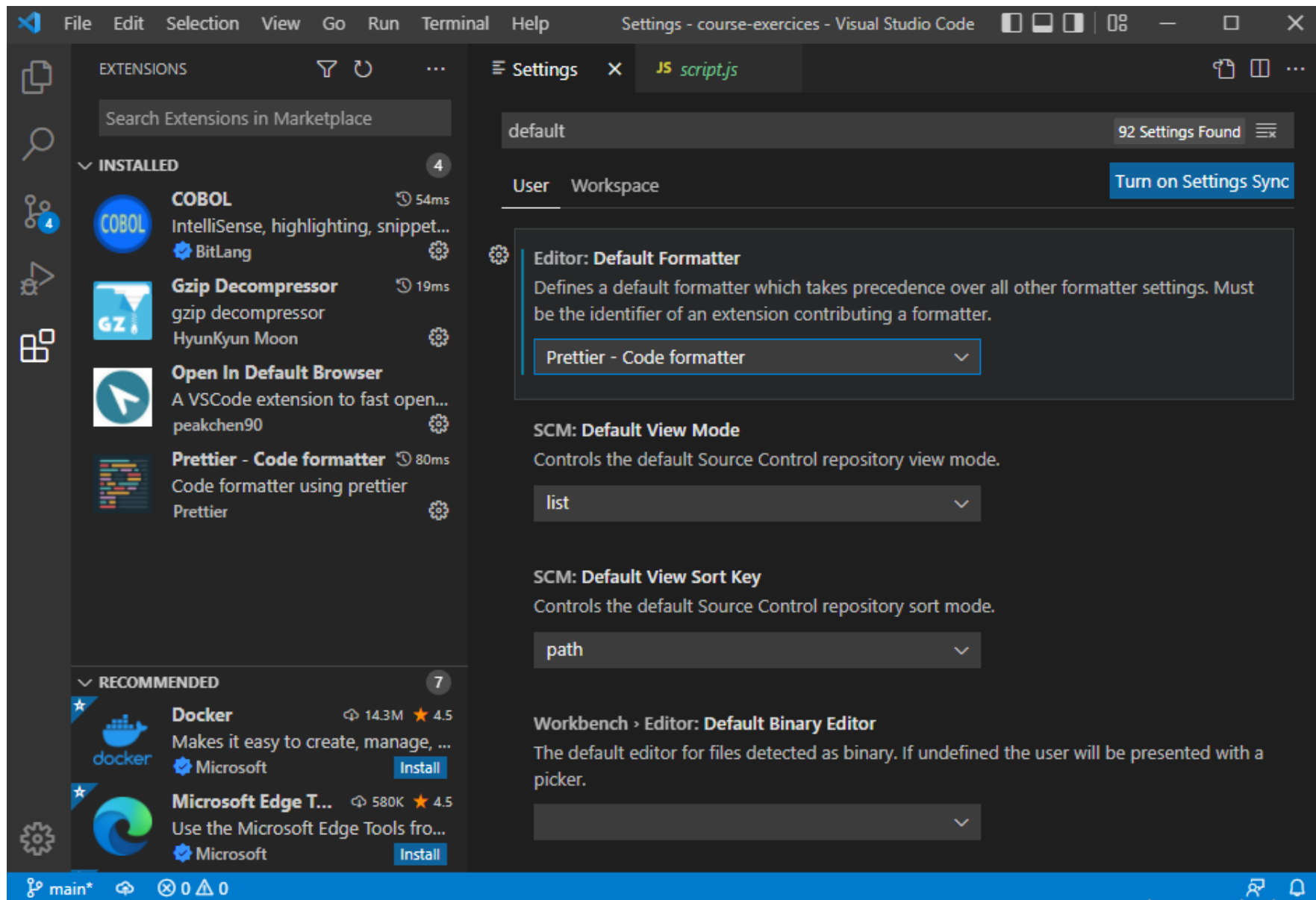
Resources
[Marketplace](#)
[Repository](#)
[License](#)

Marketplace Info
Released 10/01/2017,
on 20:52:02
Last 29/03/2022,
updated 02:25:08
Identifier esbenp.prettier-
vscode

main* 0 0 0

Activate Windows

Activate prettier as the default formatter in settings.



Install Live server. This will automatically reload the browser on code changes. Use the go live button at the bottom of the page.

The screenshot shows the Visual Studio Code interface with the Live Server extension installed. The left sidebar displays the 'EXTENSIONS' view with a search bar and a list of installed extensions. The 'Live Server' extension by Ritwick Dey is highlighted. The main editor area shows the 'Live Server' extension details page, including its version (v5.7.5), download count (20,546,430), and a five-star rating. The page also features tabs for 'Details', 'Feature Contributions', 'Changelog', and 'Runtime Status'. The 'Details' tab is active, showing a description of the extension and a link to the GitHub repository. The bottom status bar indicates the current file is 'main*' and the port is 5500.

EXTENSIONS

Search Extensions in Marketplace

INSTALLED 6

- Gzip Decompressor** 24ms
gzip decompressor
HyunKyun Moon
- Live Server** 353ms
Launch a development local Serv...
Ritwick Dey
- Open In Default Browser**
A VSCode extension to fast open...
peakchen90
- Prettier - Code formatt...** 404ms
Code formatter using prettier
Prettier
- TODO Highlight** 4ms
highlight TODOs, FIXMEs, and an...
Wayou Liu
- COBOL**
IntelliSense, highlighting, snippet...
BitLang

RECOMMENDED 7

Live Server v5.7.5
Ritwick Dey | 20,546,430 | ★★★★★
Launch a development local Server with live rel...
Disable Uninstall ⚙️
This extension is enabled globally.

Details Feature Contributions Changelog Runtime Status

[Wanna try **LIVE SERVER++ (BETA)** ? It'll enable live changes without saving file.
<https://github.com/ritwickdey/vscode-live-server-plus-plus>]

Live Server

Live Server loves ❤️ your multi-root workspace

Live Server for server side pages like PHP. [Check Here](#)

[For 'command not found error' #78]

Categories
Other

Resources
[Marketplace](#)
[Repository](#)
[License](#)

Marketplace Info
Released 27/06/2017, on 23:14:04
Last updated 08/03/2022, 12:20:23
Identifier ritwickdey.liveser

main* Port : 5500

Debugging

Identify

Becoming aware that there is a bug

During Development

Testing Software

User Reports
During production

Context: browsers, users etc.

Find

Isolating where exactly the bug is happening in the code

Developer Console
(simple Code)

Debugger
(complex code)

Fix

Correct the code

Replace wrong solution with correct solution

Prevent

Prevent it from happening again.

Searching for same bug in similar code

Writing tests using testing software.

```
///// DEBUGGING /////
```

```
// Lets say that from our previous temperature example we need to work in kelvins.
```

```
const Measkelvin = function () {  
  const measmt = {  
    type: 'temp',  
    unit: 'celcius',  
    // C - FIX  
    value: Number(prompt('Degrees celcius:')),  
    // value: prompt('Degrees celcius:')  
  };  
  // console.log(measmt);  
  console.table(measmt);  
  // console.log(measmt.value);  
  // console.warn(measmt.value);  
  // console.error(measmt.value);  
  const kelvin = measmt.value + 273;  
  return kelvin;  
};
```

```
// A) IDENTIFY
```

```
console.log(Measkelvin());
```

The console has various output options such as a plain log or table to help with debugging.

A good method is to put the variable into console table or log to see if it is an array, number, string etc.

In this example the prompt was returning 10 as a string when it needed to be a number to enter the Celsius to degrees conversion.

script.js:93

(index)	Value
type	'temp'
unit	'celcius'
value	1

Object

274 script.js:102

In the dev tools of chrome, we can go to sources and select our JS file. In the lines of code we can click in the left margin to insert a breakpoint.

The screenshot displays the Chrome DevTools interface. The browser window shows a green page titled "Developer Skills & Editor Setup". The DevTools panel is open, with the "Sources" tab selected. The file explorer on the left shows the directory structure, with "script.js" selected. The code editor displays the contents of "script.js", with line 97 highlighted and a breakpoint icon (a small circle) placed in the left margin. The code includes a function "Measkelvin" and a variable "kelvin". The right sidebar shows the "Breakpoints" section, indicating that a breakpoint is set at "script.js:97".

Developer Skills & Editor Setup

```
77
78 */
79
80 // DEBUBGING
81
82 // Lets say that from our previous temperatu
83
84 const Measkelvin = function () {
85   const measmt = {
86     type: 'temp',
87     unit: 'celcius',
88     // C - FIX
89     value: Number(prompt('Degrees celcius:'))
90     // value: prompt('Degrees celcius:')
91   };
92   // console.log(measmt);
93   console.table(measmt);
94   // console.log(measmt.value);
95   // console.warn(measmt.value);
96   // console.error(measmt.value);
97   const kelvin = measmt.value + 273;
98   return kelvin;
99 };
100
101 // A) IDENTIFY
102 console.log(Measkelvin());
103
```

Breakpoints

- script.js:97
const kelvin = measmt.va;

Scope

Not paused

Call Stack

Not paused

XHR/fetch Breakpoints

DOM Breakpoints

Global Listeners

Event Listener Breakpoints

CSP Violation Breakpoints

Line 97, Column 1 Coverage: n/a

Console Issues Search

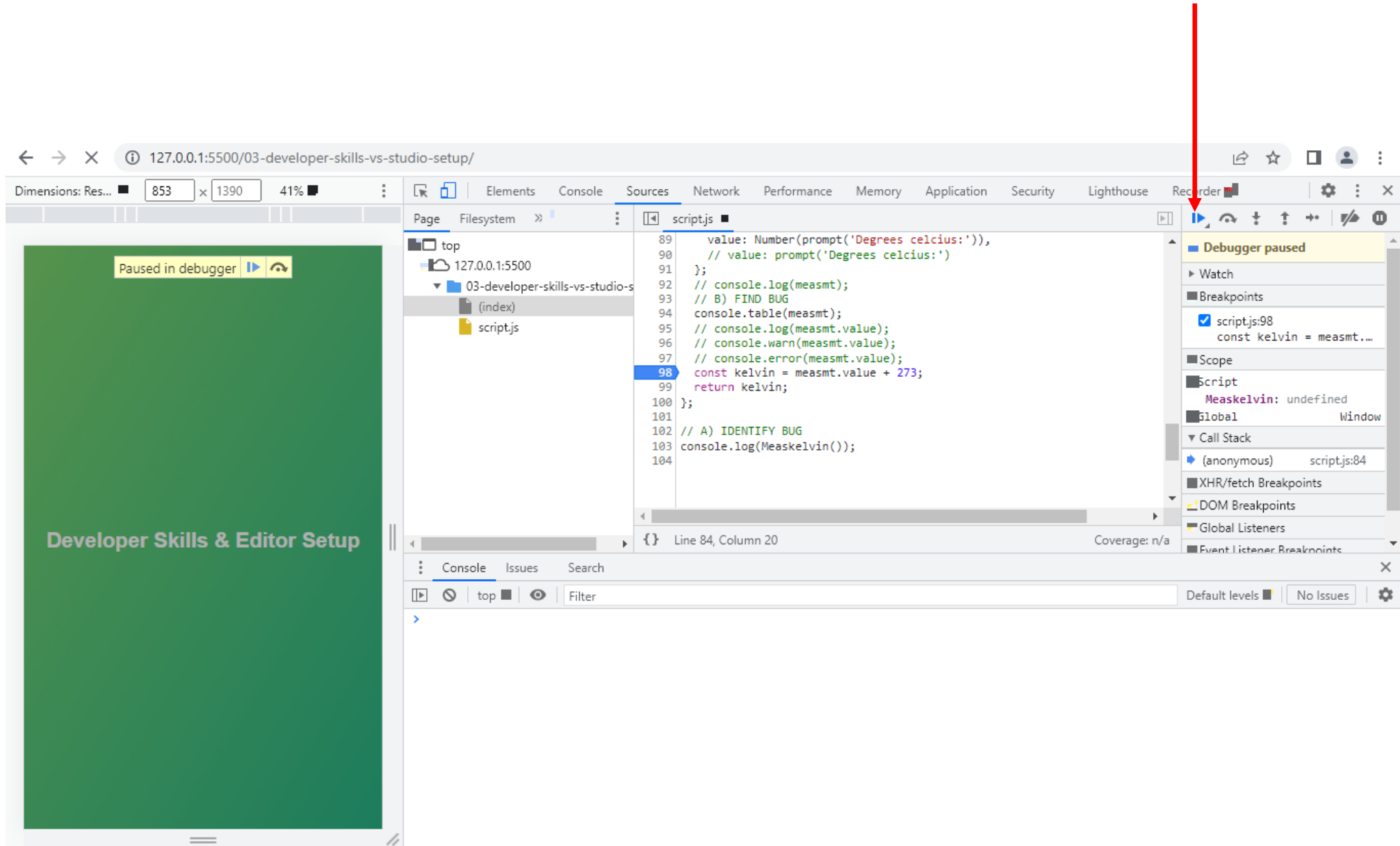
Object

283

Activate Windows
Go to Settings to activate Windows.

script.js:102

If we reload the page with the breakpoint we can see a button called debugger paused. When we press this button the code will resume execution from the beginning up to the breakpoint.



The screenshot shows a web browser window with the address bar displaying `127.0.0.1:5500/03-developer-skills-vs-studio-setup/`. The browser's developer tools are open, showing the `Sources` panel with a breakpoint set at line 98 of `script.js`. The code in the editor is as follows:

```
89 value: Number(prompt('Degrees celcius:')),
90 // value: prompt('Degrees celcius:')
91 };
92 // console.log(measmt);
93 // B) FIND BUG
94 console.table(measmt);
95 // console.log(measmt.value);
96 // console.warn(measmt.value);
97 // console.error(measmt.value);
98 const kelvin = measmt.value + 273;
99 return kelvin;
100 };
101
102 // A) IDENTIFY BUG
103 console.log(Measkelvin());
104
```

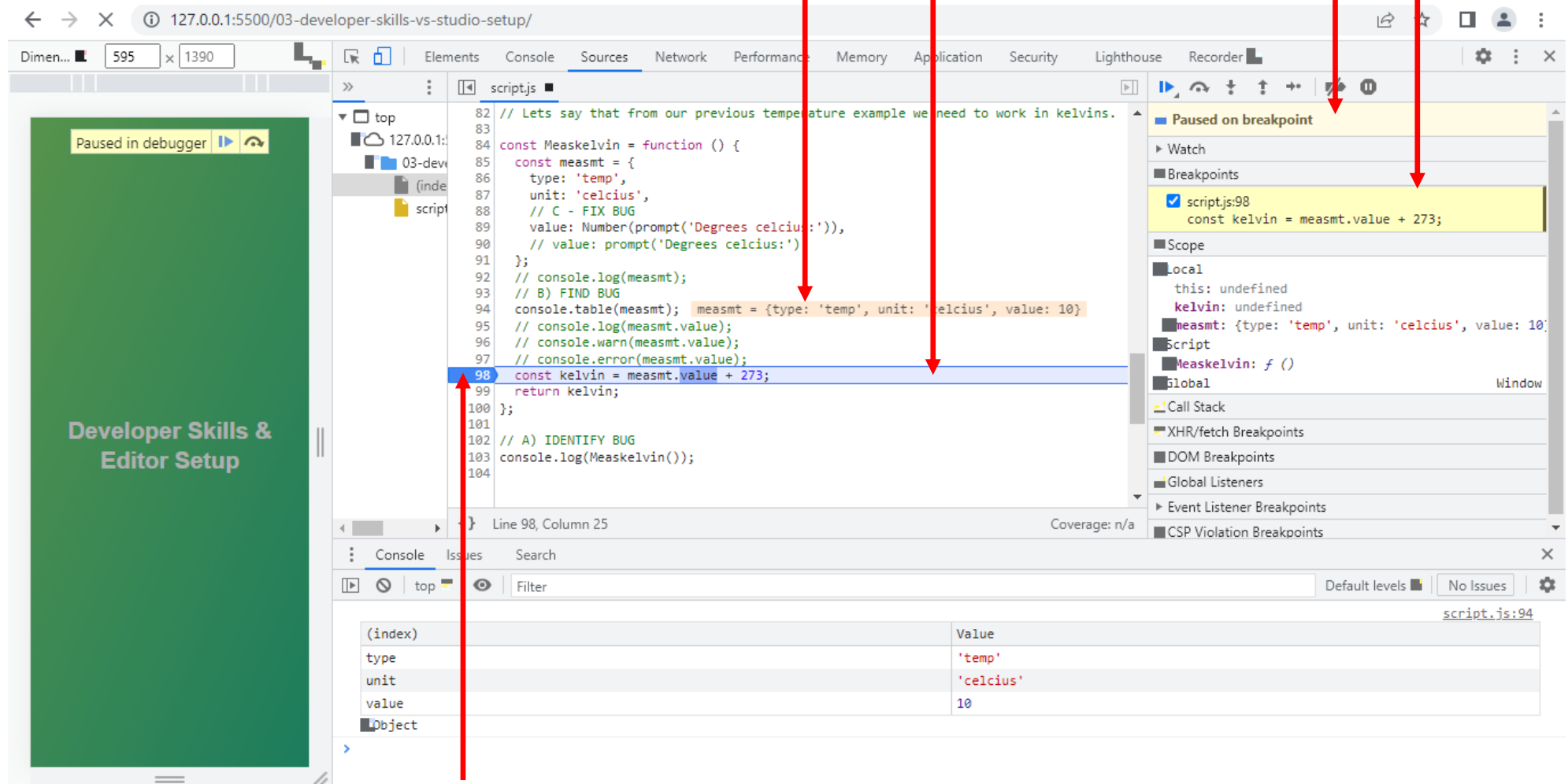
The `Debugger paused` button is highlighted in the top right corner of the browser window, indicated by a red arrow. The button is located in the top right corner of the browser window, next to the address bar. The button is labeled "Paused in debugger" and has a play icon. The browser's developer tools are open, showing the `Sources` panel with a breakpoint set at line 98 of `script.js`. The code in the editor is as follows:

```
89 value: Number(prompt('Degrees celcius:')),
90 // value: prompt('Degrees celcius:')
91 };
92 // console.log(measmt);
93 // B) FIND BUG
94 console.table(measmt);
95 // console.log(measmt.value);
96 // console.warn(measmt.value);
97 // console.error(measmt.value);
98 const kelvin = measmt.value + 273;
99 return kelvin;
100 };
101
102 // A) IDENTIFY BUG
103 console.log(Measkelvin());
104
```

The `Debugger paused` button is highlighted in the top right corner of the browser window, indicated by a red arrow. The button is located in the top right corner of the browser window, next to the address bar. The button is labeled "Paused in debugger" and has a play icon. The browser's developer tools are open, showing the `Sources` panel with a breakpoint set at line 98 of `script.js`. The code in the editor is as follows:

```
89 value: Number(prompt('Degrees celcius:')),
90 // value: prompt('Degrees celcius:')
91 };
92 // console.log(measmt);
93 // B) FIND BUG
94 console.table(measmt);
95 // console.log(measmt.value);
96 // console.warn(measmt.value);
97 // console.error(measmt.value);
98 const kelvin = measmt.value + 273;
99 return kelvin;
100 };
101
102 // A) IDENTIFY BUG
103 console.log(Measkelvin());
104
```

After pressing the resume button the code runs until it pauses on the previously set breakpoint. In the scope window the results of the last executed piece of code before the breakpoint will be shown for analysis. The script window shows the last executed line in the code before the breakpoint and the breakpoint is highlighted.



The screenshot shows a web browser with the URL `127.0.0.1:5500/03-developer-skills-vs-studio-setup/`. The developer tools are open, and the code is paused at line 98 of `script.js`. The code defines a function `Measkelvin` that takes a temperature in Celsius and returns it in Kelvin. The function is called with the value `10`, and the result is `283`. The scope window shows the current state of variables: `this` is `undefined`, `kelvin` is `undefined`, `measmt` is `{type: 'temp', unit: 'celcius', value: 10}`, and `Measkelvin` is `f ()`. The console shows the output of the function call: `(index) type unit value Object`.

Developer Skills & Editor Setup

```
82 // Lets say that from our previous temperature example we need to work in kelvins.
83
84 const Measkelvin = function () {
85   const measmt = {
86     type: 'temp',
87     unit: 'celcius',
88     // C - FIX BUG
89     value: Number(prompt('Degrees celcius:')),
90     // value: prompt('Degrees celcius:'),
91   };
92   // console.log(measmt);
93   // B) FIND BUG
94   console.table(measmt); measmt = {type: 'temp', unit: 'celcius', value: 10};
95   // console.log(measmt.value);
96   // console.warn(measmt.value);
97   // console.error(measmt.value);
98   const kelvin = measmt.value + 273;
99   return kelvin;
100 };
101
102 // A) IDENTIFY BUG
103 console.log(Measkelvin());
104
```

Paused on breakpoint

Watch

Breakpoints

- script.js:98
const kelvin = measmt.value + 273;

Scope

- local
 - this: undefined
 - kelvin: undefined
 - measmt: {type: 'temp', unit: 'celcius', value: 10}
- script
 - Measkelvin: f ()
- global
 - Window

Call Stack

- XHR/fetch Breakpoints
- DOM Breakpoints
- Global Listeners
- Event Listener Breakpoints
- CSP Violation Breakpoints

Console

(index)	Value
type	'temp'
unit	'celcius'
value	10
Object	

When finished with the debugger, don't forget to remove the breakpoint by clicking on the blue tab in the left margin.

DEBUGGING 2: Here we have changed the previous code to start the min and max at zero rather than the first values of the arrays. The temperature array values are modified also.

```
const tempArr1 = [3, 5, 1]; // const tempArr1 = [3, -2, -6, -1, 'error', 9, 13, 17];
const tempArr2 = [9, 4, 5]; // const tempArr2 = [-3, 5, 1, 5, 18, 16, 30, 35];
```

```
const calcTempAmplBug = function (tempArr1, tempArr2) {
  const tempArr = tempArr1.concat(tempArr2);
```

```
  let max = 0; // let max = tempArr[0];
  let min = 0; // let min = tempArr[0];
```

```
  for (let i = 0; i < tempArr.length; i++) {
    const curTemp = tempArr[i];
    if (typeof curTemp !== 'number') continue;
```

```
    if (curTemp > max) max = curTemp;
    if (curTemp < min) min = curTemp;
```

```
  }
```

```
  console.log(max, min);
```

```
  return max - min;
```

```
};
```

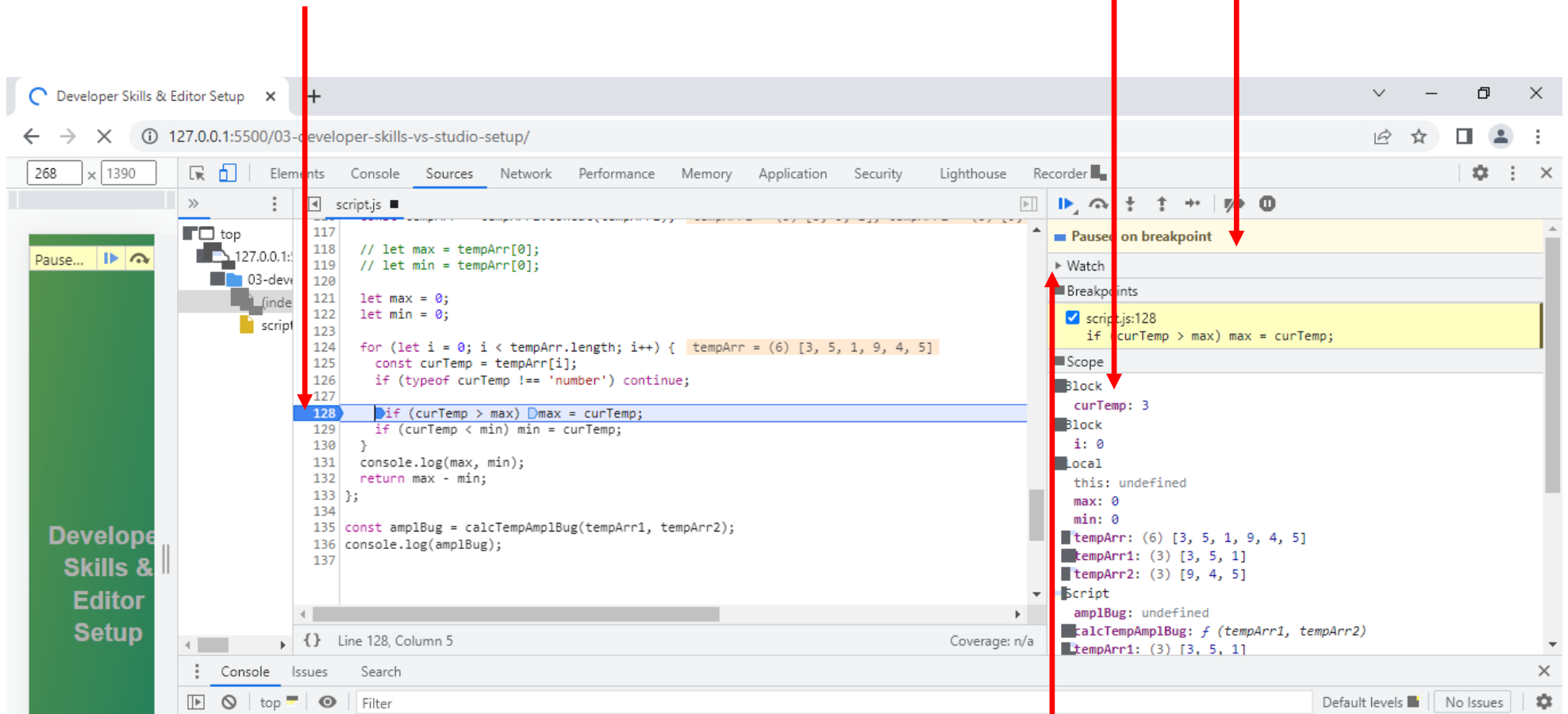
```
const amplBug = calcTempAmplBug(tempArr1, tempArr2);
console.log(amplBug);
```

This code now produces a bug with the output of max temp being 9 and min temp being 0. However looking in the arrays of values we see that the lowest temp is 1.

9 0

9

Debugger is especially useful for diagnosing problems in loops. Here I have a breakpoint set at the if statement for max temp and I can see that it has run up to this breakpoint and paused producing a curTemp of three which is the first value in the array.



When I hit the resume button the code reverts back to the beginning again running up to the breakpoint again and pausing. This time it gives curTemp of 5 which is the second value in the array. This time the max value is set to 5 because this is the highest value so far in the arrays.

If I resume the code again it starts from the beginning and runs through the loop getting a curTemp of 1. But note how the min value stays at zero. Note also The screen outputs showing the calculated values in the arrays.

The screenshot shows a web browser window with the URL `127.0.0.1:5500/03-developer-skills-vs-studio-setup/`. The browser's developer tools are open, displaying the 'Sources' panel. The file `script.js` is open, and a breakpoint is set at line 128, column 5. The code in the editor is as follows:

```
110 // const tempArr2 = [-3, 5, 1, 5, 18, 16, 30, 35];
111
112 const tempArr1 = [3, 5, 1];
113 const tempArr2 = [9, 4, 5];
114
115 const calcTempAmplBug = function (tempArr1, tempArr2) {
116   const tempArr = tempArr1.concat(tempArr2); tempArr1 = (3) [3, 5, 1], tempArr2 = (3) [9,
117
118   // let max = tempArr[0];
119   // let min = tempArr[0];
120
121   let max = 0;
122   let min = 0;
123
124   for (let i = 0; i < tempArr.length; i++) { tempArr = (6) [3, 5, 1, 9, 4, 5]
125     const curTemp = tempArr[i];
126     if (typeof curTemp !== 'number') continue;
127
128     if (curTemp > max) max = curTemp;
129     if (curTemp < min) min = curTemp;
130   }
131   console.log(max, min);
132   return max - min;
133 }
```

The breakpoint is paused at line 128. The 'Watch' panel on the right shows the state of the variables:

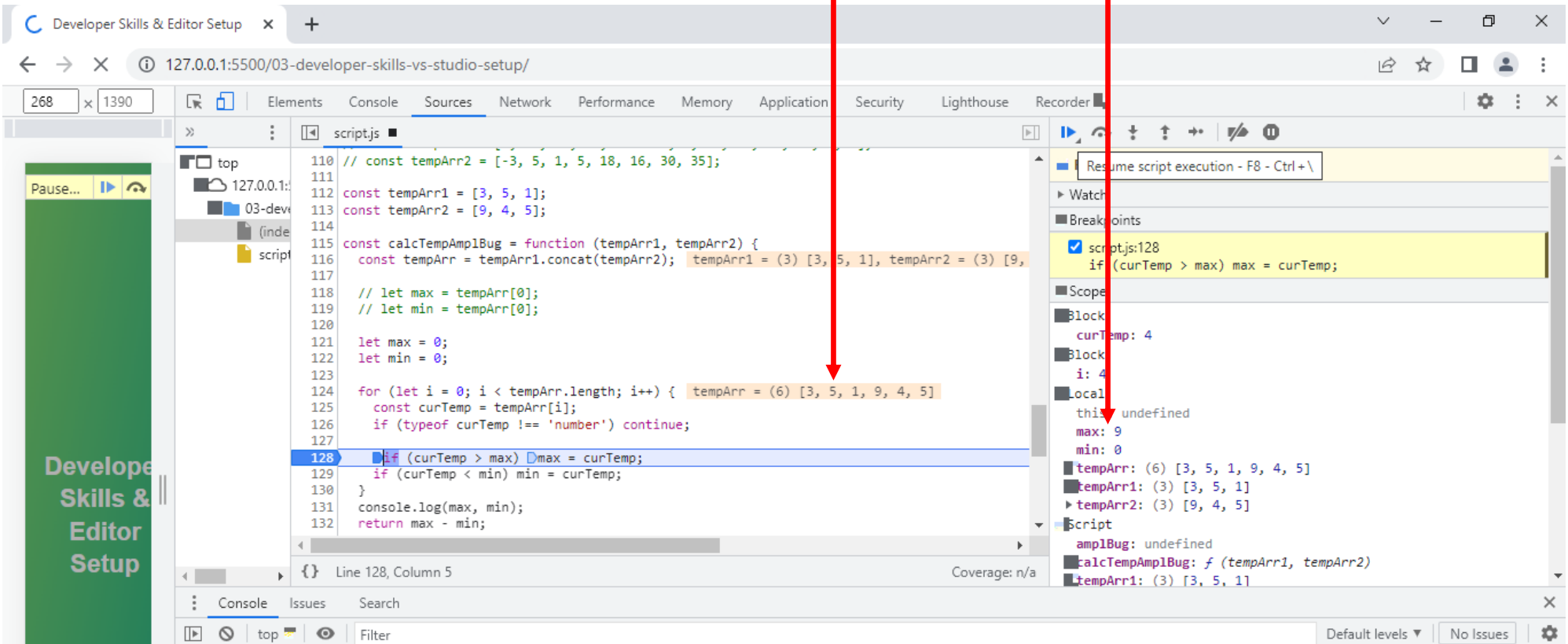
- `tempArr: (6) [3, 5, 1, 9, 4, 5]`
- `tempArr1: (3) [3, 5, 1]`
- `tempArr2: (3) [9, 4, 5]`
- `max: 5`
- `min: 0`

The 'Scope' panel on the right shows the current scope, which is the function `calcTempAmplBug`. The 'Console' panel at the bottom shows the output of the code, which is `5 0`.

Red arrows point from the text in the first block to the following elements in the image:

- Arrow 1 points to the `tempArr` variable in the code.
- Arrow 2 points to the `curTemp` variable in the code.
- Arrow 3 points to the `max` variable in the code.
- Arrow 4 points to the `min` variable in the code.
- Arrow 5 points to the `max` variable in the Watch panel.
- Arrow 6 points to the `min` variable in the Watch panel.

Two more runs through the loop and it now sets the max to 9 but the min is still set to zero despite 1 being the lowest value in the array.



Two more runs through the loop and it now sets the max to 9 but the min is still set to zero despite 1 being the lowest value in the array.

Now the breakpoint can be moved to the if statement for min. First run through the loop and it returns a curTemp of 3 and sets the max value.

The screenshot shows a web browser's developer tools interface. The main pane displays a JavaScript file named `script.js` with the following code:

```
110 // const tempArr2 = [-3, 5, 1, 5, 18, 16, 30, 35];
111 const tempArr1 = [3, 5, 1];
112 const tempArr2 = [9, 4, 5];
113
114 const calcTempAmp1Bug = function (tempArr1, tempArr2) {
115   const tempArr = tempArr1.concat(tempArr2); tempArr1 = (3) [3, 5, 1], tempArr2 = (3) [9,
116
117   // let max = tempArr[0];
118   // let min = tempArr[0];
119
120   let max = 0;
121   let min = 0;
122
123   for (let i = 0; i < tempArr.length; i++) { tempArr = (6) [3, 5, 1, 9, 4, 5]
124     const curTemp = tempArr[i];
125     if (typeof curTemp !== 'number') continue;
126     if (curTemp > max) max = curTemp; max = 3
127
128     if (curTemp < min) min = curTemp;
129   }
130   console.log(max, min);
131   return max - min;
132 }
```

A breakpoint is set at line 129, which is highlighted in blue. The right-hand pane shows the "Paused on breakpoint" state, with the following variables:

- `curTemp`: 3
- `i`: 0
- `tempArr`: (6) [3, 5, 1, 9, 4, 5]
- `tempArr1`: (3) [3, 5, 1]
- `tempArr2`: (3) [9, 4, 5]
- `max`: 3
- `min`: 0

The bottom pane shows the console with the output: `3 0`.

I can keep resuming the loop and running through it multiple times observing that the min temp stays at zero rather than being set to 1 which is the lowest value in the array so I therefore know that I need to change the `min = 0` line of code.

We can actually call the debugger direct from our code by inserting the line debugger; above the breakpoint that we want.

```
const tempArr1 = [3, 5, 1]; // const tempArr1 = [3, -2, -6, -1, 'error', 9, 13, 17];
const tempArr2 = [9, 4, 5]; // const tempArr2 = [-3, 5, 1, 5, 18, 16, 30, 35];

const calcTempAmp1Bug = function (tempArr1, tempArr2) {
  const tempArr = tempArr1.concat(tempArr2);

  let max = 0; // let max = tempArr[0];
  let min = 0; // let min = tempArr[0];

  for (let i = 0; i < tempArr.length; i++) {
    const curTemp = tempArr[i];
    if (typeof curTemp !== 'number') continue;

    debugger;
    if (curTemp > max) max = curTemp;
    if (curTemp < min) min = curTemp;
  }
  console.log(max, min);
  return max - min;
};

const amp1Bug = calcTempAmp1Bug(tempArr1, tempArr2);
console.log(amp1Bug);
```


Coding Challenge:

Given an array of forecasted maximum temperatures, the thermometer displays a string with these temperatures.

Example: [17,21,23] will print "...17°C in 1 days ...21°C in 2 days ...23°C in 3 days"

Create a function 'printForecast' which takes in an array 'arr' and logs a string like the above to the console.

Use the problem solving framework, understand the problem and break it up into sub problems.

Test data 1: [17,21,23]

Test data 2: [12, -5, 0, 4]

```
const arr1 = [17, 21, 23];
const arr2 = [12, -5, 0, 4];
```

```
// METHOD 1
```

```
const printForecast = function (arr) {
  let str = '';
  for (let i = 0; i < arr.length; i++) {
    const dayTemp = `...${arr[i]}°C in ${i + 1} days`;
    str += dayTemp;
  }
  console.log(str);
};
printForecast(arr1);
```

```
// METHOD 2
```

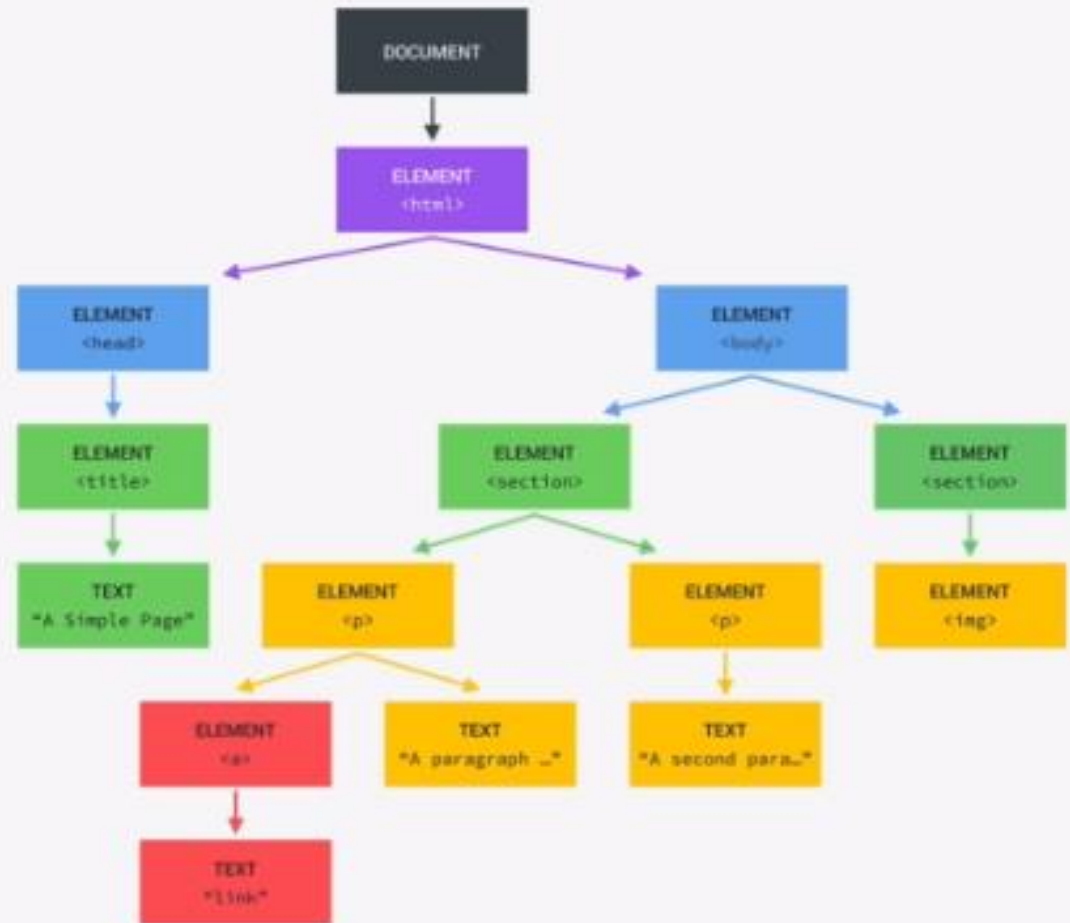
```
const printForecast2 = function (arr) {
  let str = '';
  for (let i = 0; i < arr.length; i++) {
    str += `${arr[i]}°C in ${i + 1} days ...`;
  }
  console.log('...' + str);
};
printForecast2(arr2);
```

JavaScript in the browser: DOM and Events Fundamentals – Guess My Number

What is the DOM?

We are going to make JavaScript Interact with the elements on the page using DOM manipulation. DOM stands for Document Object Model and it is basically a structured representation of the html document. The DOM can be considered a connection point between HTML and JavaScript. The DOM is represented in tree structure and is automatically created by the browser.

```
<html>
<head>
  <title>A Simple Page</title>
</head>
<body>
  <section>
    <p>A paragraph with a <a href="#">link</a></p>
    <p>A second paragraph</p>
  </section>
  <section>
    
  </section>
</body>
</html>
```



We can interact with each of the HTML elements in JavaScript.

There is a special element in the DOM called the document which is the root and the entry point into the tree structure and we need it to start selecting elements.

HTML usually has two child elements, Head and Body so they are siblings to the document.

As we move deeper into the tree we see that the relationship is child to parent so for example the head html tag has child element of title. Elements also have sibling relationships.

Whatever is in the html document also has to be in the DOM. The DOM is a complete representation of the HTML document.

DOM methods & properties for DOM manipulation. For example `document.querySelector`

DOM methods & Properties
NOT part of JavaScript



JAVASCRIPT



JavaScript is simply a dialect of ECMAScript



The DOM methods and properties are part of web APIs which the browser implements and that we can access from our JavaScript code. APIs are libraries that are written and available for us to use in the browser.

```
'use strict';
```

We can inspect a html element's text content using document.querySelector and .textContent. Note that we identify the element by it's class like in CSS, in this case .message.

```
console.log(document.querySelector('.message').textContent);
```

Using querySelector and .textContent we can manipulate the textContent of the class message.

```
document.querySelector('.message').textContent = 'Correct Number! 🎉';  
console.log(document.querySelector('.message').textContent);
```

We can also manipulate other elements of the DOM by querySelect the class and setting the textContent.

```
document.querySelector('.number').textContent = '13';  
console.log(document.querySelector('.number').textContent);
```

```
document.querySelector('.score').textContent = '10';  
console.log(document.querySelector('.score').textContent);
```

Class 'guess' is an input so what we want to querySelect is the .value of that input property. We can also manipulate value.

```
document.querySelector('.guess').value = 23;  
console.log(document.querySelector('.guess').value);
```

```
<section class="left">  
  <input type="number" class="guess" />  
  <button class="btn check">Check!</button>  
</section>
```

Now we can make our application actually do something when we click on the page. For that we can use an event listener so that JavaScript reacts to the event.

In the below code snippet we querySelect the element by its class name then add an event listener to it.

The event listener is an inbuilt function that takes two parameters, We have to specify what type of event, in this case a click. The we specify a function that we want it to execute, in this case a simple console log of the value of another element, the guess element.

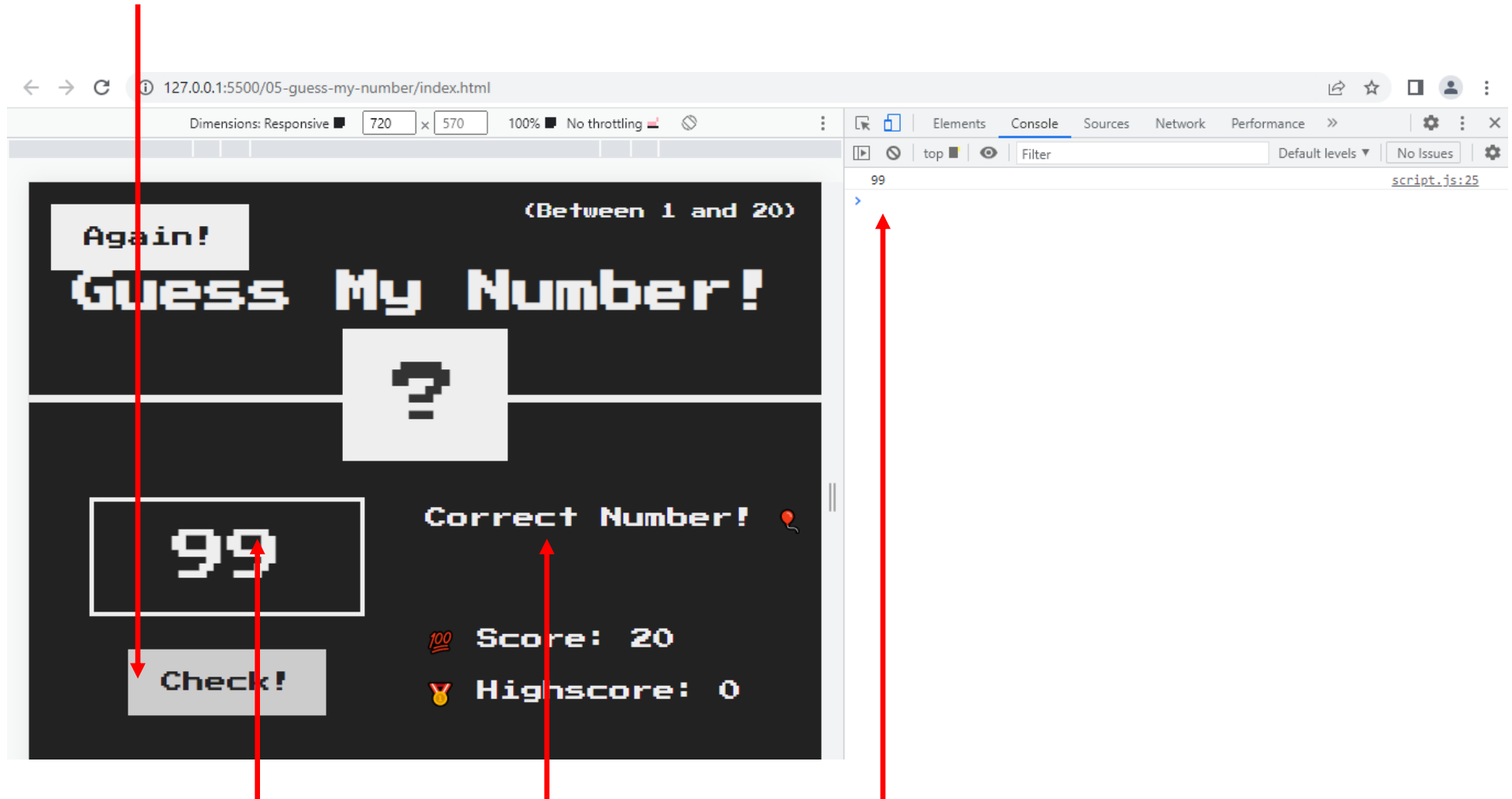
```
document.querySelector('.check').addEventListener('click', function () {  
    console.log(document.querySelector('.guess').value);  
});
```

Note that the function within the event Listener will only be executed if the 'check' element is clicked.

```
document.querySelector('.check').addEventListener('click', function () {  
    console.log(document.querySelector('.guess').value);  
  
    document.querySelector('.message').textContent = 'Correct Number! 🎉';  
});
```

We can also do some DOM manipulation within the function within the event listener, In this case we change the text Content of another element called 'message'.

1) The event listener is attached to the element with class `‘.check’`



2) Once the event listener is triggered with a click, the function within the event listener console logs the value of the input with class `‘.guess’`. The DOM manipulation of class `‘.message’` also occurs.

Now let's store the input value as a variable called guess. Note that most inputs, by default have a data type of string so we will need to convert the variable to a number for it to be useful later.

```
document.querySelector('.check').addEventListener('click', function () {  
  
  const guess = Number(document.querySelector('.guess').value);  
  console.log(guess, typeof guess);  
});
```

But what if a user has not entered a number before triggering the event listener?. Here we check for a null of the variable guess and DOM manipulate the 'message' element to say No number!

```
if (!guess) {  
  document.querySelector('.message').textContent = '⊖ No number!';  
}  
});
```



IMPLEMENT GAME LOGIC:

Define the secret random number:

Math.Random() will produce a random number between zero and 1.

To get a decimal number between 0 and 20 we can multiply by 20.

Now we can remove the decimal part by truncating the number using *Math.trunc()*.

But this will give a number between zero and 19 when what we want is a number between 1 and 20 so we can just add 1 to it.

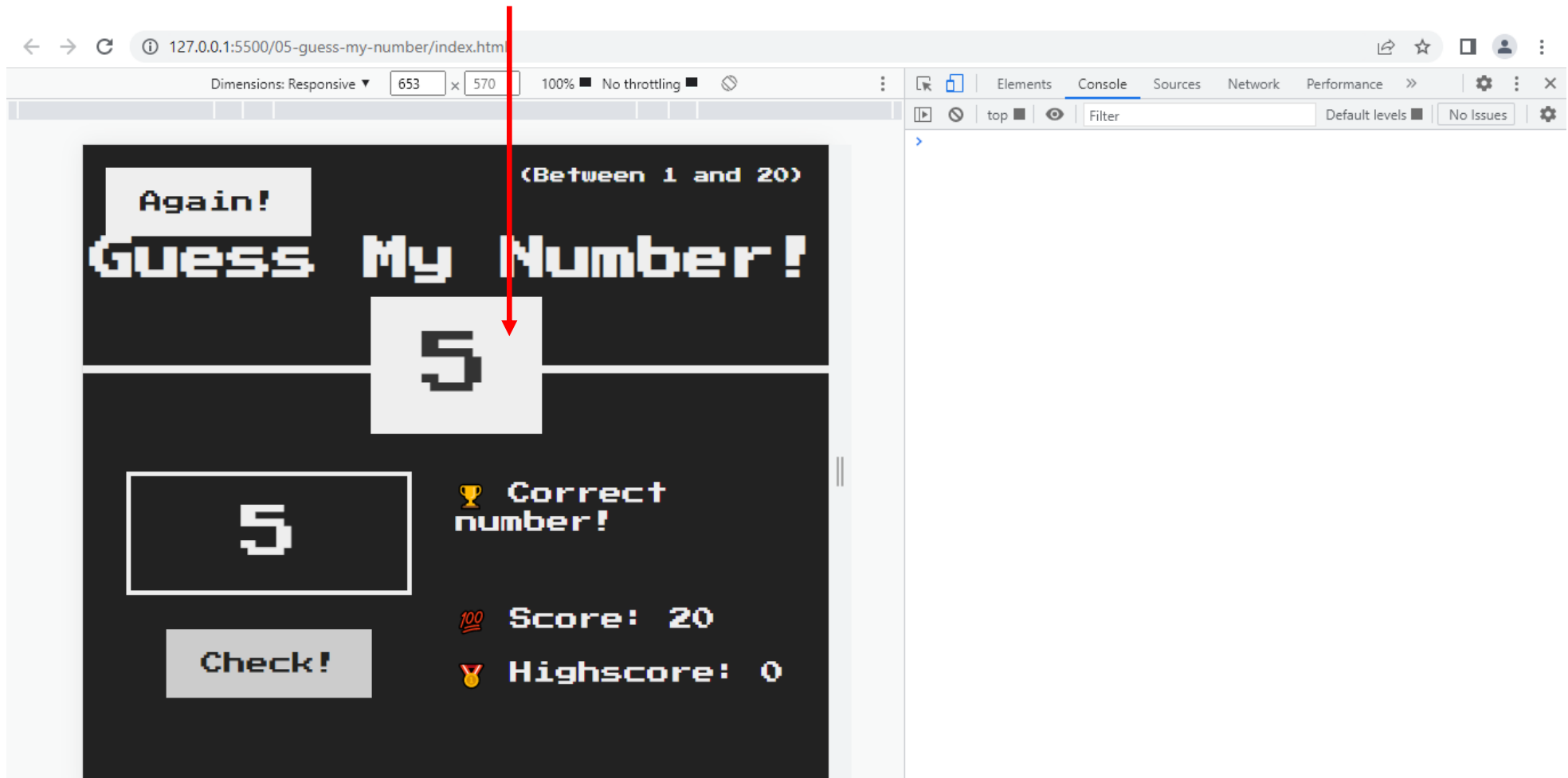
```
const secretNumber = Math.trunc(Math.random() * 20) + 1;
document.querySelector('.number').textContent = secretNumber;

document.querySelector('.check').addEventListener('click', function () {
  const guess = Number(document.querySelector('.guess').value);

  if (!guess) {
    document.querySelector('.message').textContent = '⊖ No number!';
  } else if (guess === secretNumber) {
    document.querySelector('.message').textContent = '🏆 Correct number!';
  }
});
```

If the user guesses the secret number we want a success message of 'correct number'

- 1) Here we are DOM manipulating the element with class 'number' to change the text content to variable secret number for testing purposes.



- 2) When we input 5 and trigger the event listener function with a click, The if statement first checks that variable guess is not empty/null. With the if Else we check if the guess is equal to the secret number and dom manipulate the element '.message' to say correct number if true.

Now lets handle the scenarios for when the guess is bigger or smaller than the secret number.

```
const secretNumber = Math.trunc(Math.random() * 20) + 1;
document.querySelector('.number').textContent = secretNumber;

document.querySelector('.check').addEventListener('click', function () {
  const guess = Number(document.querySelector('.guess').value);

  if (!guess) {
    document.querySelector('.message').textContent = '⊖ No number!';
  } else if (guess === secretNumber) {
    document.querySelector('.message').textContent = '🏆 Correct number!';
  } else if (guess > secretNumber) {
    document.querySelector('.message').textContent = '❌ Too high!';
  } else if (guess < secretNumber) {
    document.querySelector('.message').textContent = '❌ Too low!';
  }
});
```



Now let's implement the score where the score starts at 20 and decreases by 1 each time an incorrect guess is made. We want to recalculate the score and update the element score with the new value for an incorrect guess.

```
const secretNumber = Math.trunc(Math.random() * 20) + 1;
let score = 20;
document.querySelector('.number').textContent = secretNumber;

document.querySelector('.check').addEventListener('click', function () {
  const guess = Number(document.querySelector('.guess').value);

  if (!guess) {
    document.querySelector('.message').textContent = '⊖ No number!';
  } else if (guess === secretNumber) {
    document.querySelector('.message').textContent = '🏆 Correct number!';
  } else if (guess > secretNumber) {
    document.querySelector('.message').textContent = '✖ Too high!';
    score--;
    document.querySelector('.score').textContent = score;
  } else if (guess < secretNumber) {
    document.querySelector('.message').textContent = '✖ Too low!';
    score--;
    document.querySelector('.score').textContent = score;
  }
});
```



Each time an incorrect guess is made, the score is decreased by 1.

However, if we keep clicking the score will decrease by 1. lets say that we guess incorrectly more than 20 times then the score will be a negative number. What we need is some logic that says if score is above zero then allow guessing. Else DOM manipulate '.message' to you loose and set the score to zero.

```
const secretNumber = Math.trunc(Math.random() * 20) + 1;
let score = 20;

document.querySelector('.check').addEventListener('click', function () {
  const guess = Number(document.querySelector('.guess').value);

  if (!guess) {
    document.querySelector('.message').textContent = '⊖ No number!';
  } else if (guess === secretNumber) {
    document.querySelector('.message').textContent = '🏆 Correct number!';
  } else if (guess > secretNumber) {
    if (score > 1) {
      document.querySelector('.message').textContent = '❌ Too high!';
      score--;
      document.querySelector('.score').textContent = score;
    } else {
      document.querySelector('.message').textContent = '💩 You lost the Game!';
      document.querySelector('.score').textContent = 0;
    }
  } else if (guess < secretNumber) {
    if (score > 1) {
      document.querySelector('.message').textContent = '❌ Too low!';
      score--;
      document.querySelector('.score').textContent = score;
    } else {
      document.querySelector('.message').textContent = '💩 You lost the Game!';
      document.querySelector('.score').textContent = 0;
    }
  }
});
```

Now the game only lets us guess if the score is above zero, otherwise it will change the '.message' to you loose and set the score to zero.



Manipulating CSS styles using JavaScript.

When the player wins the game we want the background colour of the whole screen to change to green. We also want to make the element `.number` wider when the player wins and display the secret number.

```
const secretNumber = Math.trunc(Math.random() * 20) + 1;
let score = 20;

document.querySelector('.check').addEventListener('click', function () {
  const guess = Number(document.querySelector('.guess').value);

  // When there is no input
  if (!guess) {
    document.querySelector('.message').textContent = '⊖ No number!';

    // When player wins
  } else if (guess === secretNumber) {
    document.querySelector('.message').textContent = '🏆 Correct number!';
    document.querySelector('.number').textContent = secretNumber;

    document.querySelector('body').style.backgroundColor = '#60b347';

    document.querySelector('.number').style.width = '30rem';
```

Note 1: when css style property has a two word name like background-color in javascript it becomes backgroundColor. We remove the hyphen and use camel case.

Note 2: The style property value must always be a string, i.e. within quotes.

```
// When guess is too high
} else if (guess > secretNumber) {
  if (score > 1) {
    document.querySelector('.message').textContent = '❌ Too high!';
    score--;
    document.querySelector('.score').textContent = score;
  } else {
    document.querySelector('.message').textContent = '🐹 You lost the Game!';
    document.querySelector('.score').textContent = 0;
  }
}

// When guess is too low
} else if (guess < secretNumber) {
  if (score > 1) {
    document.querySelector('.message').textContent = '❌ Too low!';
    score--;
    document.querySelector('.score').textContent = score;
  } else {
    document.querySelector('.message').textContent = '🐹 You lost the Game!';
    document.querySelector('.score').textContent = 0;
  }
}
});
```

Note that when we change CSS style in JavaScript, It does not modify the CSS file but applies the changes as inline style to the html element in the DOM.

The screenshot displays a web browser window with the address bar showing '127.0.0.1:5500/05-guess-my-number/index.html'. The browser's developer tools are open, showing the 'Elements' panel on the right. The HTML structure is as follows:

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body style="background-color: rgb(96, 179, 71);">
    <header>
      <h1>Guess My Number!</h1>
      <p class="between">(Between 1 and 20)</p>
      <button class="btn again">Again!</button>
    </header>
    <main>...</main>
    <script src="script.js"></script>
  </body>
</html>
```

The 'body' element has an inline style 'background-color: rgb(96, 179, 71);' and the 'div' element with class 'number' has an inline style 'width: 30rem;'. The 'Styles' panel shows the default styles for the 'div.number' element, including 'background-color: #eee', 'color: #333', 'font-size: 6rem', and 'width: 15rem;'. The game interface on the left features a green background with white text. It includes a 'Guess My Number!' title, a '(Between 1 and 20)' hint, a '3' input field, a 'Check!' button, and a 'Correct number!' message with a trophy icon. The score is 20 and the highscore is 0.

CODING CHALLENGE 1: Implement a game reset function, so the player can make a new guess.

- 1. Select the element with the 'again' class and attach a click event handler.**
- 2. In the handler function, restore initial values of the score and SecretNumber variables.**
- 3. Restore the initial conditions of the message, number, score and guess input field.**
- 4. Also restore the original background colour (#222) and number width (15rem);**

```
let secretNumber = Math.trunc(Math.random() * 20) + 1;
let score = 20;
```

```
document.querySelector('.check').addEventListener('click', function () {
  const guess = Number(document.querySelector('.guess').value);

  // When there is no input
  if (!guess) {
    document.querySelector('.message').textContent = '⊖ No number!';

    // When player wins
  } else if (guess === secretNumber) {
    document.querySelector('.message').textContent = '🏆 Correct number!';
    document.querySelector('.number').textContent = secretNumber;
    document.querySelector('body').style.backgroundColor = '#60b347';
    document.querySelector('.number').style.width = '30rem';
  }
});
```

```

    // When guess is too high
  } else if (guess > secretNumber) {
    if (score > 1) {
      document.querySelector('.message').textContent = '❌ Too high!';
      score--;
      document.querySelector('.score').textContent = score;
    } else {
      document.querySelector('.message').textContent = '🐹 You lost the Game!';
      document.querySelector('.score').textContent = 0;
    }
  }

  // When guess is too low
  } else if (guess < secretNumber) {
    if (score > 1) {
      document.querySelector('.message').textContent = '❌ Too low!';
      score--;
      document.querySelector('.score').textContent = score;
    } else {
      document.querySelector('.message').textContent = '🐹 You lost the Game!';
      document.querySelector('.score').textContent = 0;
    }
  }
}
});

document.querySelector('.again').addEventListener('click', function () {
  secretNumber = Math.trunc(Math.random() * 20) + 1;
  let score = 20;

```

```
document.querySelector('.again').addEventListener('click', function () {
  secretNumber = Math.trunc(Math.random() * 20) + 1;
  score = 20;

  document.querySelector('.message').textContent = 'Start guessing...';
  document.querySelector('.score').textContent = score;
  document.querySelector('.number').textContent = '?';
  document.querySelector('.guess').value = '';

  document.querySelector('body').style.backgroundColor = '#222';
  document.querySelector('.number').style.width = '15rem';
});
```



IMPLEMENT HIGH SCORE FUNCTION: The highscore value is the highest score achieved over multiple games. i.e. first game score 10, second game score 15, third game score 17. the highscore would be 17.

```
let secretNumber = Math.trunc(Math.random() * 20) + 1;
let score = 20;
let highscore = 0;

document.querySelector('.check').addEventListener('click', function () {
  const guess = Number(document.querySelector('.guess').value);

  // When there is no input
  if (!guess) {
    document.querySelector('.message').textContent = '⊖ No number!';

    // When player wins
  } else if (guess === secretNumber) {
    document.querySelector('.message').textContent = '🏆 Correct number!';
    document.querySelector('.number').textContent = secretNumber;
    document.querySelector('body').style.backgroundColor = '#60b347';
    document.querySelector('.number').style.width = '30rem';

    if (score > highscore) {
      highscore = score;
      document.querySelector('.highscore').textContent = highscore;
    }
  }
});
```

```
// When guess is too high
} else if (guess > secretNumber) {
  if (score > 1) {
    document.querySelector('.message').textContent = '❌ Too high!';
    score--;
    document.querySelector('.score').textContent = score;
  } else {
    document.querySelector('.message').textContent = '🐼 You lost the Game!';
    document.querySelector('.score').textContent = 0;
  }
}

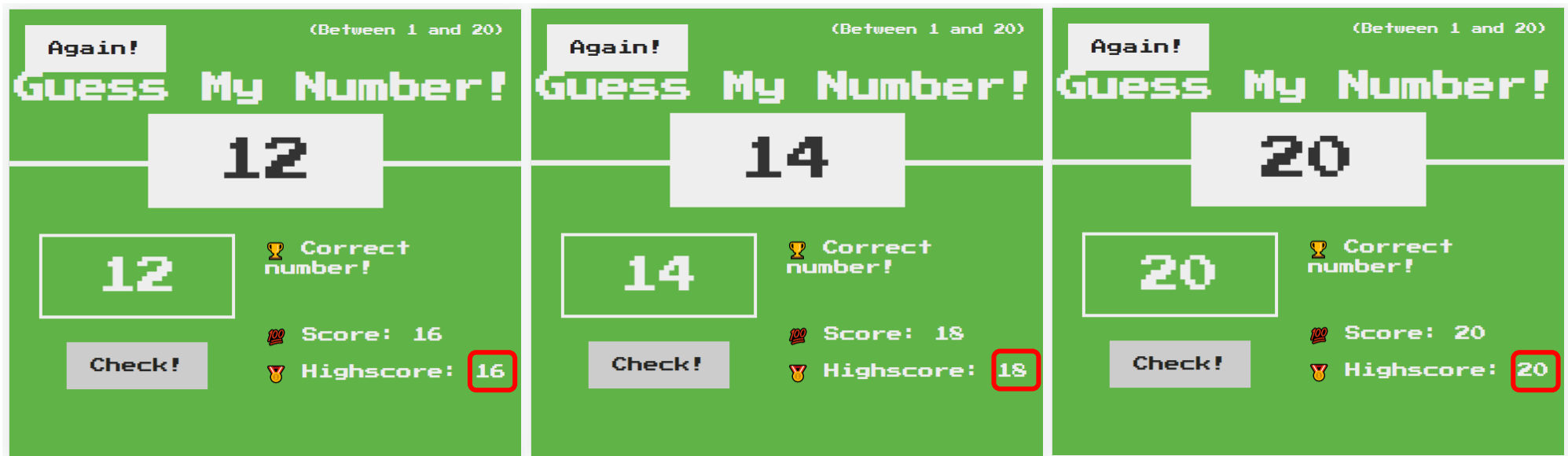
// When guess is too low
} else if (guess < secretNumber) {
  if (score > 1) {
    document.querySelector('.message').textContent = '❌ Too low!';
    score--;
    document.querySelector('.score').textContent = score;
  } else {
    document.querySelector('.message').textContent = '🐼 You lost the Game!';
    document.querySelector('.score').textContent = 0;
  }
}
});
```



```
document.querySelector('.again').addEventListener('click', function () {
  secretNumber = Math.trunc(Math.random() * 20) + 1;
  score = 20;

  document.querySelector('.message').textContent = 'Start guessing...';
  document.querySelector('.score').textContent = score;
  document.querySelector('.number').textContent = '?';
  document.querySelector('.guess').value = '';

  document.querySelector('body').style.backgroundColor = '#222';
  document.querySelector('.number').style.width = '15rem';
});
```



Now the highscore changes if in subsequent rounds the player scores more than before.

Refactoring out code: The dry principle

Looking over the code we have sections that are almost duplicate code. We can tidy this up by reducing duplicate code. The advantage of this is that if we change something like an element class then we would have to change it in multiple places in the code and it can become a nightmare of bugs. We do refactoring to eliminate duplicate code and clean it up.

```
// When guess is too high
} else if (guess > secretNumber) {
  if (score > 1) {
    document.querySelector('.message').textContent = '❌ Too high!';
    score--;
    document.querySelector('.score').textContent = score;
  } else {
    document.querySelector('.message').textContent = '🙌 You lost the Game!';
    document.querySelector('.score').textContent = 0;
  }

// When guess is too low
} else if (guess < secretNumber) {
  if (score > 1) {
    document.querySelector('.message').textContent = '❌ Too low!';
    score--;
    document.querySelector('.score').textContent = score;
  } else {
    document.querySelector('.message').textContent = '🙌 You lost the Game!';
    document.querySelector('.score').textContent = 0;
  }
}
});
```

Step 1 is to identify duplicate or almost identical code.

If the guess is too high or the guess is too low are basically the same code. What we can really refactor this to is if the guess is not equal to secretNumber.

This part of the code
stays the same

```
let secretNumber = Math.trunc(Math.random() * 20) + 1;
let score = 20;
let highscore = 0;

document.querySelector('.check').addEventListener('click', function () {
  const guess = Number(document.querySelector('.guess').value);

  // When there is no input
  if (!guess) {
    document.querySelector('.message').textContent = '⊖ No number!';

    // When player wins
  } else if (guess === secretNumber) {
    document.querySelector('.message').textContent = '🏆 Correct number!';
    document.querySelector('.number').textContent = secretNumber;
    document.querySelector('body').style.backgroundColor = '#60b347';
    document.querySelector('.number').style.width = '30rem';

    if (score > highscore) {
      highscore = score;
      document.querySelector('.highscore').textContent = highscore;
    }
  }
});
```

```

// When guess is not equal to secretNumber
} else if (guess !== secretNumber) {
  if (score > 1) {
    document.querySelector('.message').textContent =
      guess > secretNumber ? '❌ Too high!' : '❌ Too low!';
    score--;
    document.querySelector('.score').textContent = score;
  } else {
    document.querySelector('.message').textContent = '🎉 You lost the Game!';
    document.querySelector('.score').textContent = 0;
  }
}
});

```

Here we can reduce the guess not equal to secretNumber to a ternary operator to change the message. The rest of the code is the same.

```

document.querySelector('.again').addEventListener('click', function () {
  secretNumber = Math.trunc(Math.random() * 20) + 1;
  score = 20;

  document.querySelector('.message').textContent = 'Start guessing...';
  document.querySelector('.score').textContent = score;
  document.querySelector('.number').textContent = '?';
  document.querySelector('.guess').value = '';

  document.querySelector('body').style.backgroundColor = '#222';
  document.querySelector('.number').style.width = '15rem';
});

```

```
let secretNumber = Math.trunc(Math.random() * 20) + 1;
let score = 20;
let highscore = 0;
```

```
const displayMessage = function (message) {
  document.querySelector('.message').textContent = message;
};
```

Step 2 is to identify duplicate or almost identical lines of code and consider putting it in a function and call that function multiple times when we need it.

```
document.querySelector('.check').addEventListener('click', function () {
  const guess = Number(document.querySelector('.guess').value);
```

```
// When there is no input
```

```
if (!guess) {
```

```
  displayMessage('☹ No number!');
```

Call the function

```
// When player wins
```

```
} else if (guess === secretNumber) {
```

```
  displayMessage('🏆 Correct number!');
```

Call the function

```
  document.querySelector('.number').textContent = secretNumber;
```

```
  document.querySelector('body').style.backgroundColor = '#60b347';
```

```
  document.querySelector('.number').style.width = '30rem';
```

```
  if (score > highscore) {
```

```
    highscore = score;
```

```
    document.querySelector('.highscore').textContent = highscore;
```

```
  }
```

```

if (score > highscore) {
    highscore = score;
    document.querySelector('.highscore').textContent = highscore;
}

// When guess is not equal to secretNumber
} else if (guess !== secretNumber) {
    if (score > 1) {
        displayMessage(guess > secretNumber ? '❌ Too high!' : '❌ Too low!');
        score--;
        document.querySelector('.score').textContent = score;
    } else {
        displayMessage('🙌 You lost the Game!');
        document.querySelector('.score').textContent = 0;
    }
}
});

document.querySelector('.again').addEventListener('click', function () {
    secretNumber = Math.trunc(Math.random() * 20) + 1;
    score = 20;
    displayMessage('Start guessing...');
    document.querySelector('.score').textContent = score;
    document.querySelector('.number').textContent = '?';
    document.querySelector('.guess').value = '';

    document.querySelector('body').style.backgroundColor = '#222';
    document.querySelector('.number').style.width = '15rem';
});

```

Call the
function

Call the function

Call the function

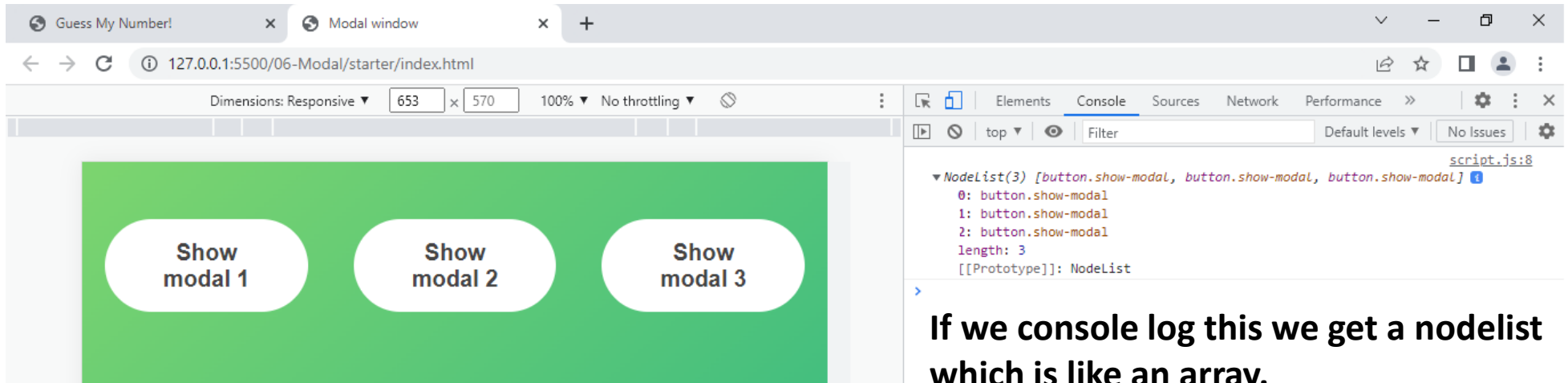
JavaScript in the browser: DOM and Events Fundamentals – Pop up Modal Window

Project: Modal Window.

We first start by defining all the elements we want to manipulate as variables. Note how in the HTML we have three modal buttons so when selecting multiple elements by class we need to use `querySelectorAll`.

```
'use strict';
```

```
const modal = document.querySelector('.modal');  
const overlay = document.querySelector('.overlay');  
const btnCloseModal = document.querySelector('.close-modal');  
const btnsOpenModel = document.querySelectorAll('.show-modal');  
  
console.log(btnsOpenModel);
```




```

'use strict';

const modal = document.querySelector('.modal');
const overlay = document.querySelector('.overlay');
const btnCloseModal = document.querySelector('.close-modal');
const btnsOpenModel = document.querySelectorAll('.show-modal');

console.log(btnsOpenModel);

for (let i = 0; i < btnsOpenModel.length; i++)
  btnsOpenModel[i].addEventListener('click', function () {
    console.log('button clicked');
    modal.classList.remove('hidden');
    overlay.classList.remove('hidden');
  });

btnCloseModal.addEventListener('click', function () {
  modal.classList.add('hidden');
  overlay.classList.add('hidden');
});

overlay.addEventListener('click', function () {
  modal.classList.add('hidden');
  overlay.classList.add('hidden');
});

```

We create a for loop to decide what happens to each of the buttons.

And then add an event listener for clicks of the button. Within the event listener we want to remove the hidden class for two elements.

The modal can be closed by either clicking on the overlay outside of the modal or using the close button in the modal.

Refine the close modal function with a named function.

```
const modal = document.querySelector('.modal');
const overlay = document.querySelector('.overlay');
const btnCloseModal = document.querySelector('.close-modal');
const btnsOpenModel = document.querySelectorAll('.show-modal');

console.log(btnsOpenModel);

for (let i = 0; i < btnsOpenModel.length; i++)
  btnsOpenModel[i].addEventListener('click', function () {
    console.log('button clicked');
    modal.classList.remove('hidden');
    overlay.classList.remove('hidden');
  });
```

```
const closeModal = function () {
  modal.classList.add('hidden');
  overlay.classList.add('hidden');
};
```

Named Function

```
btnCloseModal.addEventListener('click', closeModal);
overlay.addEventListener('click', closeModal);
```

Call Named Function in Event Handler

Refine the open modal function with a named function.

```
const modal = document.querySelector('.modal');
const overlay = document.querySelector('.overlay');
const btnCloseModal = document.querySelector('.close-modal');
const btnsOpenModel = document.querySelectorAll('.show-modal');
```

```
const openModal = function () {
  console.log('button clicked');
  modal.classList.remove('hidden'); Named Function
  overlay.classList.remove('hidden');
};
```

```
const closeModal = function () {
  modal.classList.add('hidden');
  overlay.classList.add('hidden');
};
```

```
for (let i = 0; i < btnsOpenModel.length; i++)
  btnsOpenModel[i].addEventListener('click', openModal);
```

Call Named Function in Event Handler

```
btnCloseModal.addEventListener('click', closeModal);
overlay.addEventListener('click', closeModal);
```

Key press events are global so they normally happen on the whole document and have three varieties: Keyup, Keydown, Keypress.

Keydown: is when we strike a key

keyup: is when we lift off a key

Keypress: Depreciated - event is fired when a key that produces a character value is pressed down

```
document.addEventListener('keydown', function (e) {  
  console.log('A key was pressed');  
  console.log(e);  
});
```

The event listener for a key press event stores the key value in the function so we can call it e for event and console log it

```

KeyboardEvent {isTrusted: true, key: 'Escape', code: 'Escape', location: 0, ctrl
  LKey: false, ...} ⓘ
  isTrusted: true
  altKey: false
  bubbles: true
  cancelBubble: false
  cancelable: true
  charCode: 0
  code: "Escape"
  composed: true
  ctrlKey: false
  currentTarget: null
  defaultPrevented: false
  detail: 0
  eventPhase: 0
  isComposing: false
  key: "Escape"
  keyCode: 27
  location: 0
  metaKey: false

```

Here we see a property of key which holds the value of “escape” because I pressed the escape key

We want to execute the close modal function if the modal is visible, i.e. does not contain the class of ‘hidden’ and if the e property of “key” is Escape.

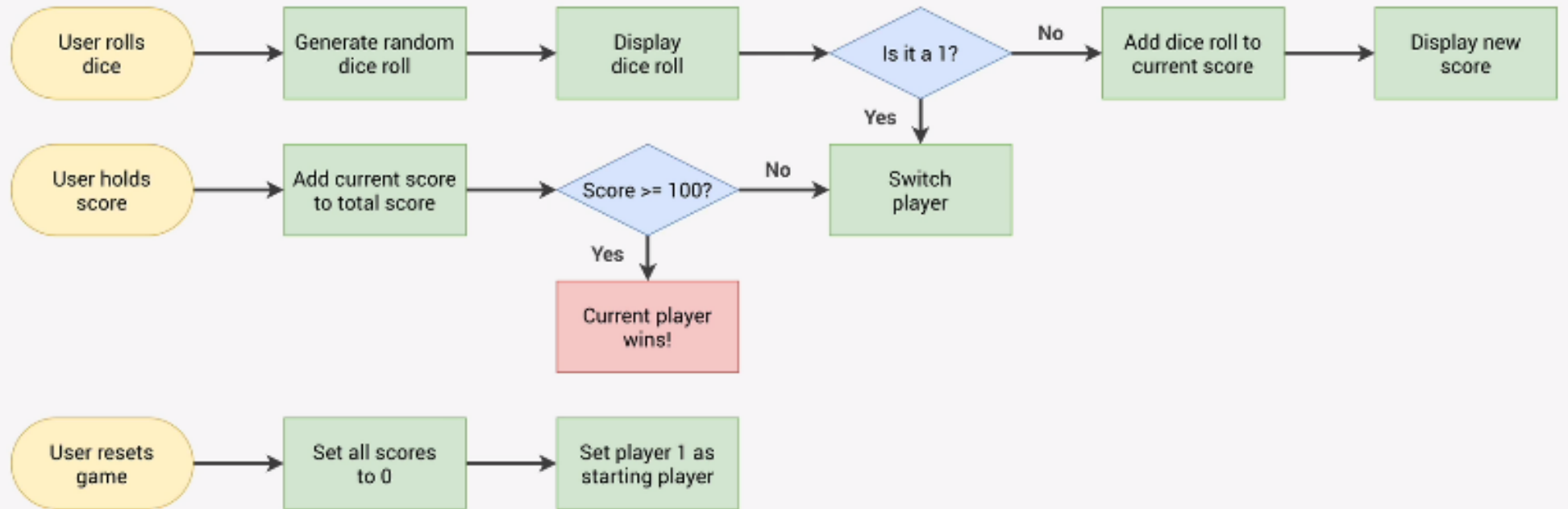
```

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

```

JavaScript in the browser: DOM and Events Fundamentals – Pig Game

Project: Pig Game.



```
'use strict';
```

```
// Selecting elements
```

```
const score0El = document.querySelector('#score--0');  
const score1El = document.querySelector('#score--1');  
const diceEl = document.querySelector('.dice');
```

```
const btnNew = document.querySelector('.btn--new');  
const btnRoll = document.querySelector('.btn--roll');  
const btnHold = document.querySelector('.btn--hold');
```

```
// Setting initial values for elements
```

```
score0El.textContent = 0;  
score1El.textContent = 0;  
diceEl.classList.add('hidden');
```

```
// Rolling dice functionality
```

```
btnRoll.addEventListener('click' function() {  
  // 1. Generate random dice roll number
```

```
  // 2. Display dice
```

```
  // 3. Check for rolled 1: if true, switch to other player
```

```
})
```

First we define the variables that we are going to use.

Then we define some initial values for the elements

And we can make a start on the roll dice functionality by first defining what we want it to do.


```
// Selecting elements
```

```
const score0El = document.querySelector('#score--0');  
const score1El = document.querySelector('#score--1');  
const current0El = document.querySelector('#current--0');  
const current1El = document.querySelector('#current--1');  
const diceEl = document.querySelector('.dice');
```

```
const btnNew = document.querySelector('.btn--new');  
const btnRoll = document.querySelector('.btn--roll');  
const btnHold = document.querySelector('.btn--hold');
```

```
const Player0 = document.querySelector('#name--0');  
const Player1 = document.querySelector('#name--1');
```

```
// Starting conditions
```

```
score0El.textContent = 0;  
score1El.textContent = 0;  
diceEl.classList.add('hidden');
```

```
let currentScore = 0;
```

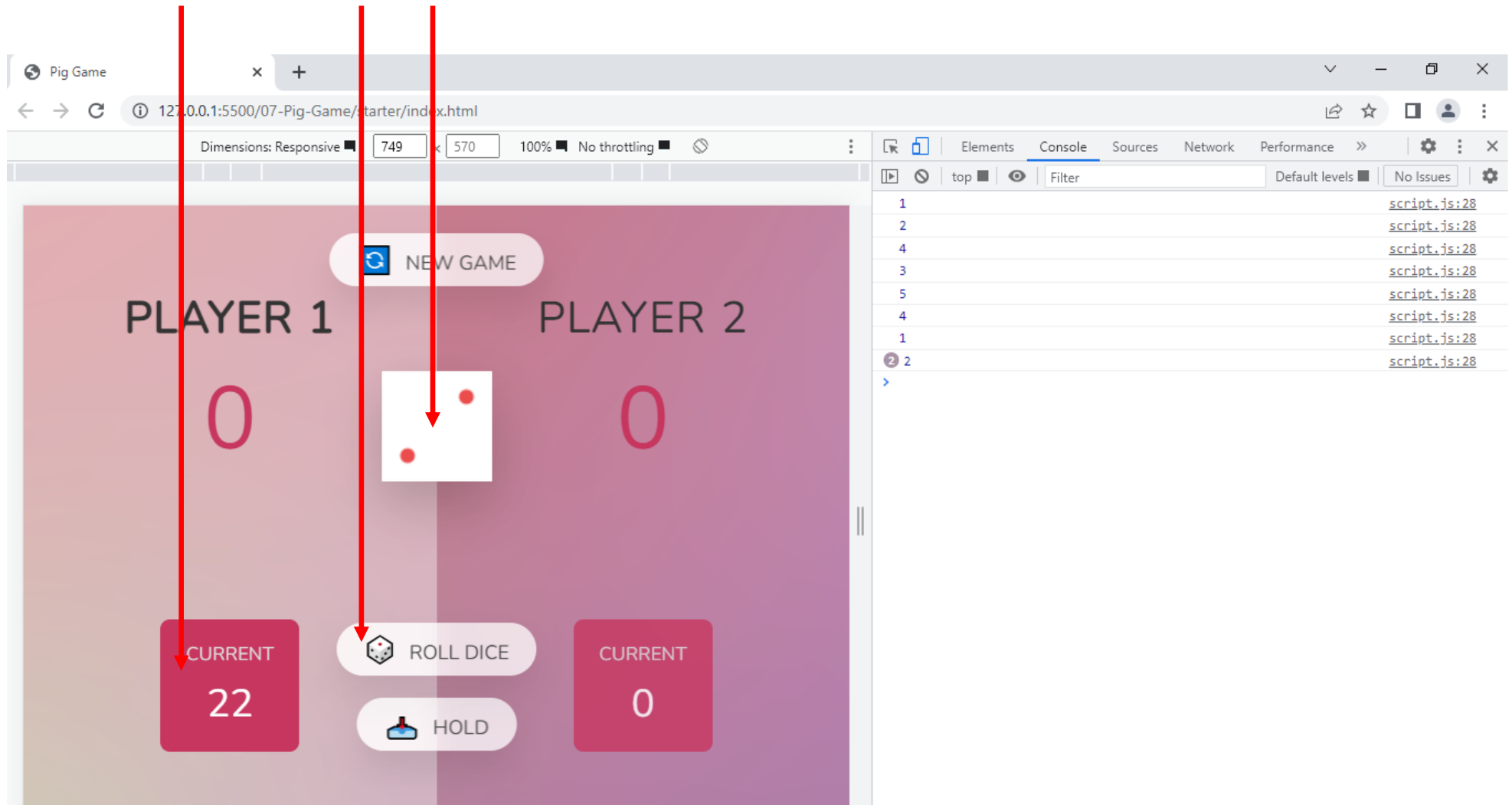
```
// Rolling dice functionality
btnRoll.addEventListener('click', function () {
  // 1. Generate random dice roll number
  const dice = Math.trunc(Math.random() * 6) + 1;
  console.log(dice);

  // 2. Display dice
  diceEl.classList.remove('hidden');
  diceEl.src = `dice-${dice}.png`;

  // 3. Check for rolled 1:
  if (dice !== 1) {
    // Add dice to current score
    currentScore += dice;
    currentEl.textContent = currentScore; // Change later
  } else {
    // Switch to other player
  }
});
```

If the dice roll is not 1 then we need to add the dice rolled value to the current score and update the current score on screen.

Now when the player clicks the roll dice button a random dice value is selected which is continually added to current score and updating the current element. Note that the image of the dice changes depending on the value because the filename for the image is dice-(value).png.



Switching the Active player

```
// Selecting elements
const player0El = document.querySelector('.player--0');
const player1El = document.querySelector('.player--1');

const score0El = document.querySelector('#score--0');
const score1El = document.querySelector('#score--1');
const current0El = document.querySelector('#current--0');
const current1El = document.querySelector('#current--1');
const diceEl = document.querySelector('.dice');

const btnNew = document.querySelector('.btn--new');
const btnRoll = document.querySelector('.btn--roll');
const btnHold = document.querySelector('.btn--hold');

// Starting conditions
score0El.textContent = 0;
score1El.textContent = 0;
diceEl.classList.add('hidden');

const scores = [0, 0];
let currentScore = 0;
let activePlayer = 0;
```

We define the variables for the two players.

We define a new immutable variable for active player and set it to zero. (the active player is either zero or one)

```
// Rolling dice functionality
btnRoll.addEventListener('click', function () {
  // 1. Generate random dice roll number
  const dice = Math.trunc(Math.random() * 6) + 1;
  console.log(dice);

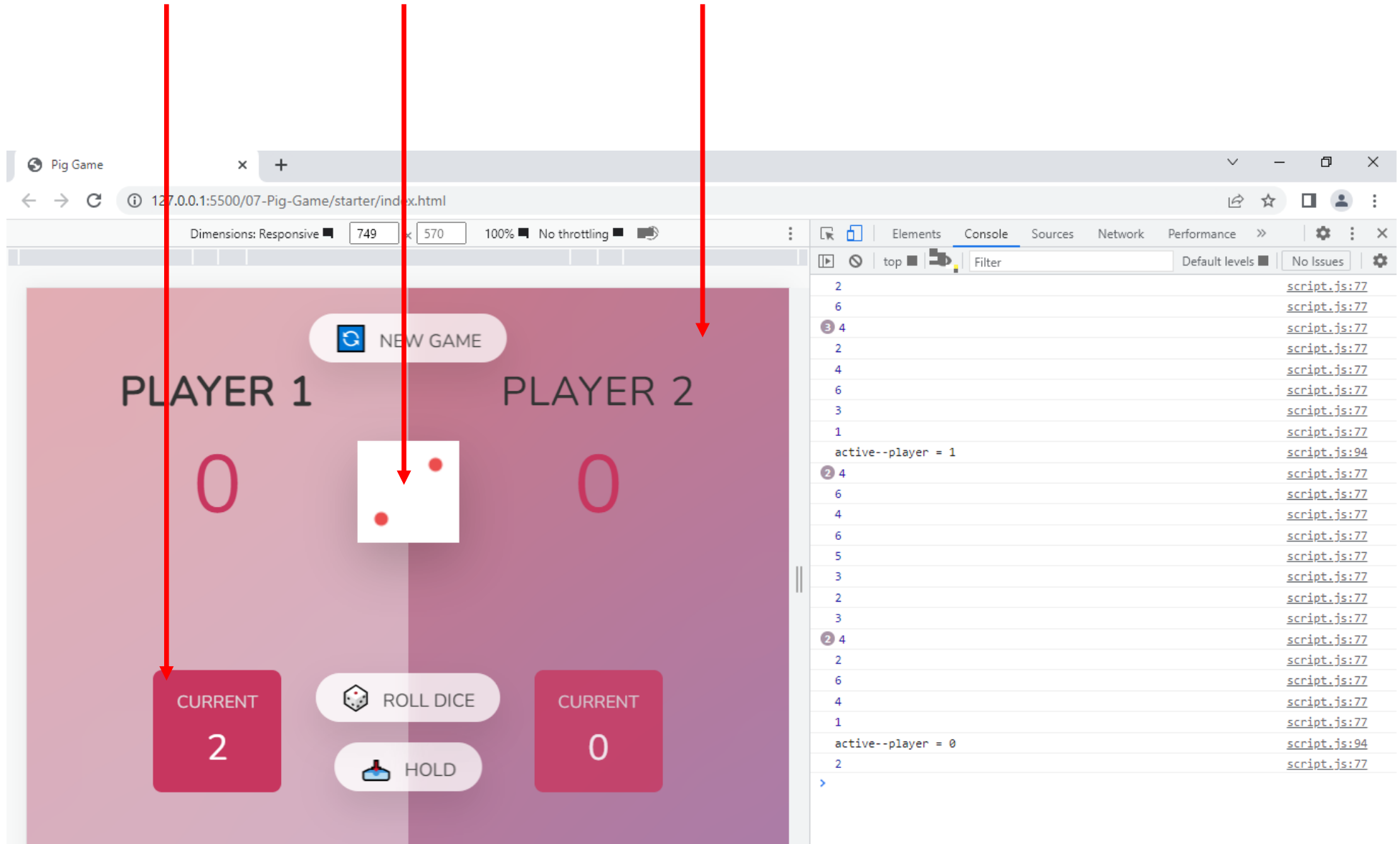
  // 2. Display dice
  diceEl.classList.remove('hidden');
  diceEl.src = `dice-${dice}.png`;

  // 3. Check for rolled 1:
  if (dice !== 1) {
    // Add dice to current score
    currentScore += dice;
    document.getElementById(`current--${activePlayer}`).textContent = currentScore;
  } else {
    // Switch to other player
    document.getElementById(`current--${activePlayer}`).textContent = 0;
    currentScore = 0;
    activePlayer = activePlayer === 0 ? 1 : 0;
    player0El.classList.toggle('player--active');
    player1El.classList.toggle('player--active');
  }
});
```

When the player switches the current score needs to be reset to zero.

We can also toggle the class list to change the background-color of to show the current player. Toggle just means if it has class 'player—active' remove it. If it does not have class 'player—active' add it.

Now when the dice lands on 1 the active player changes, the score `currentScore` resets to zero and the background-colour flips to the other player.



Holding the current Score part 1

```
// Selecting elements
```

```
const player0El = document.querySelector('.player--0');
```

```
const player1El = document.querySelector('.player--1');
```

```
const score0El = document.querySelector('#score--0');
```

```
const score1El = document.querySelector('#score--1');
```

```
const current0El = document.querySelector('#current--0');
```

```
const current1El = document.querySelector('#current--1');
```

```
const diceEl = document.querySelector('.dice');
```

```
const btnNew = document.querySelector('.btn--new');
```

```
const btnRoll = document.querySelector('.btn--roll');
```

```
const btnHold = document.querySelector('.btn--hold');
```

```
// Starting conditions
```

```
score0El.textContent = 0;
```

```
score1El.textContent = 0;
```

```
diceEl.classList.add('hidden');
```

```
const scores = [0, 0]; 2)
```

```
let currentScore = 0;
```

```
let activePlayer = 0;
```

2) We have an array of scores that contains two items. Scores[0] holds the score for player 0 and score[1] holds the score for player 1. Note that the position in the array is also the player

```
// Switch player function
```

```
const switchPlayer = function () {  
  document.getElementById(`current--${activePlayer}`).textContent = 0;  
  currentScore = 0;  
  activePlayer = activePlayer === 0 ? 1 : 0;  
  console.log(`active--player = ${activePlayer}`);  
  player0El.classList.toggle('player--active');  
  player1El.classList.toggle('player--active');  
};
```

```
// Rolling dice functionality
```

```
btnRoll.addEventListener('click', function () {  
  // 1. Generate random dice roll number  
  const dice = Math.trunc(Math.random() * 6) + 1;  
  console.log(dice);  
  
  // 2. Display dice  
  diceEl.classList.remove('hidden');  
  diceEl.src = `dice-${dice}.png`;  
  
  // 3. Check for rolled 1:  
  if (dice !== 1) {  
    // Add dice to current score  
    currentScore += dice;  
    document.getElementById(`current--${activePlayer}`).textContent = currentScore;  
  } else {
```



```
    // Switch to other player  
    switchPlayer(); 5)  
  }  
});
```

1)

```
btnHold.addEventListener('click', function () {  
  // 1. Add current score to active players score  
  scores[activePlayer] += currentScore; 3)  
  console.log(scores);  
  document.getElementById(`score--${activePlayer}`).textContent =  
scores[activePlayer]; 4)  
  // 2. Check if player's score is >= 100  
  
  // Finish the Game  
  // Switch to the other player  
  switchPlayer(); 5)  
});
```

1) We add an event listener to the hold button and define the actions that we want this function to do.

3) As the active player is either 0 or 1 we can use this to identify the position in the scores array and add the current score to that position value.

4) We also need to update the score element of the active player.

5) Now when the hols is clicked we also want to switch player. But this is code we have used before so we put that code in a function and call it.

Now when the hold button is clicked the current value of the active player is added to the scores array.

The screenshot displays a web browser window with the URL `127.0.0.1:5500/07-Pig-Game/starter/index.html`. The game interface features two players, Player 1 and Player 2, with their current scores shown in large red numbers: 22 for Player 1 and 32 for Player 2. A central dice icon is positioned between the players. At the bottom, there are buttons for 'NEW GAME', 'ROLL DICE', and 'HOLD'. The 'HOLD' button is highlighted with a red arrow pointing to it. The browser's console shows the following logs:

```
active--player = 1 script.js:204
3 script.js:213
6 script.js:213
2 script.js:213
4 script.js:213
(2) [10, 32] script.js:234
active--player = 0 script.js:204
2 script.js:213
6 script.js:213
1 script.js:213
active--player = 1 script.js:204
3 script.js:213
2 5 script.js:213
3 script.js:213
5 script.js:213
1 script.js:213
active--player = 0 script.js:204
1 script.js:213
active--player = 1 script.js:204
6 script.js:213
1 script.js:213
active--player = 0 script.js:204
2 5 script.js:213
2 script.js:213
(2) [22, 32] script.js:234
active--player = 1 script.js:204
```

Red arrows indicate the flow of data: one arrow points from the 'HOLD' button to the console log entry `(2) [10, 32]`, and another arrow points from the same log entry to the console log entry `(2) [22, 32]`, showing how the current score of the active player is added to the scores array.

Holding the current Score part 2

// Selecting elements

```
const player0El = document.querySelector('.player--0');
```

```
const player1El = document.querySelector('.player--1');
```

```
const score0El = document.querySelector('#score--0');
```

```
const score1El = document.querySelector('#score--1');
```

```
const current0El = document.querySelector('#current--0');
```

```
const current1El = document.querySelector('#current--1');
```

```
const diceEl = document.querySelector('.dice');
```

```
const btnNew = document.querySelector('.btn--new');
```

```
const btnRoll = document.querySelector('.btn--roll');
```

```
const btnHold = document.querySelector('.btn--hold');
```

// Starting conditions

```
score0El.textContent = 0;
```

```
score1El.textContent = 0;
```

```
diceEl.classList.add('hidden');
```

```
const scores = [0, 0];
```

```
let currentScore = 0;
```

```
let activePlayer = 0;
```

```
// Switch player function
const switchPlayer = function () {
  document.getElementById(`current--${activePlayer}`).textContent = 0;
  currentScore = 0;
  activePlayer = activePlayer === 0 ? 1 : 0;
  console.log(`active--player = ${activePlayer}`);
  player0El.classList.toggle('player--active');
  player1El.classList.toggle('player--active');
};
```

```
// Rolling dice functionality
btnRoll.addEventListener('click', function () {
  // 1. Generate random dice roll number
  const dice = Math.trunc(Math.random() * 6) + 1;
  console.log(dice);

  // 2. Display dice
  diceEl.classList.remove('hidden');
  diceEl.src = `dice-${dice}.png`;

  // 3. Check for rolled 1:
  if (dice !== 1) {
    // Add dice to current score
    currentScore += dice;
    document.getElementById(`current--${activePlayer}`).textContent =
      currentScore;
  }
});
```

```

    } else {
      // Switch to other player
      switchPlayer();
    }
  });

```

```

btnHold.addEventListener('click', function () {
  // 1. Add current score to active players score
  scores[activePlayer] += currentScore;
  console.log(scores);
  document.getElementById(`score--${activePlayer}`).textContent =
    scores[activePlayer];

  // 2. Check if player's score is >= 100
  if (scores[activePlayer] >= 100) {
    // Finish the Game
    document.querySelector(`.player--${activePlayer}`).classList.add('player--
winner');
    document.querySelector(`.player--${activePlayer}`).classList.remove('player--
active');
  } else {
    // Switch to the other player
    switchPlayer();
  }
});

```

When the active player's score is greater than or equal to 100 we want to finish the game by adding a player--winner class and removing the player--active class.

Player-winner class added and player—active class removed.

The screenshot displays a web browser window with the title "Pig Game" and the URL "127.0.0.1:5500/07-Pig-Game/starter/index.html". The browser's developer tools are open, showing the Console tab with a list of log messages. A red arrow points from the top left of the browser window down to the game interface.

The game interface is split into two columns for Player 1 and Player 2. Player 1's side is dark grey, and Player 2's side is light purple. In the center, a white die with red pips shows a 6. At the top, a "NEW GAME" button is visible. At the bottom, there are "CURRENT" score boxes, a "ROLL DICE" button with a die icon, and a "HOLD" button with a hand icon.

Player 1's score is 105, and Player 2's score is 77. Player 1's current score is 6, and Player 2's current score is 0.

The console logs show the following sequence of events:

- Log 4: `script.js:213`
- Log 2: `(2) [72, 59]` `script.js:234`
- Log: `active--player = 1` `script.js:204`
- Log 2: `4` `script.js:213`
- Log 2: `2` `script.js:213`
- Log 4: `4` `script.js:213`
- Log 2: `(2) [72, 73]` `script.js:234`
- Log: `active--player = 0` `script.js:204`
- Log 6: `6` `script.js:213`
- Log 4: `4` `script.js:213`
- Log 5: `5` `script.js:213`
- Log 2: `(2) [87, 73]` `script.js:234`
- Log: `active--player = 1` `script.js:204`
- Log 5: `5` `script.js:213`
- Log 1: `1` `script.js:213`
- Log: `active--player = 0` `script.js:204`
- Log 3: `3` `script.js:213`
- Log 4: `4` `script.js:213`
- Log 5: `5` `script.js:213`
- Log 2: `(2) [99, 73]` `script.js:234`
- Log: `active--player = 1` `script.js:204`
- Log 4: `4` `script.js:213`
- Log 2: `(2) [99, 77]` `script.js:234`
- Log: `active--player = 0` `script.js:204`
- Log 6: `6` `script.js:213`
- Log 2: `(2) [105, 77]` `script.js:234`

However, We can still continue to play the game until both players have player-winner class!
This needs to be fixed!

The screenshot displays a web browser window with the title "Pig Game" and the URL "127.0.0.1:5500/07-Pig-Game/starter/index.html". The browser's developer tools are open, showing the Console tab with a list of log messages.

The game interface has a dark background. At the top, there is a "NEW GAME" button. Below it, the scores for "PLAYER 1" and "PLAYER 2" are displayed in large red numbers: 208 and 101, respectively. In the center, a white square represents a die with four red dots. At the bottom, there are two red boxes labeled "CURRENT" containing the numbers 0 and 4. Between these boxes are two buttons: "ROLL DICE" with a die icon and "HOLD" with a hand icon.

The console logs show the following sequence of events:

- Log 4: `script.js:213`
- Log 6: `script.js:213`
- Log 2: `script.js:213`
- Log 1: `script.js:213`
- Log: `active--player = 1` (`script.js:204`)
- Log 4: `script.js:213`
- Log 2: `script.js:213`
- Log 1: `script.js:213`
- Log: `active--player = 0` (`script.js:204`)
- Log 5: `script.js:213`
- Log 1: `script.js:213`
- Log: `active--player = 1` (`script.js:204`)
- Log 6: `script.js:213`
- Log 3: `script.js:213`
- Log: `(2) [208, 97]` (`script.js:234`)
- Log: `active--player = 0` (`script.js:204`)
- Log 2: `script.js:213`
- Log 3: `script.js:213`
- Log 2: `script.js:213`
- Log 3: `script.js:213`
- Log 1: `script.js:213`
- Log: `active--player = 1` (`script.js:204`)
- Log 4: `script.js:213`
- Log: `(2) [208, 101]` (`script.js:234`)

Holding the current Score part 2

```
// Selecting elements
const player0El = document.querySelector('.player--0');
const player1El = document.querySelector('.player--1');

const score0El = document.querySelector('#score--0');
const score1El = document.querySelector('#score--1');
const current0El = document.querySelector('#current--0');
const current1El = document.querySelector('#current--1');
const diceEl = document.querySelector('.dice');

const btnNew = document.querySelector('.btn--new');
const btnRoll = document.querySelector('.btn--roll');
const btnHold = document.querySelector('.btn--hold');

// Starting conditions
score0El.textContent = 0;
score1El.textContent = 0;
diceEl.classList.add('hidden');

const scores = [0, 0];
let currentScore = 0;
let activePlayer = 0;
let playing = true; 1)
```

1) We add a Boolean variable called **playing** to hold a value of **true** if the game is playing and **false** if the game has finished.


```
// Switch player function
const switchPlayer = function () {
  document.getElementById(`current--${activePlayer}`).textContent = 0;
  currentScore = 0;
  activePlayer = activePlayer === 0 ? 1 : 0;
  console.log(`active--player = ${activePlayer}`);
  player0El.classList.toggle('player--active');
  player1El.classList.toggle('player--active');
};
```

```
// Rolling dice functionality
btnRoll.addEventListener('click', function () {
  // Is game playing
  if (playing) { 2)
    // 1. Generate random dice roll number
    const dice = Math.trunc(Math.random() * 6) + 1;
    console.log(dice);

    // 2. Display dice
    diceEl.classList.remove('hidden');
    diceEl.src = `dice-${dice}.png`;
  }
});
```

2) We wrap all the code that we want to only run if the game is playing in a if (playing) clause. Note that because playing is a Boolean we do not need to say if (playing === true)

```

// 3. Check for rolled 1:
if (dice !== 1) {
    // Add dice to current score
    currentScore += dice;
    document.getElementById(`current--${activePlayer}`).textContent =
        currentScore;
} else {
    // Switch to other player
    switchPlayer();
}
} // End of If playing 2)
});

```

3) For the event listener that handles the hold button we also wrap it in an If (playing)

```

btnHold.addEventListener('click', function () {
    if (playing) { 3)
        // 1. Add current score to active players score
        scores[activePlayer] += currentScore;
        console.log(scores);
        document.getElementById(`score--${activePlayer}`).textContent =
            scores[activePlayer];
    }
});

```

```

// 2. Check if player's score is >= 100
if (scores[activePlayer] >= 100) {
  // Finish the Game
  playing = false; 4)
  diceEl.classList.add('hidden');
  btnRoll.classList.add('hidden'); 5)
  btnHold.classList.add('hidden');
  document
    .querySelector(`.player--${activePlayer}`)
    .classList.add('player--winner');
  document
    .querySelector(`.player--${activePlayer}`)
    .classList.remove('player--active');
} else {
  // Switch to the other player
  switchPlayer();
}
} // End of if playing 3)
});

```

4) If the current players scores are greater than or equal to 100 the we want to stop the game playing so we set playing to false.

5) We can also set the dice, button roll and button hold to hidden.

Now the game can only have one winner. Once a players score reaches 100 then the dice, roll and hold buttons are hidden.

The screenshot displays a web browser window titled "Pig Game" with the address bar showing "127.0.0.1:5500/07-Pig-Game/starter/index.html". The browser's developer tools are open, showing the Console tab with a list of log messages.

The game interface is split into two columns for Player 1 and Player 2. Player 1's side has a dark background, while Player 2's side has a light purple background. A red rounded rectangle highlights the area between the two players' scores.

Player 1:

- Score: 108
- Current Score: 27

Player 2:

- Score: 26
- Current Score: 0

A "NEW GAME" button is located at the top center of the interface.

Console Log:

```
(2) [81, 0] script.js:320
active--player = 1 script.js:286
4 script.js:297
2 6 script.js:297
3 script.js:297
1 script.js:297
active--player = 0 script.js:286
1 script.js:297
active--player = 1 script.js:286
5 script.js:297
4 script.js:297
2 5 script.js:297
1 script.js:297
active--player = 0 script.js:286
1 script.js:297
active--player = 1 script.js:286
3 script.js:297
4 script.js:297
3 5 script.js:297
4 script.js:297
(2) [81, 26] script.js:320
active--player = 0 script.js:286
2 3 script.js:297
6 script.js:297
3 5 script.js:297
(2) [108, 26] script.js:320
```

New Game Button

```
// Selecting elements
const player0El = document.querySelector('.player--0');
const player1El = document.querySelector('.player--1');

const score0El = document.querySelector('#score--0');
const score1El = document.querySelector('#score--1');
const current0El = document.querySelector('#current--0');
const current1El = document.querySelector('#current--1');
const diceEl = document.querySelector('.dice');

const btnNew = document.querySelector('.btn--new');
const btnRoll = document.querySelector('.btn--roll');
const btnHold = document.querySelector('.btn--hold');

// Scoping the variables
let scores, currentScore, activePlayer, playing;
```

2) Now that we have the starting game variables set to the correct values it makes no sense to declare the values for the variables do we can just define them in one line of code.

```
// Game initialise function
const init = function () {
  // 1. reset scores array to zero.
  scores = [0, 0];
  // 2. reset current score to zero
  currentScore = 0;
  // 3. Set active player to 0
  activePlayer = 0;
  // 4. reset playing to true
  playing = true;
  // 5. reset player scores to zero
  score0El.textContent = 0;
  score1El.textContent = 0;
  // 6. reset current player scores to zero
  current0El.textContent = 0;
  current1El.textContent = 0;
  // 7. remove hidden class from btnRoll and BtnHold, hide dice
  diceEl.classList.add('hidden');
  btnRoll.classList.remove('hidden');
  btnHold.classList.remove('hidden');
  // 3. Remove player--winner class
  player0El.classList.remove('player--winner');
  player1El.classList.remove('player--winner');
  // 4. Remove player--active class from player 1 and add it to player 0
  player0El.classList.add('player--active');
  player1El.classList.remove('player--active');
};
```

1) For the new game button to work we need to reset all the variables to values that they were at the start of the game. It makes sense to put this code in an initialise function.

3) We will run the initialise function after creating it so that values are set to the correct numbers straight away.

```
init();
```

```
// Switch player function
```

```
const switchPlayer = function () {  
  document.getElementById(`current--${activePlayer}`).textContent = 0;  
  currentScore = 0;  
  activePlayer = activePlayer === 0 ? 1 : 0;  
  console.log(`active--player = ${activePlayer}`);  
  player0El.classList.toggle('player--active');  
  player1El.classList.toggle('player--active');  
};
```

```
// Rolling dice functionality
```

```
btnRoll.addEventListener('click', function () {  
  // Is game playing  
  if (playing) {  
    // 1. Generate random dice roll number  
    const dice = Math.trunc(Math.random() * 6) + 1;  
    console.log(dice);  
  
    // 2. Display dice  
    diceEl.classList.remove('hidden');  
    diceEl.src = `dice-${dice}.png`;  
  }  
});
```

```
// 3. Check for rolled 1:
if (dice !== 1) {
    // Add dice to current score
    currentScore += dice;
    document.getElementById(`current--${activePlayer}`).textContent =
        currentScore;
} else {
    // Switch to other player
    switchPlayer();
}
} // End of If playing
});
```

```
btnHold.addEventListener('click', function () {
    if (playing) {
        // 1. Add current score to active players score
        scores[activePlayer] += currentScore;
        console.log(scores);
        document.getElementById(`score--${activePlayer}`).textContent =
            scores[activePlayer];
    }
});
```



```

// 2. Check if player's score is >= 100
if (scores[activePlayer] >= 100) {
    // Finish the Game
    playing = false;
    diceEl.classList.add('hidden');
    btnRoll.classList.add('hidden');
    btnHold.classList.add('hidden');
    document
        .querySelector(`.player--${activePlayer}`)
        .classList.add('player--winner');
    document
        .querySelector(`.player--${activePlayer}`)
        .classList.remove('player--active');
} else {
    // Switch to the other player
    switchPlayer();
}
} // End of if playing
});

```

```

btnNew.addEventListener('click', init);

```

4) Now the event listener for the new game button just calls the initialise function when it is clicked.

How JavaScript works behind the Scenes.

High-Level

Garbage Collected

Interpreted or Just-in-time Compiled

Multi-Paradigm

Prototype-based
Object-Orientated

First Class Functions

Dynamic

Single Threaded

Non-Blocking Event
Loop

JavaScript high level overview



Memory ← Computer Resources → CPU

C+ language



Low-Level

↑
Developer has to
manage resources
manually

Javascript

Python



High-Level

↑
Developer DOES NOT have
to worry, Everything
happens automatically

High-Level

Garbage Collected

Interpreted or Just-in-time Compiled

Multi-Paradigm

**Prototype-based
Object-Orientated**

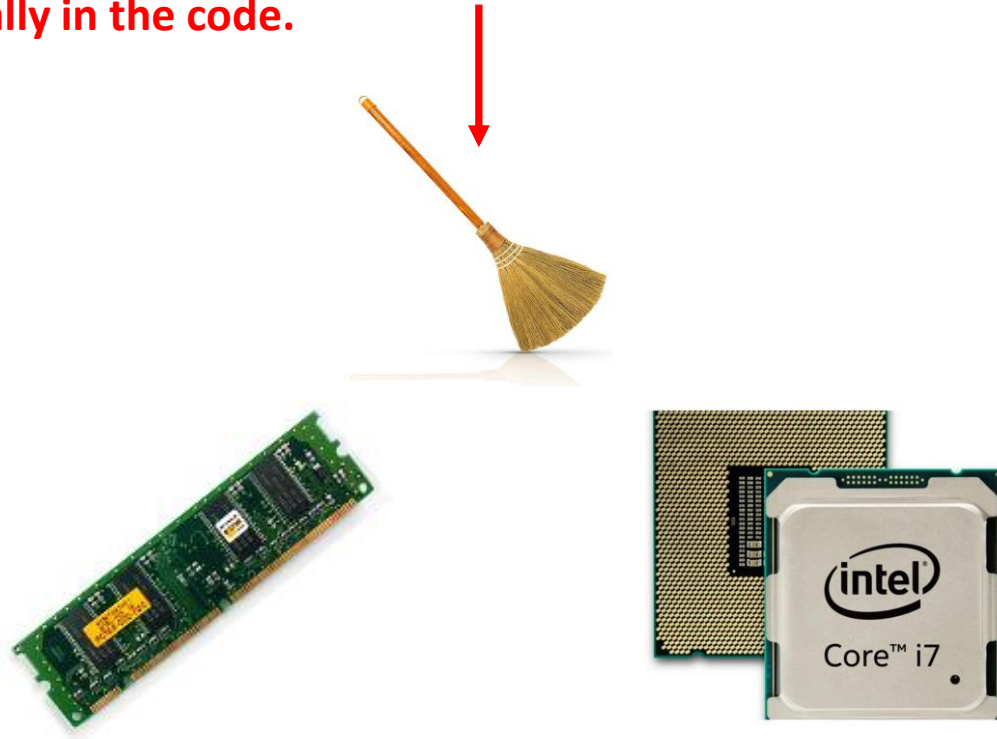
First Class Functions

Dynamic

Single Threaded

**Non-Blocking Event
Loop**

JavaScript has automatic Garbage-Collection which cleans the memory from time to time so we do not have to do it manually in the code.



Memory ← Computer Resources → CPU

High-Level

Garbage Collected

Interpreted or Just-in-time Compiled

Multi-Paradigm

Prototype-based Object-Orientated

First Class Functions

Dynamic

Single Threaded

Non-Blocking Event Loop



JavaScript is an interpreted or just-in-time compiled code. I.e. It converts human readable JavaScript code in to machine readable code i.e. 1s and 0s.

High-Level

Garbage Collected

Interpreted or Just-in-time Compiled

Multi-Paradigm

**Prototype-based
Object-Orientated**

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

This is what we have been using so far in this course

- 1** Procedural Programming
- 2** Object-Orientated Programming (OOP)
- 3** Functional-Programming (FP):

JavaScript is very versatile and flexible and can support any of the three paradigm's

High-Level

Garbage Collected

Interpreted or Just-in-time Compiled

Multi-Paradigm

Prototype-based Object-Orientated

First Class Functions

Dynamic

Single Threaded

Non-Blocking Event Loop

Almost everything in JavaScript is an object (except for primitive values like strings or numbers etc)

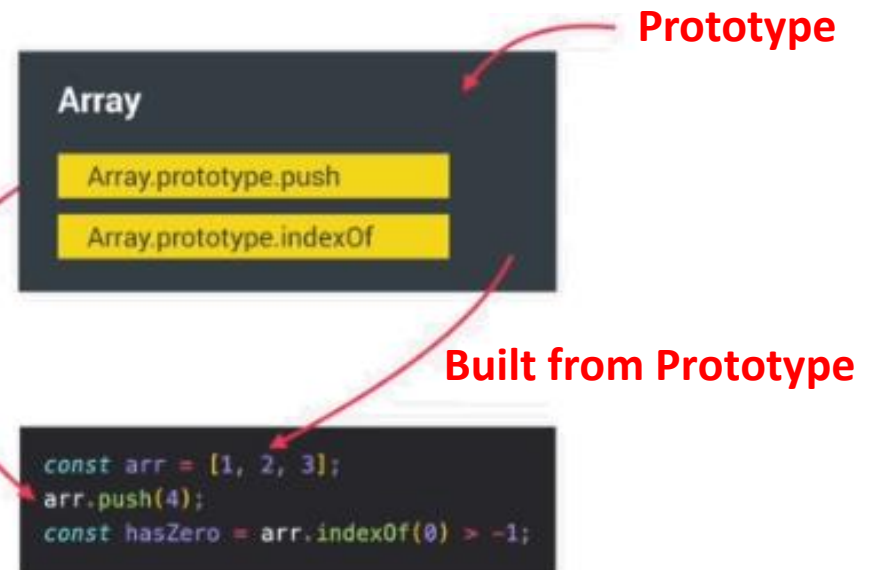
Arrays are just objects.

We can use push method on arrays because of prototype inheritance.

We create arrays from array blueprint which is called a prototype.

The prototype contains all the methods we can use on it so we can reuse methods for all prototypes such as arrays.

Our array inherits methods from Prototype



High-Level

Garbage Collected

Interpreted or Just-in-time Compiled

Multi-Paradigm

Prototype-based
Object-Orientated

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.

We have already used a first class function when we passed the close modal function into the event listener function.

Not all languages have this ability.

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

Passing a function into a
function as an argument
– First-Class Functions!

High-Level

Garbage Collected

Interpreted or Just-in-time Compiled

Multi-Paradigm

Prototype-based
Object-Orientated

First Class Functions

Dynamic

Single Threaded

Non-Blocking Event
Loop

Dynamically-typed Language:

No Data type definitions:
Types become known at
runtime.

Data type of variable is
automatically changed.

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

Many other programming languages insist on manually
assigned datatypes.



High-Level

Garbage Collected

Interpreted or Just-in-time Compiled

Multi-Paradigm

**Prototype-based
Object-Orientated**

First Class Functions

Dynamic

Single Threaded

**Non-Blocking Event
Loop**

Concurrency model: How the JavaScript Engine handles multiple tasks happening at the same time.

JavaScript runs in a **single Thread** so it can only do one thing at a time. (A Thread is like a set of instructions that run in a computer's CPU.)

A long running task would therefore block the thread but we want non-blocking behaviour.

By using an **Event loop**. This takes long running tasks and executes them in the background then puts them back in the main thread once they are finished.

JavaScript Engine and Runtime

A JavaScript engine is a program that executes JavaScript code. Below are examples of Engines.



**Google V8 - Chrome
and NodeJS**



**Nitro -
Apple**



**Spider Monkey -
Firefox.**



**Chakra Core -
Microsoft Edge.**

**Any JavaScript Engine
will include a call stack
and a heap.**

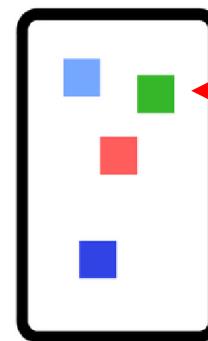
Execution Context



**Where our code is
executed.**



Call Stack

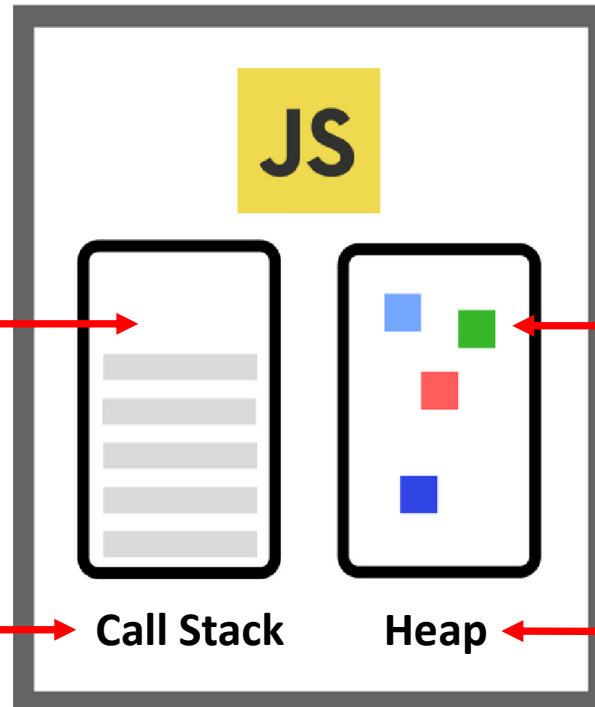


Heap

Objects in Memory



**Where our Objects
are stored.**



Compilation vs Interpretation

Compilation: Entire code is converted into machine code at once, and written to a binary file that can be executed by a computer.



Interpretation: Interpreter runs through the source code and executes it line by line.



JavaScript used to be a compilation language but the problem is that they are much more slower so it is now a mix between interpretation and compilation which is called just in time. The execution happens just after the compilation.

Modern Just-In-Time Compilation of Javascript

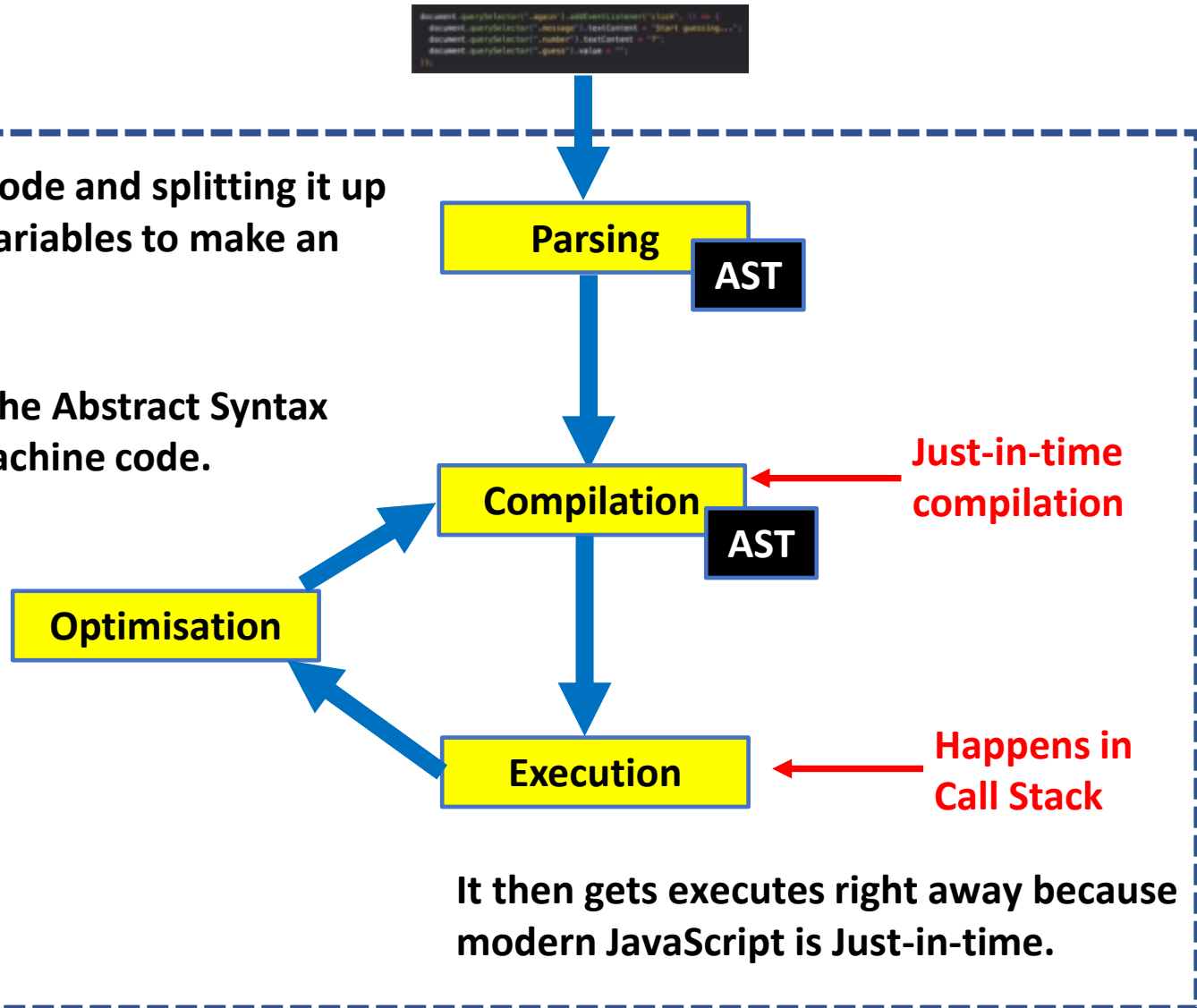


JavaScript Engine

Parsing means reading the code and splitting it up into known pieces such as variables to make an Abstract Syntax tree.

Compilation means taking the Abstract Syntax Tree and converting it to machine code.

In the beginning the Engine creates a very basic un-optimised version of the code to the get the program running then in a background thread it optimises the code for better efficiency and updates the compilation.



JavaScript Runtime in the Browser

Functionalities provided to the engine accessible on windows object

Web APIs

DOM

TIMERS

Fetch API

....

CALLBACK QUEUE

.....

Data

timer

Click

Example: Call-back function from DOM event Listener.

JS Engine

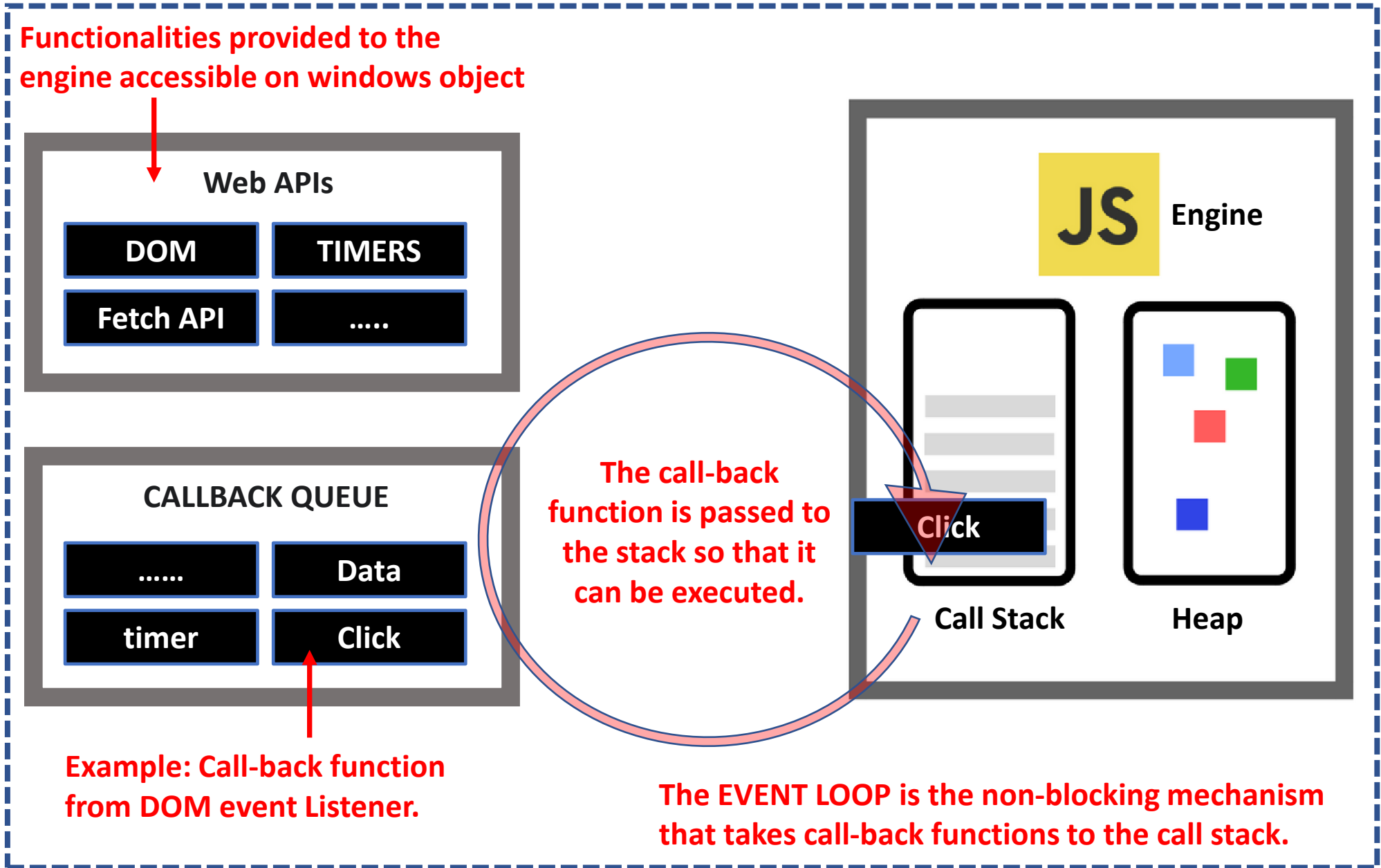
Click

Call Stack

Heap

The call-back function is passed to the stack so that it can be executed.

The EVENT LOOP is the non-blocking mechanism that takes call-back functions to the call stack.



JavaScript Runtime in the NodeJS

NodeJS uses C++ bindings instead of Web APIs

C++ Bindings and Thread Pool

CALLBACK QUEUE

.....

Data

timer

Click

The call-back function is passed to the stack so that it can be executed.

JS Engine

Click

Call Stack

Heap

The EVENT LOOP is the non-blocking mechanism that takes call-back functions to the call stack.

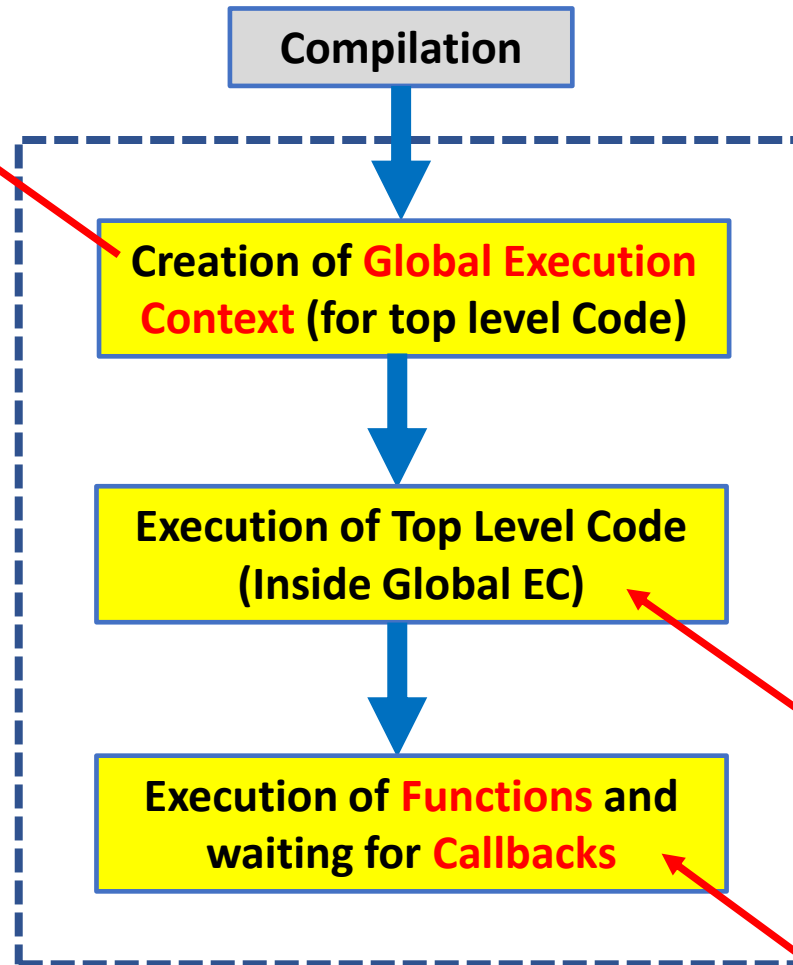
What is an Execution Context?

Top level code is code that is not inside any function, i.e a variable declaration.

Human Readable code:

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

The code within the function will only be executed when that function is called so it is NOT top level code



All together make the call stack

Execution Context:
Environment in which a piece of JavaScript is executed. Stores all the necessary information for some code to be executed.

Exactly one Global Execution Context (EC). Default context, created for code that is not inside any function.

The computer CPU processes the code it receives.

One Execution context per function. For each function call a new EC is created.

Execution Context in detail

Generated during creation phase right before execution

Execution context:

1 Variable Environment

let, const and Var declarations

Functions

Arguments Objects

NOT in arrow functions.

2 Scope Chain

(The scope chain consists of references to variables that are outside of the current function)

Arrow functions do not have the arguments object or the this keyword

3 This keyword

```
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();
```

Literally the function code!

GLOBAL

Name = 'Jonas'

First (function)

Second (function)

X (unknown)

Need to run first() first!

First()

a = 1

b = unknown

Need to run second() first!

second()

c = 2

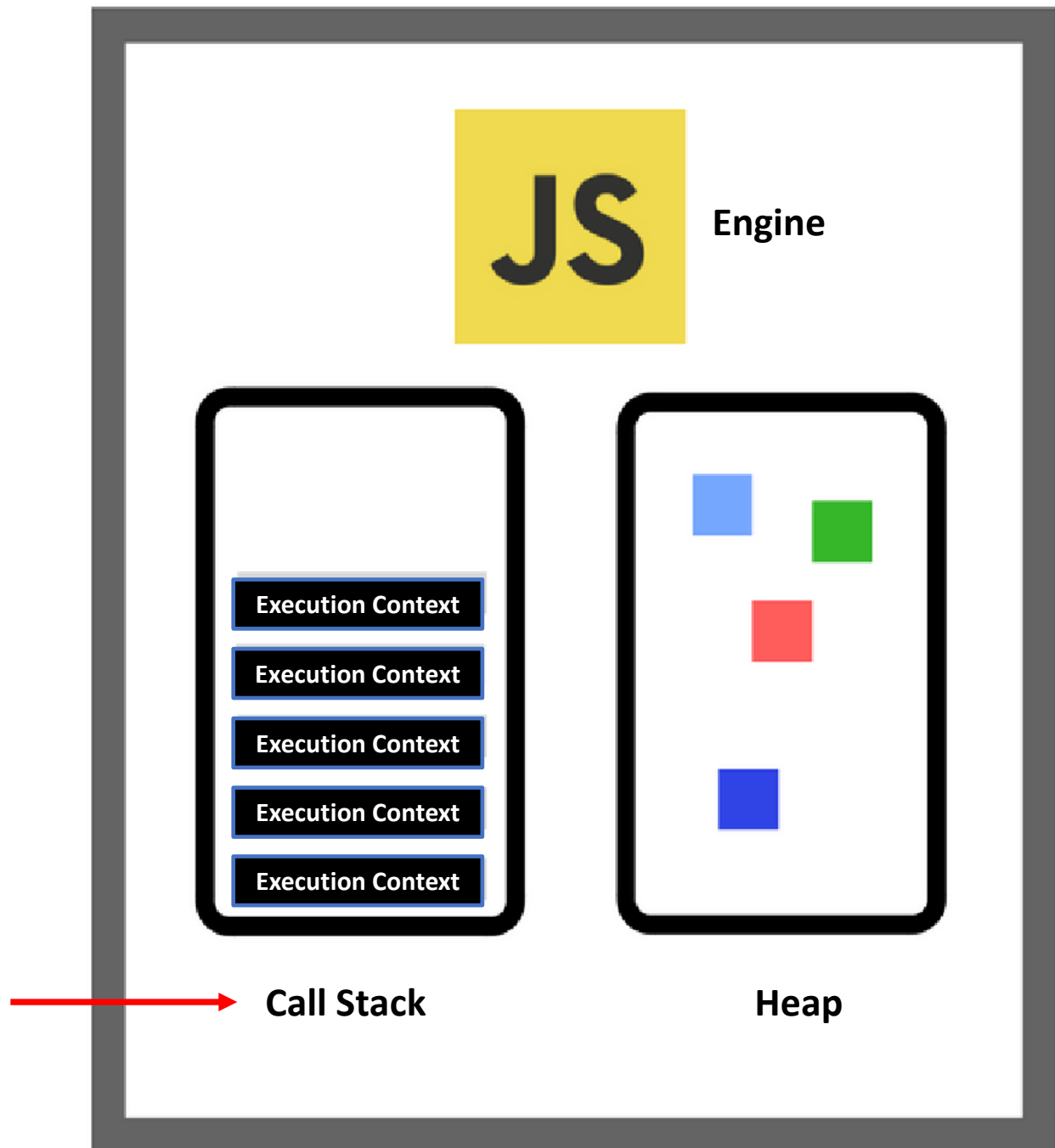
arguments = [7, 9]

(Technically values only become known during execution.)

Array of passed Arguments. Available in all regular functions (except arrow functions)

The Call Stack

Place where the execution Context's get stacked on top of each other to keep track of where we are in the execution.



The Call Stack

```
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();
```

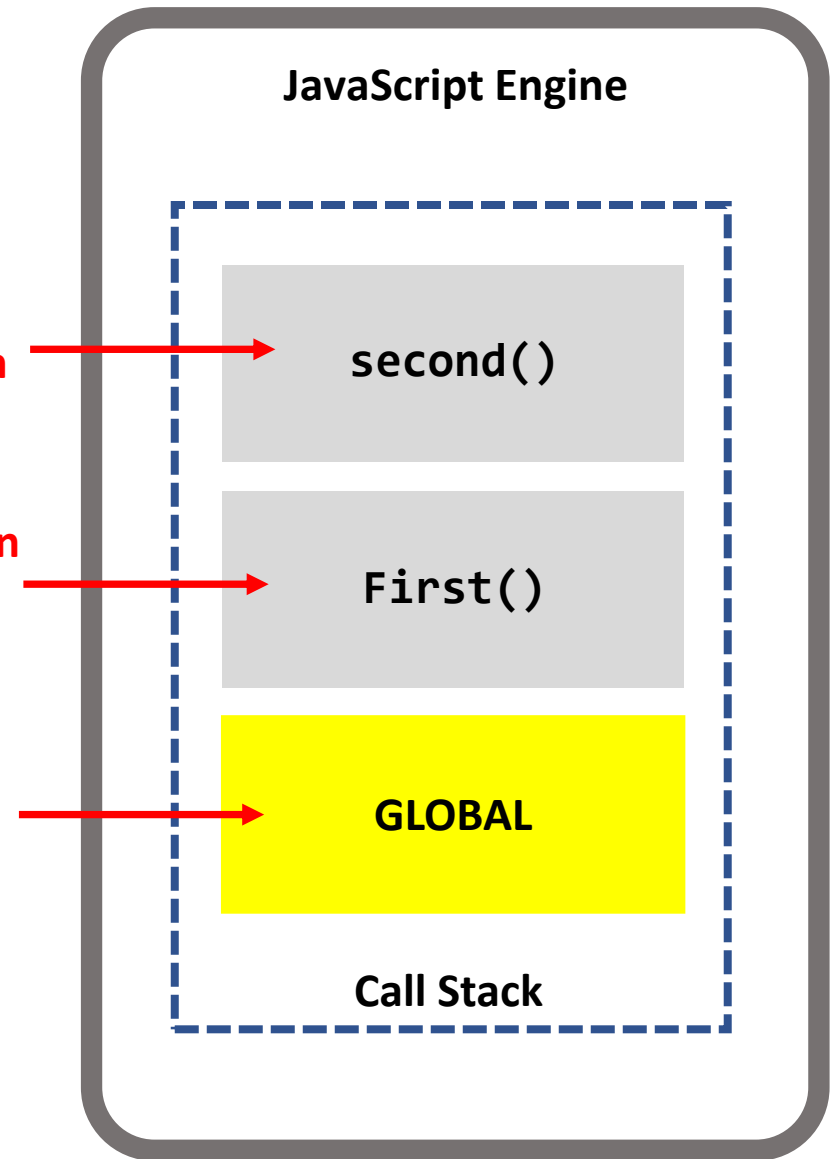
3) Now the first function is paused while the second function is executed.

First will resume when second returns

2) The first gets its own EC and gets put in the call stack

1) The Global EC gets put in the call stack first

4) Now const x can be calculated because first and second functions have been executed.



Once all functions are executed they are popped of the call stack and the stack remains in a waiting state until new EC's are passed in or the program is terminated (browser closed!)

The Scope Chain

Scoping: How our programs variables are **organised** and **accessed**. Where do variables live or where we can access a certain variable, and where not?

Lexical Scoping: Scoping is controlled by **placement** of functions and blocks in the code. For example a function that is written inside another function has access to the variables of the parent function.

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 a code where a certain variable can be **accessed**.

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 millenial = true;  
  const food = 'Avocado toast';  
} ← Example: if block, for loop block, etc.  
  
console.log(millenial); // ReferenceError
```

Variables are only accessible **inside block**.

HOWEVER, this only applies to let and const variables.

Functions are **also block scoped** (only in strict mode)

```
const myName = 'Jonas';

function first() {
  const age = 30;

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

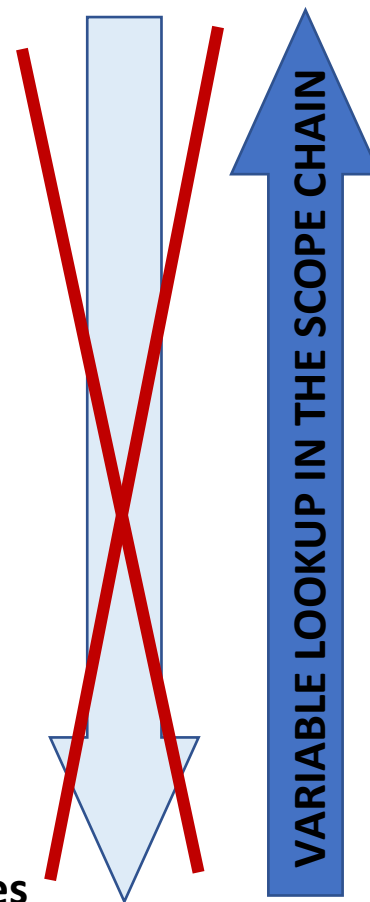
  function second() {
    const job = 'teacher';

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

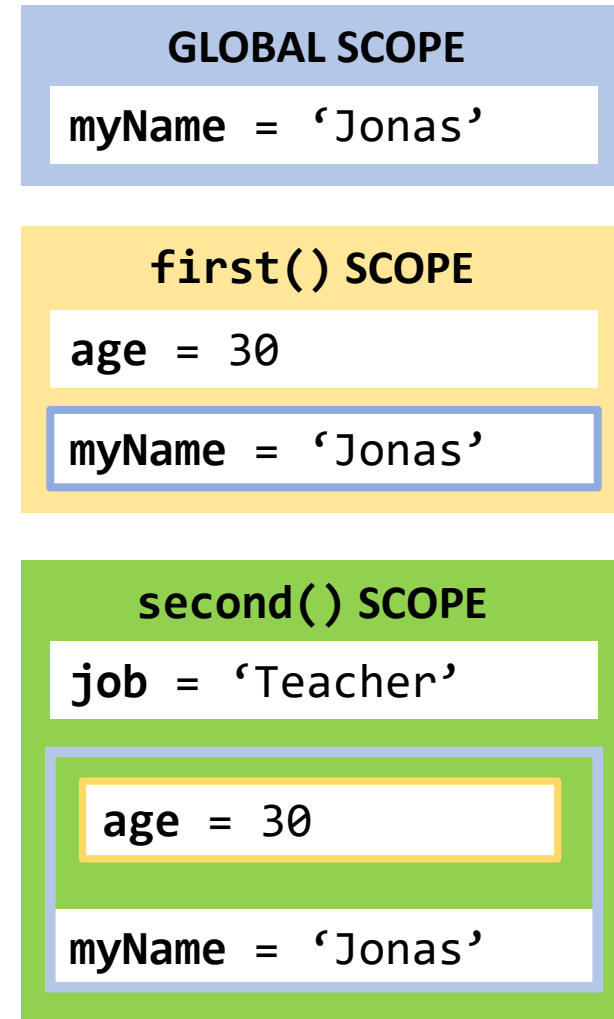
  second();
}

first();
```

Variables not in current scope



The Scope Chain



How can the second function access the variables myName and age that are out of it's scope?

Child functions have access to the variables of parent functions.

The first function has access to its parent global scope variable of myName.

Scopes look up to find missing variables. They never look down. i.e. first scope will never get access the job variable in the second scope. Global scope cannot access any first or second scope variables.

```

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();

```

**Let and const are
block scoped**

**Block scoped
variables are not
accessible outside
of the block**

GLOBAL SCOPE

myName = 'Jonas'

first() SCOPE

age = 30

myName = 'Jonas'

If block SCOPE

decade = 3

Var Millennial = true

second() SCOPE

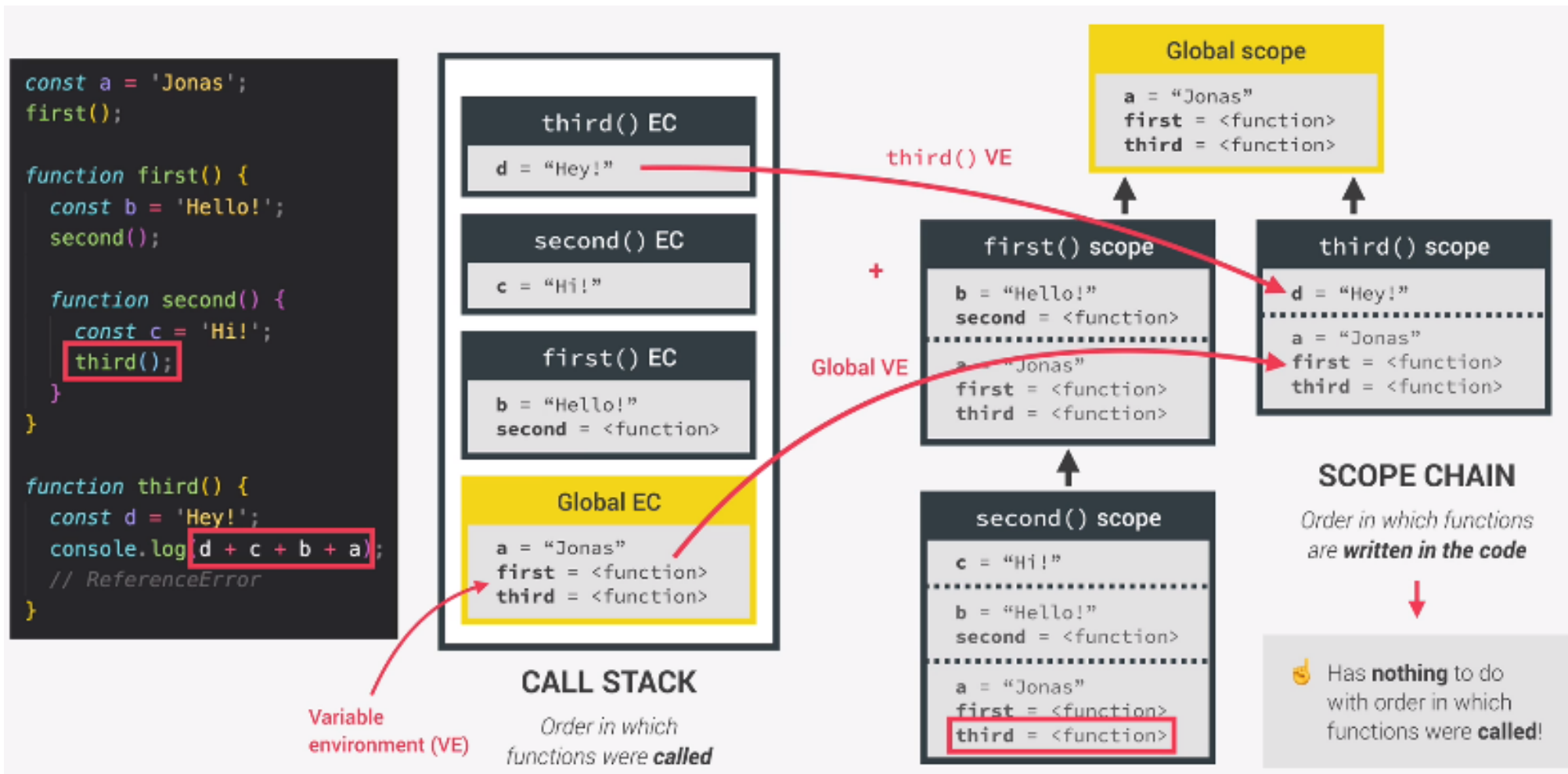
job = 'Teacher'

age = 30

myName = 'Jonas'

Var variables are function scoped, not block scoped.

The Scope Chain vs Call Stack



Scoping has nothing to do with the call stack. Each function call has an Execution context in the call stack which is filled with the variables. The variables are formed when the function is called, not if it is scoped. Note that in this diagram `const c` is not accessible to the third function.

Scoping summary

Scoping asks the question: “where do variables live?” or “where can we access a certain variable, and where not?”

There are three types of scope in Javascript, 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 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 is looking for. This is called variable lookup.

The scope chain is a one way street: a scope will never, ever have access the variables in an inner scope.

The scope chain in a certain scope is equal to adding together all the variable environments of all the 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!

Hoisting in JavaScript

Hoisting: Makes some types of variables accessible/usable in the code before they actually declared. “Variables lifted to the top of the scope”.

Behind the scenes

Before Execution: Code is scanned for variable declarations, and for each new variable; a new property is created in the **variable environment object**.

	Hoisted	Initial Value	Scope
Function Declarations	YES	Actual Function	Block * ₁
var Variables	YES	Undefined	Function
let and const Variables	NO * ₂	<uninitialized>, TDZ * ₃	Block
Function expressions and arrows	Depends if using var or let/const variables * ₄		

*₁ In strict mode, otherwise scope is function.

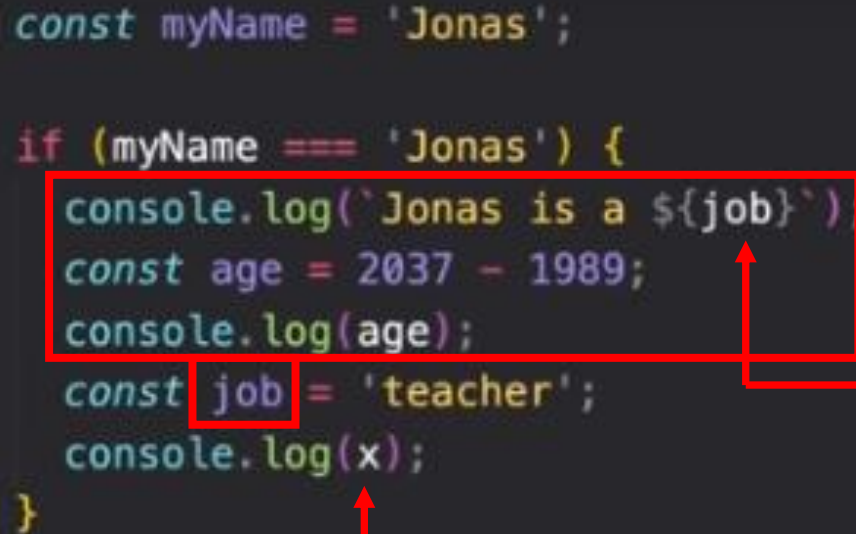
*₂ Technically yes, but not in practice.

*₃ Temporal dead zone.

*₄ If function expressions and arrows are created using var then they assume the characteristics of var, i.e. they are hoisted, are undefined and have function scope. If they are created using let or const then they are not hoisted, are uninitialized, TDZ and have block scope.

Temporal Dead Zone

```
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);  
}
```



The variable `job` is a `const` so it is scoped as block and is so only accessible from the `if` block where it is defined.

There exists a temporal dead zone where the variable is defined but cannot be used. In this temporal dead zone it is as if the variable does not even exist.

Reference error: cannot access 'job' before initialization.

If we try to access a variable, `x`, that is not even created we get a different error message.

Reference error: `x` is not defined.

JavaScript knows about the variable `job`, it is in the variable environment but in a state of uninitialized. Once JavaScript reaches the line of code where 'job' is declared then it is removed from the temporal dead zone and initialized.

In conclusion, `const` and `let` variables each get their own temporal dead zone that starts at the beginning of the scope and ends where they are declared.

Why TDZ? The temporal dead zone **makes it easier to avoid and catch errors**: accessing variables before declaration is bad practice and should be avoided. The TDZ makes this more obvious with an error message.

Why Hoisting? Hoisting allows **using functions before actual declaration** var hoisting is a by-product of this because it var was originally the only way to declare variables.

```
var x = 1;
let y = 2;
const z = 3;
```

3 variables are defined using var, let and const.

In console log if we look at the window property. Window is the global object of the JavaScript browser. Scrolling down some way we get to x = 1. The var defined variable is showing up in the window but not let or const.

Variables created with let and const so not show up on the window property.

In some cases this can have implications.

```
> window
< Window {window: Window, self: Window, document: document, name: '', location: Location, ...} ⓘ
  alert: f alert()
  atob: f atob()
  blur: f blur()
  btoa: f btoa()
  caches: CacheStorage {}
  cancelAnimationFrame: f cancelAnimationFrame()
  cancelIdleCallback: f cancelIdleCallback()
  captureEvents: f captureEvents()
  chrome: {loadTimes: f, csi: f}
  clearInterval: f clearInterval()
  clearTimeout: f clearTimeout()
  clientInformation: Navigator {vendorSub: '', productSub: '20030107', vendor: 'Google Inc.', maxTouchPoints: 1, userActivat
  close: f close()

  webkitCancelAnimationFrame: f webkitCancelAnimationFrame()
  webkitRequestAnimationFrame: f webkitRequestAnimationFrame()
  webkitRequestFileSystem: f webkitRequestFileSystem()
  webkitResolveLocalFileSystemURL: f webkitResolveLocalFileSystemURL()
  webkitStorageInfo: DeprecatedStorageInfo {}
  window: Window {window: Window, self: Window, document: document, name: '', location: Location, ...}
    x: 1
    Infinity: Infinity
  AbortController: f AbortController()
  AbortSignal: f AbortSignal()
  AbsoluteOrientationSensor: f AbsoluteOrientationSensor()
  AbstractRange: f AbstractRange()
  Accelerometer: f Accelerometer()
  AggregateError: f AggregateError()
```

This Keyword

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 **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**.

METHOD: this = <Object that is calling the method>

Inside the method we are using the this keyword. The object calling the method is Jonas. As the object called Jonas contains year, this will now use the value of 1989 from Jonas object.

SIMPLE FUNCTION CALL: this = undefined

In strict mode otherwise it will point to the global object which in the case of the browser is the window object. This can be very problematic and another good reason to always use strict mode.

ARROW FUNCTIONS: this = <This of surrounding function (lexical this)>

Arrow functions do not get their own this keyword but inherit it from their parent function or surrounding function.

EVENT LISTENER: this = <DOM element that the listener is attached to>

Simple. This on an event listener just points to the DOM element that the event listener is attached to.

NOTE: This does not point to the function itself, and also NOT its variable environment.



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

CalcAge is the method.

Primitive vs Object (primitive vs Reference type)

```
let age = 30;
let oldAge = age;
age = 31;
console.log(age);
console.log(oldAge);
```

```
const me = {
  name: 'Jonas',
  age: 30,
};
```

```
const friend = me;
friend.age = 27;
```

```
console.log('friend:', friend);
console.log('me', me);
```

31	script.js:698
30	script.js:699
friend: {name: 'Jonas', age: 27}	script.js:709
me {name: 'Jonas', age: 27}	script.js:710

In this scenario we have some primitives to define age and new age. After age is set to a different value we can see that new age still holds the value of age because the age was set to a different value after new age was defined.

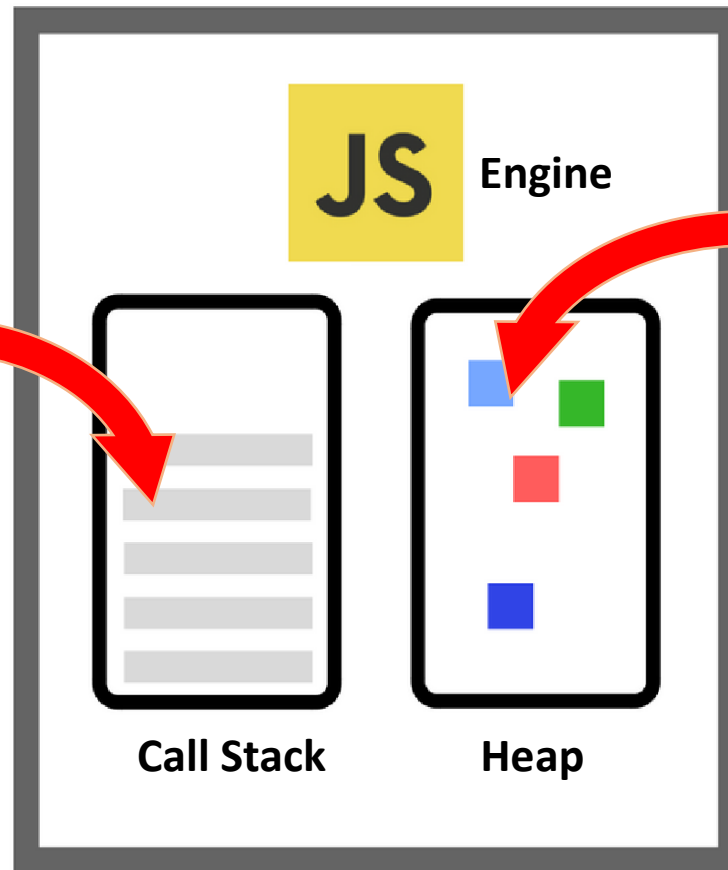
Now lets do something similar with objects. However this time the age of me is now set to the value of friend.age.

Primitive Types

PRIMITIVE
Number
String
Boolean
Undefined
Null
Symbol
BigInt

Reference Types

OBJECTS
Object Literals
Arrays
Functions
Many More...



All our reference types get stored in the memory heap

Primitives get stored in the call stack because they are part of the execution contexts where they are defined.

PRIMITIVE TYPES

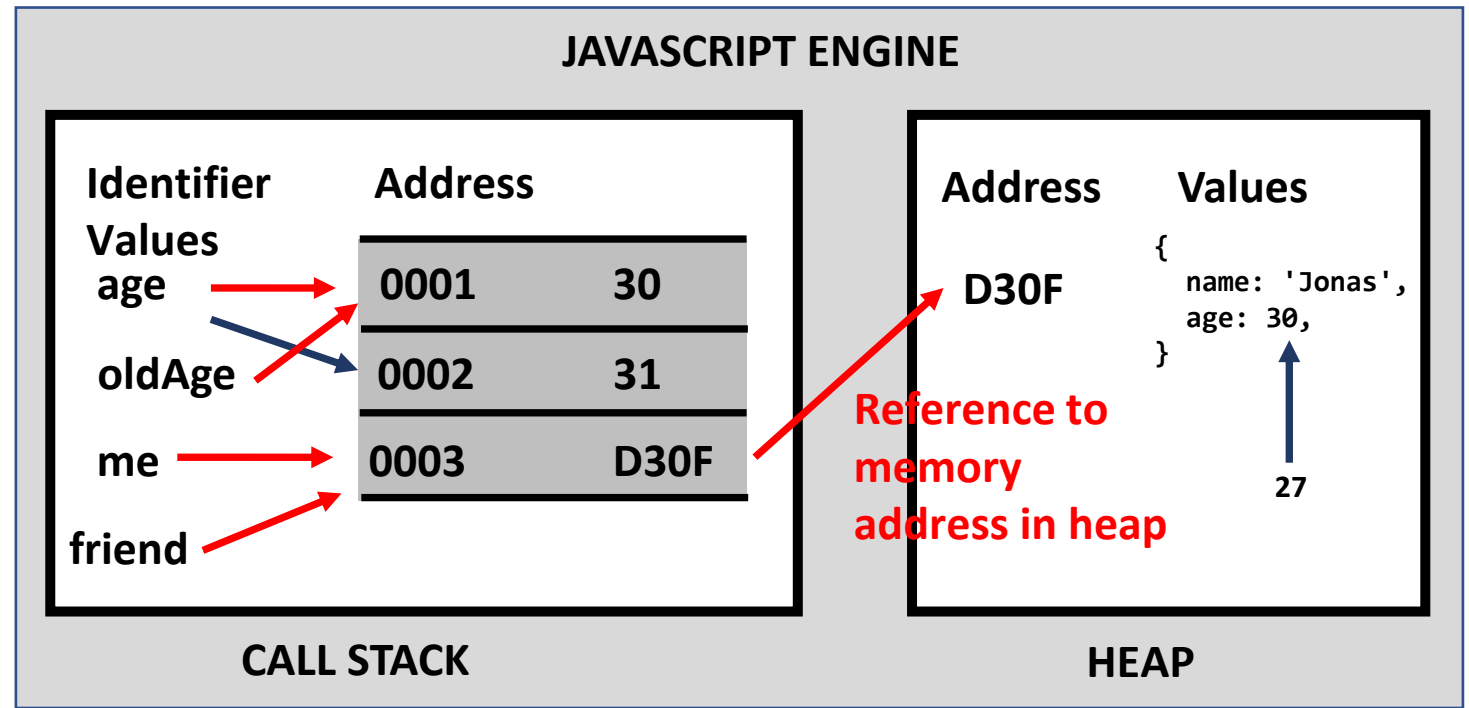
```
let age = 30;
let oldAge = age;
age = 31;
console.log(age);
console.log(oldAge);
```

REFERENCE TYPES

```
const me = {
  name: 'Jonas',
  age: 30,
};

const friend = me;
friend.age = 27;

console.log('friend:', friend);
console.log('me', me);
```



A primitive type is stored in the computer's memory in the JavaScript engine in the Call stack. The identifier points to an address which holds the value. Old Age does not create a new memory address because the value is the same. When we redefine the age value, a new memory address is created to hold the value because the value at a certain memory address is immutable.

Reference types could be too large to go in the memory stack so go in the heap where an address is created for each reference type which holds the values. Variable me in the call stack has an address which holds the address for the reference type in the heap.

When friend is created it also points to the memory of me which contains the same address to the heap where the object is stored. When friend changes a property of the object it changes in the heap, thus when the me object now has altered values for age as well.

ONLY PRIMITIVE CONST VARIABLES ARE IMMUTABLE, REFERENCE TYPES CAN BE CHANGED!

Data Structures, Modern Operators & Strings.

De-structuring an Array

```
const arr = [2, 3, 4];  
const a = arr[0];  
const b = arr[1];  
const c = arr[2];
```

An array that holds values 2, 3, and 4. We can de-structure it by declaring a const variable for each position in the array, but this is not very efficient.

```
const [x, y, z] = arr;
```

We can also de-structure an array like this, declaring all the variables in one line.

```
console.log(a, b, c);  
console.log(x, y, z);
```

Note that de-structuring an array does not destroy it, we are only reading what is in the array.

```
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'],

  openingHours: {
    thu: {
      open: 12,
      close: 22,
    },
    fri: {
      open: 11,
      close: 23,
    },
    sat: {
      open: 0, // Open 24 hours
      close: 24,
    },
  },
};
```

```
const [first, second] = restaurant.categories;
console.log(first, second);
```

Here we have an object that contains strings and arrays and we can de-structure it in a similar way.

categories: is an array within the object of restaurant. We can de-structure it by selecting only the elements we want, i.e. the first and second positions.

```
const [first, , second] = restaurant.categories;  
console.log(first, second);
```

categories: is an array within the object of restaurant. But lets say we want to take the element from the first and third position in the array. We can just skip the second position by leaving an empty space delimited by a coma. In this case we would get 'Italian Vegetarian'.

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

```
const temp = main;  
main = secondary;  
secondary = temp;  
console.log(main, secondary);
```

But lets say we want to swap the contents of `restaurant.categories` position 0 and 2 in the array so that Italian Vegetarian becomes Vegetarian Italian?

We can use main and secondary to name the variables then assign one of them to another const, temp so that we can perform a switch.

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

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

But we do not even have to assign a variable to do this, we can simply define the main and secondary with a let then swap them in an array.

```

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'],

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

  openingHours: {
    thu: {
      open: 12,
      close: 22,
    },
    fri: {
      open: 11,
      close: 23,
    },
    sat: {
      open: 0, // Open 24 hours
      close: 24,
    },
  },
};

```

Within the object of restaurant we now have a function called order that takes two parameters, starterIndex and MainIndex.

The function returns an array with the corresponding item from the position in starterMenu and the corresponding item from the position in the array MainMenu.

```
console.log(restaurant.order(2, 0));
```

```
1. (2) ['Garlic Bread', 'Pizza']
  1. 0: "Garlic Bread"
  2. 1: "Pizza"
  3. length: 2
  4. [[Prototype]]: Array(0)
```

Now when we call the function order within the restaurant object and specify the index positions it returns us an array of the two values.

```
const [starter, mainCourse] = restaurant.order(2, 0);
console.log(starter, mainCourse);
```

We can simply de-structure that to get an output of Garlic Bread Pizza.

```
const nested = [2, 4, [5, 6]];
const [i, , j] = nested;
console.log(i, j);
```

So what if we want values 2, 5 and 6 where 5 and 6 are inside a nested array?

```
2 (2) [5, 6]
  0: 5
  1: 6
  length: 2
  [[Prototype]]: Array(0)
```

We get a number then an array.

But this is not very useful. What we have to do is de-structuring within de-structuring.

```
const [i, , [j, k]] = nested;
console.log(i, j, k);
```

Now we get an output of 2 5 6.

```
const [p, q, r] = [8, 9];  
console.log(p, q, r);
```

8 9 undefined

Here we have an array with two elements of values 8 and 9. But lets say that this array is not known to us and we do not know the length and we try and de-structure it to get three elements from it. The third element is undefined because of course it does not exist.

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

8 9 1

We can assign default values in the de-structuring so that if a value is undefined then it will assume the default value. In this case the default value is 1.

De-structuring an Object

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

```
Classico Italiano {thu: {...}, fri: {...}, sat: {...}}  
    fri: {open: 11, close: 23}  
    sat: {open: 0, close: 24}  
    thu: {open: 12, close: 22}  
[[Prototype]]: Object (4) ['Italian', 'Pizzeria', 'Vegetarian', 'Organic']
```

De-structuring an object is similar to de-structuring an array except we use curly braces (because that is what we use to create an object!)

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

We can de-structure an object giving different names to the items and we get the same output as above.

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

1.[]

2.(4) ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad']

1. 0: "Focaccia"

2. 1: "Bruschetta"

3. 2: "Garlic Bread"

4. 3: "Caprese Salad"

5. length: 4

6. [[Prototype]]: Array(0)

Here we are trying to find the property of menu from the restaurant object but it does not exist so we give it a default value of an empty array, equally for property StarterMenu we rename it to starters and give it a default value of an empty array.

From the console.log output we see that 'menu' has just returned an empty array because 'menu' does not exist in the restaurant object. 'starters' has given us the contents of the property starterMenu which is an array.

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

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

Here we are manually defining the object and reassigning values for a and b.

Note that when we deconstruct an object the syntax is:

{ a, b } = object but this does not work because JavaScript expects an object declaration when a line starts with { so we have to wrap the whole structuring assignment in normal brackets.


```
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'],

  openingHours: {
    thu: {
      open: 12,
      close: 22,
    },
    fri: {
      open: 11,
      close: 23,
    },
    sat: {
      open: 0, // Open 24 hours
      close: 24,
    },
  },
};
```

OpeningHours is an object that contains nested objects of thu, fri and sat.

How do we de-structure it?

```
const { openingHours } = restaurant;  
const { fri } = openingHours;  
console.log(fri);
```

```
1. {open: 11, close: 23}  
  1. close: 23  
  2. open: 11  
  3. [[Prototype]]: Object
```

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

```
11 23
```

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

```
11 23
```

Here we destructure openingHours from the object restaurant then destructure fri from object OpeningHours.

This logs us an object with two items in it.

We can perform nested de-structuring to extract just the values of the nested object.

We can also reassign the names of the variables. For example rename open to just o and rename close to just c.

```
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'],

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

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

2) The orderDelivery function performs de-structuring based on the values passed in from the external object.

```
restaurant.orderDelivery({
  time: '23:30',
  address: 'via del Sole, 21',
  mainIndex: 2,
  starterIndex: 2,
});
```

1) Here we are passing in an object into the order delivery function that contains 2 values from the restaurant object to lookup.

Order received! Garlic Bread and Risotto will be delivered to via del Sole, 21 at 23:30

```

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'],

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

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

```

2) The orderDelivery function performs de-structuring based on the values passed in from the external object, but this time with default values.

```

restaurant.orderDelivery({
  address: 'via del Sole, 21',
  starterIndex: 2,
});

```

1) Here we are passing in an object into the order delivery but this time it is missing two parameters.

Order received! Garlic Bread and Pasta will be delivered to via del Sole, 21 at 20:00

The Spread Operator (...)

```
const arr = [7, 8, 9];  
const badNewArr = [1, 2, arr[0], arr[1], arr[2]];  
console.log(badNewArr);  
(5) [1, 2, 7, 8, 9]
```

In this scenario we want to take the values out of array, arr, and put them in badNewArr after 1 and 2. We can do this manually but this does not scale well for large arrays.

```
const goodNewArr = [1, 2, ...arr];  
console.log(goodNewArr);  
(5) [1, 2, 7, 8, 9]
```

A better solution is to use the spread operator (...array name) which is three dots followed by the array we want to insert into the new array. This takes care of arrays of any length.

```
console.log(...goodNewArr);  
1 2 7 8 9
```

We can use the spread operator to expand an array for writing it.

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

```
1.(3) [1, 2, Array(3)]
```

```
1.0: 1
```

```
2.1: 2
```

```
3.2: (3) [7, 8, 9]
```

```
4.length: 3
```

```
5.[[Prototype]]: Array(0)
```

If we do not use the spread operator then we get a nested array, i.e. an array within an array.

The spread operator unpacks the array being inserted and places the values in one by one in order as if doing it manually.

We can use the spread operator whenever we would type multiple values separated by commas.

```
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'],  
};
```

```
const newMenu = [...restaurant.mainMenu, 'Gnocchi'];  
console.log(newMenu);
```

The spread operator can be used to build a new array by appending new items to an existing array.

The spread operator is like de-structuring but it takes ALL the elements out of the array and it does not create variables.

```
const mainMenuCopy = [...restaurant.mainMenu];  
console.log(mainMenuCopy);  
(3) ['Pizza', 'Pasta', 'Risotto']
```

Another use of the spread operator is to create a copy of an array.

```
const menu = [...restaurant.starterMenu, ...restaurant.mainMenu];  
console.log(menu);  
(7) ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad', 'Pizza', 'Pasta', 'Risotto']
```

The spread operator can also be used to join two arrays into a new array.

```
const str = 'jonas';  
const letters = [...str, ' ', 's.'];  
console.log(letters);  
console.log(...str);  
(7) ['j', 'o', 'n', 'a', 's', ' ', 's.']  
j o n a s
```

**The spread operator works on all iterables:
Arrays, Strings, Maps, Sets, NOT Objects**

```
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'],

```

1) The Restaurant object has a simple function to order pasta. It takes an input of three ingredients!

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

```

2) Const ingredients is an array that is filled by the prompt pop-ups to type in the three ingredients!

```
const ingredients = [
  prompt("let's make pasta! Ingredient 1?"),
  prompt('Ingredient 2?'),
  prompt('Ingredient 3?'),
];
console.log(ingredients); (3) ['tomato', 'basil', 'bacon']

```

3) We can call order pasta function passing in our ingredients one by one or use the spread operator

```
restaurant.orderPasta(ingredients[0], ingredients[1], ingredients[2]);

```

```
restaurant.orderPasta(...ingredients);
```

here is your delicious pasta
with tomato, basil and bacon


```
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'],  
};
```

Since 2016 ES6 release the spread operator works on Objects.

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

1. {foundedIn: 1998, name: 'Classico Italiano', location: 'Via Angelo Tavanti 23, Firenze, Italy', categories: Array(4), starterMenu: Array(4), ...}

1. categories: (4) ['Italian', 'Pizzeria', 'Vegetarian', 'Organic']
2. foundedIn: 1998
3. founder: "Guiseppe"
4. location: "Via Angelo Tavanti 23, Firenze, Italy"
5. mainMenu: (3) ['Pizza', 'Pasta', 'Risotto']
6. name: "Classico Italiano"
7. starterMenu: (4) ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad']
8. [[Prototype]]: 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'],  
};
```

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

ristorante Roma
Classico Italiano

The spread operator can be used to make a deep clone of an object. We can modify the clone without changing the data in the original object.

Rest Pattern and Parameters

SPREAD operator. it is on the RIGHT side of the =

```
const arr = [1, 2, ...[3, 4]];
console.log(arr);
```

```
(4) [1, 2, 3, 4]
  0: 1
  1: 2
  2: 3
  3: 4
 length: 4
 [[Prototype]]: Array(0)
```

REST pattern. It is on the LEFT side of the =

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

```
1 2 (3) [3, 4, 5]
  0: 3
  1: 4
  2: 5
 length: 3
 [[Prototype]]: Array(0)
```

The rest pattern has the same syntax as the spread operator using the three dots, except it does the opposite.

The spread operator is used to unpack and array. In this example we are using it to unpack the array of 3 and 4. The spread operator is used on the right side of assignment operator (the =).

The rest pattern does the opposite, it is used to pack an array. In this example we are packing the unused values into a new array called others. The Rest pattern is used on the left hand side of the assignment operator (the =).

```
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'],
};
```

In this example on the Right hand side we are using the spread operator to de-structure the mainMenu and starterMenu arrays.

```
const [pizza, , rissoto, ...otherFood] = [
  ...restaurant.mainMenu,
  ...restaurant.starterMenu,];
console.log(pizza, rissoto, otherFood);
```

1. Pizza Risotto (4) ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad']

1. 0: "Focaccia"
2. 1: "Bruschetta"
3. 2: "Garlic Bread"
4. 3: "Caprese Salad"
5. length: 4
6. [[Prototype]]: Array(0)

On the left hand side we are using the rest pattern to construct an array of all other foods from mainMenu and starterMenu that are not pizza or risotto.

Note that the rest pattern should always be the last in the array otherwise JavaScript will not know how to structure the array and produce an error: Uncaught SyntaxError: Rest element must be last element

There must only be one rest pattern in one de-structuring assignment.

```
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'],

  openingHours: {
    thu: {
      open: 12,
      close: 22,
    },
    fri: {
      open: 11,
      close: 23,
    },
    sat: {
      open: 0, // Open 24 hours
      close: 24,
    },
  },
};
```

In this example on the left hand side we are using the rest pattern to construct a new object called weekdays that only has thu and fri in it.

```
1. {thu: {...}, fri: {...}}
   1. fri: {open: 11, close: 23}
   2. thu: {open: 12, close: 22}
   3. [[Prototype]]: Object
```

```
const { sat, ...weekdays } = restaurant.openingHours;
console.log(weekdays);
```

```
const add = function (...numbers) {  
  console.log(numbers);  
};
```

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

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

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

```
(2) [2, 3]
```

```
(4) [5, 3, 7, 2]
```

```
(7) [8, 2, 5, 3, 2, 1, 4]
```

Here we are using the rest pattern to pack a set of numbers into an array called numbers.

```
const add2 = function (...numbers2) {  
  let sum = 0;  
  for (let i = 0; i < numbers2.length; i++) sum += numbers2[i];  
  console.log(sum);  
};
```

```
add2(2, 3);
```

```
add2(5, 3, 7, 2);
```

```
add2(8, 2, 5, 3, 2, 1, 4);
```

```
5
```

```
17
```

```
25
```

We can iterate through each array in a loop to get the sums.

```
const x = [23, 5, 7];
```

```
add2(...x);
```

```
35
```

We can use the rest pattern to pass a known array into the add2 function.

```

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'],

  openingHours: {
    thu: {
      open: 12,
      close: 22,
    },
    fri: {
      open: 11,
      close: 23,
    },
    sat: {
      open: 0, // Open 24 hours
      close: 24,
    },
  },
};

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

```

Here we are using the rest pattern to pack a set of ingredients into an array within the orderPizza function. The pizza must have one ingredient then an array of however many ingredients.

```
restaurant.orderPizza('mushroom', 'Onions', 'olives', 'spinach');
```

Mushroom

```
(3) ['Onions', 'olives', 'spinach']  
  1. 0: "Onions"  
  2. 1: "olives"  
  3. 2: "spinach"  
  4. length: 3  
  5. [[Prototype]]: Array(0)
```

```
restaurant.orderPizza('tomatoes');
```

Tomatoes

```
[]length:  
0[[Prototype]]: Array(0)
```

We pass into the function a list of ingredients with the first becoming the mainIngredient and the rest becoming an array of otherIngredients.

If We only pass in one item into the function the otherIngredients will be an empty array.

Short Circuiting (&& and ||)

And (&&), OR (||) operators will return any datatype, not just Boolean and perform short circuiting.

```
console.log(3 || 'Jonas');  
3 (returns 3)
```

In this example we are asking 3 (number datatype) OR Jonas (string Datatype).

The or operator will return the first truthy value which is three. It is short circuiting to the first truthy value. JavaScript is not even looking at the operand of Jonas.

```
console.log('' || 'Jonas');
```

 It returns Jonas because '' is a falsy value.

```
console.log(true || 0);
```

 It returns true because 0 is a falsy value.

```
console.log(undefined || null);
```

 Undefined is a falsy value so it goes to null which is also a falsy value.

```
console.log(undefined || 0 || '' || 'Hello' || 23 || null);
```

 It returns Hello because it is the first truthy value it finds.

```
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'],  
};
```

Using a ternary statement to check if numGuests exists in an object.

```
const guests1 = restaurant.numGuests ? restaurant.numGuests : 10;  
console.log(guests1);  
  
restaurant.numGuests2 = 23;  
const guests2 = restaurant.numGuests2 ? restaurant.numGuests2 : 10;  
console.log(guests2);
```

```
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'],  
};
```

However we could just use the or operator because numGuests is falsy so it will chose 10.

```
const guests3 = restaurant.numGuests || 10;  
console.log(guests3);
```

```
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'],  
};
```

```
restaurant.numGuests2 = 0;
```

Both the ternary statement and the OR statement will now return 10 because restaurant.numGuests is set to zero which is a falsy value.

```
const guests2 = restaurant.numGuests2 ? restaurant.numGuests2 : 10;  
console.log(guests2);
```

```
const guests3 = restaurant.numGuests || 10;
```

```
console.log(0 && 'Jonas');  
Jonas
```

```
console.log(7 && 'Jonas');  
Jonas
```

```
console.log('Hello' && 23 && null && 'Jonas');  
null
```

The and operator works in the exact opposite way to the or operator and it will short circuit to the first falsy value without evaluation any other operands.

```
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'],

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

We can use an if condition to check if the orderPizza function exists and only call it if it exists.

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

We can shorten our code by using an && to replace the if condition

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

Nullish coalescing Operator (??)

```
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'],

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

restaurant.numGuests2 = 0;
const guestsCorrect = restaurant.numGuests2 ?? 10;
console.log(guestsCorrect);
```

The nullish coalescing operator takes an empty string or zero to be truthy values so using the ?? Will cause them to short circuit rather than skipping over to the next value. This solves the problem of when we want to use zero as a truthy value, i.e. number of guests is zero.

Logical Assignment Operators

```
const rest1 = {  
  name: 'Capri',  
  numGuests: 20,  
};
```

```
const rest2 = {  
  name: 'La Piazza',  
  owner: 'Giovanni Rossi',  
};
```

```
rest1.numGuests = rest1.numGuests || 10;  
rest2.numGuests = rest2.numGuests || 10;
```

```
console.log(rest1);  
console.log(rest2);
```

```
{name: 'Capri', numGuests: 20}
```

```
{name: 'La Piazza', owner: 'Giovanni Rossi', numGuests: 10}
```

```
rest1.numGuests ||= 10;  
rest2.numGuests ||= 10;
```

In this example we have two objects for restaurants with different properties and we want to find out the number of guests each restaurants can accommodate.

Rest1 is providing numGuests in the object but Rest 2 is not so we need to use a || to assign a default value.

Note how for rest2 the numGuests is set to 10 because it first looked in rest2.numGuests but did not find it so the || or operator skips to the next value of 10 and finds that to be truthy so uses it.

A cleaner way of writing this code is to use the OR assignment operator of ||=

```
const rest1 = {  
  name: 'Capri',  
  numGuests: 0,  
};
```

This works well except where numGuests is zero which is a falsy value and so would return 10.

```
const rest2 = {  
  name: 'La Piazza',  
  owner: 'Giovanni Rossi',  
};
```

```
rest1.numGuests ||= 10;  
rest2.numGuests ||= 10;
```

```
console.log(rest1);  
console.log(rest2);
```

```
{name: 'Capri', numGuests: 10}
```

```
{name: 'La Piazza', owner: 'Giovanni Rossi', numGuests: 10}
```

```
rest1.numGuests ??= 10;  
rest2.numGuests ??= 10;
```

The solution as already mentioned is the Nullish Coalescing operator! The Nullish Coalescing operator will assign a value if the value is Null or undefined.

```
console.log(rest1);  
console.log(rest2);
```

```
{name: 'Capri', numGuests: 0}
```

```
{name: 'La Piazza', owner: 'Giovanni Rossi', numGuests: 10}
```



```
const rest1 = {  
  name: 'Capri',  
  numGuests: 0,  
};
```

```
const rest2 = {  
  name: 'La Piazza',  
  owner: 'Giovanni Rossi',  
};
```

```
rest1.owner = rest1.owner && '<ANONYMOUS>';  
rest2.owner = rest2.owner && '<ANONYMOUS>';
```

```
console.log(rest1);  
console.log(rest2);
```

```
{name: 'Capri', numGuests: 0}  
{name: 'La Piazza', owner: '<ANONYMOUS>'}
```

```
rest1.owner &&= '<ANONYMOUS>';  
rest2.owner &&= '<ANONYMOUS>';
```

```
{name: 'Capri', numGuests: 0}  
{name: 'La Piazza', owner: '<ANONYMOUS>'}
```

With the && we are short circuiting to find the first falsy value. In this case owner of restaurant one does not exist in the object rest1 and so is falsy and does not return anything. In the case of rest2 the owner is present and therefore true so it skips over the first value of rest.owner and lands on the last option of <ANONYMOUS>

A cleaner way of writing this code is to use the AND assignment operator of &&=

Looping Arrays: The for of loop

```
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'],  
};
```

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

```
for (const item of menu) console.log(item);
```

Focaccia	script.js:770
Bruschetta	script.js:770
Garlic Bread	script.js:770
Caprese Salad	script.js:770
Pizza	script.js:770
Pasta	script.js:770
Risotto	script.js:770

We create a variable called menu from starterMenu and mainMenu.

We want to loop through all menu items and the traditional method would be to use a for loop setting parameters for i (`let i = 0; i < menu.length; i++`) but we can also use the for of loop.

```
for (const item of menu.entries()) {  
  console.log(item);  
}
```

```
(2) [0, 'Focaccia']  
    0: 0  
    1: "Focaccia"length:  
    2[[Prototype]]: Array(0)  
(2) [1, 'Bruschetta']  
(2) [2, 'Garlic Bread']  
(2) [3, 'Caprese Salad']  
(2) [4, 'Pizza']  
(2) [5, 'Pasta']  
(2) [6, 'Risotto']
```

To get an output of the index and the item we need to use the entries function.

The entries function actually created an array out of each item that contains the index in position 0 then the item in position 1

We can use the entries function to create a more beautiful output like you would see in a menu but better still would be to use de-structuring.

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

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

```
1:Focaccias  
2:Bruschetta  
3:Garlic Bread  
4:Caprese Salad  
5:Pizza  
6:Pasta  
7:Risotto
```

ES6 Enhanced object literals

```
const openingHours = {  
  thu: {  
    open: 12,  
    close: 22,  
  },  
  fri: {  
    open: 11,  
    close: 23,  
  },  
  sat: {  
    open: 0, // Open 24 hours  
    close: 24,  
  },  
};
```

Here the opening hours is a separate variable object to the restaurant variable 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'],  
  
  // before ES6 Enhanced object literals  
  openingHours: openingHours,  
};
```

Before ES6 enhancements to include openingHours in the restaurant object it would be written like this.

```
const openingHours = {
  thu: {
    open: 12,
    close: 22,
  },
  fri: {
    open: 11,
    close: 23,
  },
  sat: {
    open: 0, // Open 24 hours
    close: 24,
  },
};
```

```
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'],
```

```
  // After ES6 Enhanced object literals
  openingHours,
};
```

after ES6 enhancements we only need to write the name of the object variable to include it.

```
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'],
```

Pre ES6 Enhancements the syntax is property: method i.e. function()

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

ES6 Enhancements the syntax is simpler. We remove the colon and the word function

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

```
  orderDelivery: function ({ starterIndex, mainIndex, time, address }) {
    console.log(
      `Order recieved! ${this.starterMenu[starterIndex]} and
      ${this.mainMenu[mainIndex]} will be delivered to ${address} at ${time}`
    );
  },
};
```

```
const weekdays = ['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun'];
```

```
const openingHours = {  
  [weekdays[3]]: {  
    open: 12,  
    close: 22,  
  },  
  [weekdays[4]]: {  
    open: 11,  
    close: 23,  
  },  
  [`day-${2 + 4}`]: {  
    open: 0, // Open 24 hours  
    close: 12 + 12,  
  },  
};
```

```
console.log(openingHours);
```

ES& enhancement include being able to compute properties in an object.

```
1. {thu: {...}, fri: {...}, day-6: {...}}  
   1. day-6: {open: 0, close: 24}  
   2. fri: {open: 11, close: 23}  
   3. thu: {open: 12, close: 22}  
   4. [[Prototype]]: Object
```

Optional Chaining (?)

```
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'],

  openingHours: {
    thu: {
      open: 12,
      close: 22,
    },
    fri: {
      open: 11,
      close: 23,
    },
    sat: {
      open: 0, // Open 24 hours
      close: 24,
    },
  },
};

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

Lets say that restaurant is an API feed and we want to get openingHours for a Monday on the object.

Restaurant.openingHours.mon does not exist so this will return undefined.

Restaurant.openingHours.mon.open will return a javascript error of Uncaught TypeError: Cannot read properties of undefined (reading 'open') because it does not exist.


```

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'],

  openingHours: {
    thu: {
      open: 12,
      close: 22,
    },
    fri: {
      open: 11,
      close: 23,
    },
    sat: {
      open: 0, // Open 24 hours
      close: 24,
    },
  },
};

```

I could use an If to check if property exists before calling it but this is a lot of code for something really simple, especially if we write it for each day of the week!

```

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

if (restaurant.openingHours.thu) console.log(restaurant.openingHours.thu.open);

```

```

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'],

  openingHours: {
    thu: {
      open: 12,
      close: 22,
    },
    fri: {
      open: 11,
      close: 23,
    },
    sat: {
      open: 0, // Open 24 hours
      close: 24,
    },
  },
};

```

Optional chaining is a feature introduced in ES2020 and allows us to use a question mark as a nullish operator.

Everything before the question mark is queried as existing, i.e. not null and it immediately returns undefined if what is before the question mark does not exist.

If what is before the question mark is not null then it reads what is after the question mark and returns the values.

```

console.log(restaurant.openingHours.mon?.open);
console.log(restaurant.openingHours?.mon?.open);
console.log(restaurant.openingHours.thu?.open);

```

```
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'],
```

On mon we are open at Closed

On tue we are open at Closed

On wed we are open at Closed

On thu we are open at 12

On fri we are open at 11

On sat we are open at 0

On sun we are open at Closed

```
  openingHours: {
    thu: {
      open: 12,
      close: 22,
    },
    fri: {
      open: 11,
      close: 23,
    },
    sat: {
      open: 0, // Open 24 hours
      close: 24,
    },
  },
};
```

A practical solution using optional chaining is to use a for of loop on an array of days and get the opening hour for each day from the restaurant object.

The null coalescing operator can be used to return opening hour or 'closed' for each pass through the loop.

```
const days = ['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun'];

for (const day of days) {
  const open = restaurant.openingHours?.[day]?.open ?? 'Closed';
  console.log(`On ${day} we are open at ${open}`);
}
```

```

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'],

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

  orderDelivery: function ({
    starterIndex = 1,
    mainIndex = 1,
    time = '20:00',
    address,
  }) {
    console.log(
      `Order recieved! ${this.starterMenu[starterIndex]} and  

      ${this.mainMenu[mainIndex]} will be delivered to ${address} at ${time}`
    );
  },
};

```

The optional chaining operator can be used to check if a method exists before calling it.

```

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

```

We can use optional chaining to check if an object inside an array exists and use the nullish coalescing operator to return something else if it does not exist.

This is a really useful short hand way of writing what would be a 2 line if else statement.

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

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

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

Looping objects: Object keys, Values & Entries

```
const weekdays = ['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun'];
const openingHours = {
  [weekdays[3]]: {
    open: 12,
    close: 22,
  },
  [weekdays[4]]: {
    open: 11,
    close: 23,
  },
  [weekdays[5]]: {
    open: 0, // Open 24 hours
    close: 24,
  },
};
```

```
const properties = Object.keys(openingHours);
console.log(properties);
```

```
▼ (3) ['thu', 'fri', 'sat'] ⓘ
  0: "thu"
  1: "fri"
  2: "sat"
  length: 3
  ► [[Prototype]]: Array(0)
```

PROPERTY NAMES

We can look at the object keys for opening hours

```
let openStr = `We are open on ${properties.length} days: `;

for (const day of Object.keys(openingHours)) {
  openStr += `${day}, `;
}
console.log(openStr);
```

We are open on 3 days: thu, fri, sat,

```
const values = Object.values(openingHours);
console.log(values);

const entries = Object.entries(openingHours);
console.log(entries);
```

Now we can look at the values of the object.

The object entries will put each item into an array.

We can iterate over the entries arrays using de-structuring in the for each loop to break out the open and close values of the object.

```
for (const [key, { open, close }] of entries) {
  console.log(`On ${key} we open at ${open} and close at ${close}`);
}
```

On thu we open at 12 and close at 22

On fri we open at 11 and close at 23

On sat we open at 0 and close at 24

We can loop over the object.keys adding to the string with each iteration of the loop.

```
▼ (3) [{...}, {...}, {...}] ⓘ
  ▼ 0:
    close: 22
    open: 12
    ► [[Prototype]]: Object
  ▼ 1:
    close: 23
    open: 11
    ► [[Prototype]]: Object
  ▼ 2:
    close: 24
    open: 0
    ► [[Prototype]]: Object
  length: 3
  ► [[Prototype]]: Array(0)
```

Sets

A set is similar to an array except it cannot contain duplicate values. Above we are manually defining the set but it has three duplicates of pizza. Note that when it is console logged does not show the duplicates. Like an array its elements are itterables, but unlike an array it's order is irrelevant.

```
console.log(ordersSet.size);
```

We can determine how many items are in a set with the property of size. Note how this differs from an array where we would use length.

```
console.log(ordersSet.has('Pizza'));  
console.log(ordersSet.has('Bread'));
```

We can check if a value is in the set. This will return a true or false value.

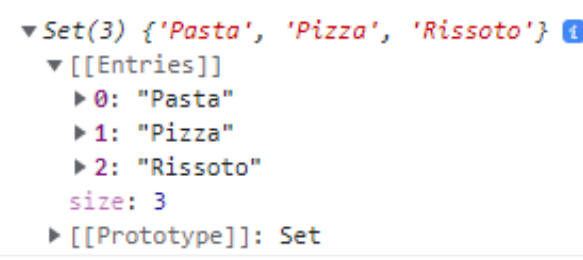
```
ordersSet.add('Garlic Bread');  
ordersSet.add('Garlic Bread');  
console.log(ordersSet);
```

We can add values to a set. Note that only one Garlic Bread will be added to the set because a set cannot contain duplicates.

```
ordersSet.delete('Risotto');  
console.log(ordersSet);
```

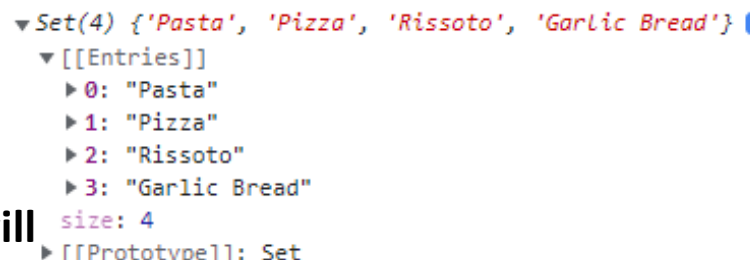
We can delete values from a set.

```
const ordersSet = new Set([  
  'Pasta',  
  'Pizza',  
  'Pizza',  
  'Rissoto',  
  'Pasta',  
  'Pizza',  
]);
```

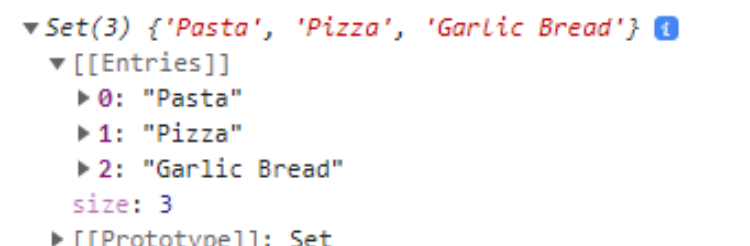


```
Set(3) {'Pasta', 'Pizza', 'Rissoto'}  
  [[Entries]]  
    0: "Pasta"  
    1: "Pizza"  
    2: "Rissoto"  
  size: 3  
  [[Prototype]]: Set
```

```
console.log(ordersSet);
```



```
Set(4) {'Pasta', 'Pizza', 'Rissoto', 'Garlic Bread'}  
  [[Entries]]  
    0: "Pasta"  
    1: "Pizza"  
    2: "Rissoto"  
    3: "Garlic Bread"  
  size: 4  
  [[Prototype]]: Set
```



```
Set(3) {'Pasta', 'Pizza', 'Garlic Bread'}  
  [[Entries]]  
    0: "Pasta"  
    1: "Pizza"  
    2: "Garlic Bread"  
  size: 3  
  [[Prototype]]: Set
```



```
console.log(ordersSet[0]);  
undefined
```

How can we get values out of a set? Like in an array by selecting the index? No. Sets do not have indexes. As sets contain unique values all we really want to do is check if a value exists in the set using the has method. For all other applications we would use an array or an object.

```
ordersSet.clear();  
console.log(ordersSet);
```

We can delete all values from a set.

```
▼ Set(0) {size: 0} ⓘ  
  ▼ [[Entries]]  
    No properties  
  size: 0  
  ► [[Prototype]]: Set
```

```
for (const order of ordersSet) console.log(order);
```

Sets are iterable so we can loop over the set.

```
Pasta  
Pizza  
Garlic Bread
```

```
const staff = ['Waiter', 'Chef', 'Waiter', 'Manager', 'Chef', 'Waiter'];  
const staffUnique = [...new Set(staff)];  
console.log(staffUnique);
```

```
▼ (3) ['Waiter', 'Chef', 'Manager'] ⓘ  
  0: "Waiter"  
  1: "Chef"  
  2: "Manager"  
  length: 3  
  ► [[Prototype]]: Array(0)
```

A set (along with the spread operator) is a good tool for converting an array of non unique values into an array of unique values, i.e. a restaurant has an array of staff, multiple people in the same position.

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

We can use a set to find the number of unique positions in an array.

```
console.log(new Set('jonasschmedtmann').size);
```

Or to find the number of letter in a string.

Maps

Maps are data structures to map values to keys but unlike arrays where keys have to be strings, map keys can have any data type such as objects or even arrays.

```
const rest = new Map();
rest.set('name', 'Classico Italiano');
rest.set(1, 'Firenze, Italy');
console.log(rest.set(2, 'Lisbon, Portugal'));
```

We define a new empty map called 'rest' and can use the set method to assign key value pairs to the map where the key can be any data type, for example 'name' is a string, '1' is an integer.

```
rest
  .set('categories', ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'])
  .set('open', 11)
  .set('close', 23)
  .set(true, 'We are open :D')
  .set(false, 'We are closed:(');
```

We can continue adding in key value pairs into the map where 'categories' is a string and the values are an array. Open and close are keys of strings. We can even add boolean keys that have string values.

```
console.log(rest.get('name'));
console.log(rest.get(true));
console.log(rest.get(1));
```

Classico Italiano

We are open :D

Firenze, Italy

```
▼ Map(3) {'name' => 'Classico Italiano', 1 => 'Firenze, Italy', 2 => 'Lisbon, Portugal'} ⓘ
```

```
▼ [[Entries]]
```

```
▶ 0: {"name" => "Classico Italiano"}
```

```
▶ 1: {1 => "Firenze, Italy"}
```

```
▶ 2: {2 => "Lisbon, Portugal"}
```

```
▶ 3: {"categories" => Array(4)}
```

```
▶ 4: {"open" => 11}
```

```
▶ 5: {"close" => 23}
```

```
▶ 6: {true => "We are open :D"}
```

```
▶ 7: {false => "We are closed:("}
```

```
size: 8
```

We are open :D

```
const time = 21;  
console.log(rest.get(time > rest.get('open') && time < rest.get('close')));
```

We can use the has property to check if a key exists.
This will return true or false.

```
console.log(rest.has('catagories'));
```

```
rest.delete(2);  
console.log(rest);
```

We can also delete key value pairs from a map by specifying the key.

```
▼ Map(7) {'name' => 'Classico Italiano', 1 => 'Firenze, Italy', 'categories' => Array(4), 'open' => 1  
1, 'close' => 23, ...} ⓘ  
  ▼ [[Entries]]  
    ▶ 0: {"name" => "Classico Italiano"}  
    ▶ 1: {1 => "Firenze, Italy"}  
    ▶ 2: {"categories" => Array(4)}  
    ▶ 3: {"open" => 11}  
    ▶ 4: {"close" => 23}  
    ▶ 5: {true => "We are open :D"}  
    ▶ 6: {false => "We are closed:("}  
    size: 7  
  ▶ [[Prototype]]: Map
```

We can get the size of the map, in this case 7 because there are 7 key value pairs in the map.

```
console.log(rest.size);
```

```
rest.clear();  
console.log(rest);
```

We can also empty a map with a clear method.

```
▼ Map(0) {size: 0} ⓘ  
  ▼ [[Entries]]  
    No properties  
  size: 0  
  ▶ [[Prototype]]: Map
```

```

▼ Map(8) {'name' => 'Classico Italiano', 1 => 'Firenze, Italy', 'categories' => Array(4), 'open' => 1
  1, 'close' => 23, ...} ⓘ
  ▼ [[Entries]]
    ▶ 0: {"name" => "Classico Italiano"}
    ▶ 1: {1 => "Firenze, Italy"}
    ▶ 2: {"categories" => Array(4)}
    ▶ 3: {"open" => 11}
    ▶ 4: {"close" => 23}
    ▶ 5: {true => "We are open :D"}
    ▶ 6: {false => "We are closed:("}
    ▼ 7: {Array(2) => "test"}
      ▶ key: (2) [1, 2]
      value: "test"
    size: 8
  ▶ [[Prototype]]: Map

```

```

rest.set([1, 2], 'test');
console.log(rest);

```

```

console.log(rest.get([1, 2]));

```

We can also set a key as an array in a map! But if we try and get the value with get and put in the array values it will return undefined.

```

const arr = [1, 2];
rest.set(arr, 'test');
console.log(rest);

```

However we can define the array as a variable and set the values then use the name of the array in the map set method.

```

console.log(rest.get(arr));

```

Now we can set the key name as the array and call the value. In this case it returns test.

```

rest.set(document.querySelector('h1'), 'Heading');
console.log(rest);

```

We can also set a key as an object. In this case we are using query select to get the object properties of the h1 tag.

Maps: Iteration

```
const question = new Map([
  ['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);
```

A more streamlined method is to define the key value pairs as an array with the first item in the array being the key and the second item being the value.

Using set to define the key value pairs in a map can be cumbersome if the map has a lot of data in it.

```
Map(7) {'question' => 'what is the best programming language in the world', 1 => 'C', 2 => 'java', 3 => 'javascript', 'correct' => 3, ...} ⓘ
  ▼ [[Entries]]
    ▼ 0: {"question" => "what is the best programming language in the world"}
      key: "question"
      value: "what is the best programming language in the world"
    ▼ 1: {1 => "C"}
      key: 1
      value: "C"
    ▼ 2: {2 => "java"}
      key: 2
      value: "java"
    ▼ 3: {3 => "javascript"}
      key: 3
      value: "javascript"
    ▼ 4: {"correct" => 3}
      key: "correct"
      value: 3
    ▼ 5: {true => "correct🏆"}
      key: true
      value: "correct🏆"
    ▼ 6: {false => "try again!"}
      key: false
      value: "try again!"
  size: 7
  ► [[Prototype]]: Map
```

```
const weekdays = ['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun'];
const openingHours = {
  [weekdays[3]]: {
    open: 12,
    close: 22,
  },
  [weekdays[4]]: {
    open: 11,
    close: 23,
  },
  [weekdays[5]]: {
    open: 0, // Open 24 hours
    close: 24,
  },
};
console.log(Object.entries(openingHours));
```

The map data structure shown previously is similar to an object.

```
const hoursMap = new Map(Object.entries(openingHours));
console.log(hoursMap);
```

Which is why an Object can be converted to a map.

```
▼ (3) [Array(2), Array(2), Array(2)] ⓘ
  ▼ 0: Array(2)
    0: "thu"
    ▼ 1:
      close: 22
      open: 12
      ► [[Prototype]]: Object
      length: 2
      ► [[Prototype]]: Array(0)
  ► 1: (2) ['fri', {...}]
  ► 2: (2) ['sat', {...}]
  length: 3
  ► [[Prototype]]: Array(0)
```

```
▼ Map(3) {'thu' => {...}, 'fri' => {...}, 'sat' => {...}} ⓘ
  ▼ [[Entries]]
    ▼ 0: {"thu" => Object}
      key: "thu"
      ► value: {open: 12, close: 22}
    ► 1: {"fri" => Object}
    ► 2: {"sat" => Object}
  size: 3
  ► [[Prototype]]: Map
```

```
console.log(question.get('question'));  
for (const [key, value] of question) {  
  if (typeof key === 'number') console.log(`Answer ${key}: ${value}`);  
}
```

what is the best programming language in the world?

Answer 1: C

Answer 2: java

Answer 3: javascript

A map is iterable. In this example we can iterate through the array and if the key datatype is a number we can console log key and value.

```
const answer = Number(prompt('your answer'));  
console.log(answer);
```

We can use a prompt to get a value of 1, 2 or 3 from the user and store it in variable 'answer'.

```
console.log(question.get(question.get('correct') === answer));
```

We can use the get method for the map to get the value of the correct answer. Using the power of Boolean values we can console log the correct or incorrect messages.

```
console.log([...question]);
```

We can convert a map back into an array, a multidimensional array, using the spread operator.

The below methods are also available and can be used to dump the keys or values into an array with the spread operator.

```
console.log(question.entries());
```

```
console.log([...question.keys()]);
```

```
console.log([...question.values()]);
```

```
▼ (7) [Array(2), Array(2), Array(2), Array(2), Array(2), Array(2), Array(2)] ⓘ  
  ▶ 0: (2) ['question', 'what is the best programming language in the world?']  
  ▼ 1: Array(2)  
    0: 1  
    1: "C"  
    length: 2  
    ▶ [[Prototype]]: Array(0)  
  ▶ 2: (2) [2, 'java']  
  ▶ 3: (2) [3, 'javascript']  
  ▶ 4: (2) ['correct', 3]  
  ▶ 5: (2) [true, 'correct🏆']  
  ▶ 6: (2) [false, 'try again!']  
    length: 7  
  ▶ [[Prototype]]: Array(0)
```

```
▼ MapIterator {'question' => 'what is the best programming language in the world?', 1 => 'C', 2 => 'java', 3 => 'javascript', 'correct' => 3, ...} ⓘ  
  ▼ [[Entries]]  
    ▶ 0: {"question" => "what is the best programming language in the world?"}  
    ▶ 1: {1 => "C"}  
    ▶ 2: {2 => "java"}  
    ▶ 3: {3 => "javascript"}  
    ▶ 4: {"correct" => 3}  
    ▶ 5: {true => "correct🏆"}  
    ▶ 6: {false => "try again!"}  
  ▶ [[Prototype]]: Map Iterator  
    [[Iterator]]: {false => "try again!"}  
    [[IteratorIndex]]: 0  
    [[IteratorKind]]: "entries"
```

```
▶ (7) ['question', 1, 2, 3, 'correct', true, false]
```

[script.js:1276](#)

```
▶ (7) ['what is the best programming language in the world?', 'C', 'java', 'javascript', 3, 'correct🏆', 'try again!']
```

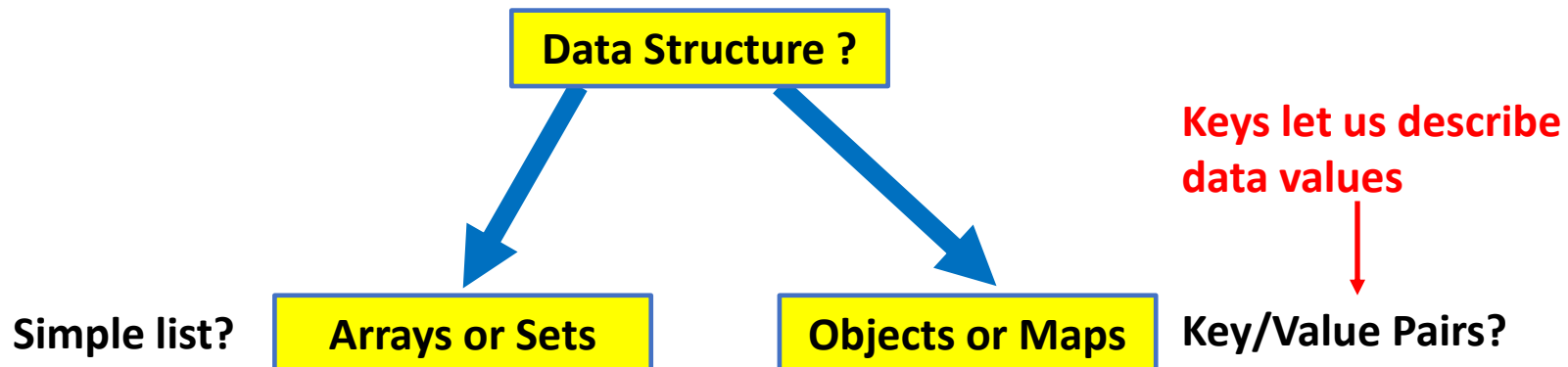
[script.js:1277](#)

What data structure to use? Array, Object, Set, Map

- Data Sources:**
- 1 **From the program itself:** Data written directly in source code (eg. Status messages)
 - 2 **From the UI:** Data inputted by the user or from the DOM (eg. Tasks in a to-do app)
 - 3 **From External Sources:** Data fetched for example from a web API (eg recipe object)

Application Programming Interface

For example we can use a web API to get weather data from a third-party weather provider, or movies currently showing at a cinema etc.. When we get the data we store it in a data structure.



Object

Array

Object

```
{
  "count": 3,
  "recipes": [
    {
      "publisher": "101 Cookbooks",
      "title": "Best Pizza Dough Ever",
      "source_url": "http://www.101cookbooks.com/archiv",
      "recipe_id": "47740",
      "image_url": "http://forkify-api.herokuapp.com/in",
      "social_rank": 100,
      "publisher_url": "http://www.101cookbooks.com"
    },
    {
      "publisher": "The Pioneer Woman",
      "title": "Deep Dish Fruit Pizza",
      "source_url": "http://thepioneerwoman.com/cooking",
      "recipe_id": "40950",
      "image_url": "http://forkify-api.herokuapp.com/in",
      "social_rank": 100,
      "publisher_url": "http://thepioneerwoman.com"
    },
    {
      "publisher": "Closet Cooking",
      "title": "Pizza Dip",
      "source_url": "http://www.closetcooking.com/2011/",
      "recipe_id": "35477",
      "image_url": "http://forkify-api.herokuapp.com/in",
      "social_rank": 99.999999999999994,
      "publisher_url": "http://closetcooking.com"
    }
  ]
}
```

Data from a web API usually comes in a special data format called JSON which is similar to the JavaScript object format. The JSON recipe objects below contains an array of recopies. Each value of the array is an object with key value pairs.

Array

```
tasks = ['Code', 'Eat', 'Code'];  
// ["Code", "Eat", "Code"]
```

Use when you need **ordered** list of values (might contain duplicates)

Use when you need to **manipulate** data.

Set

```
tasks = new Set(['Code', 'Eat', 'Code']);  
// {"Code", "Eat"}
```

Use when you need to work with **unique** values

Use when **high performance** is *really* important

Use to remove **duplicates** from arrays

Object

```
task = {  
  task: 'Code',  
  date: 'today',  
  repeat: true  
};
```

More “traditional” key/value store (“abused” objects)

Easier to write and access objects with . And []

Use when you need to include **functions** (methods)

Use when working with JSON (can convert to map)

Map

```
task = new Map([  
  ['task', 'Code'],  
  ['date', 'today'],  
  [false, 'Start coding!']  
]);
```

Better performance

Keys can have **any** data type

Easy to iterate

Easy to compute size

Use when you simply need to map key to values

Use when you need keys that are not strings

Working with Strings – part 1

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

Two variables are defined with values set as strings.

```
A console.log(plane[0]);  
3 console.log(plane[1]);  
2 console.log(plane[2]);  
B console.log('B737'[0]);
```

A string can be broken down by position.

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

A string has properties of length

```
6 console.log(airline.indexOf('r'));  
10 console.log(airline.lastIndexOf('r'));  
8 console.log(airline.indexOf('Portugal'));
```

IndexOf finds the first occurrence of. In this case the letter r first appears in the seventh position. LastIndexof finds the last occurrence of. WE can even use this to find whole words, In this case 'Portugal' starts at position 8 or starts at the 9th character of the string.

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

The slice method can be used to chop a string starting at the indicated position. In this case splicing at position 4 chop at the 5th character. We can also use splice to specify a starting and ending position.

```
console.log(airline.slice(0, airline.indexOf(' ')));
```

 TAP

Lets say that we want to get the first word but we do not know how many characters long it is – we can use indexOf to find the first space.

```
console.log(airline.slice(airline.lastIndexOf(' ') + 1));
```

 Portugal

To get the last word of the string where we do not know where it starts and begins we just need to slice it one position after the last space.

```
console.log(airline.slice(airline.indexOf(' ') + 1, airline.lastIndexOf(' ')));
```

 Air

We can even combine the two to get the middle words from a string.

```
console.log(airline.slice(-2));
```

 al

Slice and be used to extract the last two characters from a string

```
console.log(airline.slice(1, -2));
```

AP Air Portug

And to extract from position 1 until before the last two characters

```
const checkMiddleSeat = function (seat) {  
  const s = seat.slice(-1);  
  if (s === 'B' || s === 'E') console.log('You got the middle seat 😊');  
  else console.log('You got lucky 😊');  
};
```

```
checkMiddleSeat('11B'); You got the middle seat 😊  
checkMiddleSeat('23C'); You got lucky 😊  
checkMiddleSeat('3E'); You got the middle seat 😊
```

An airline has seats that are numbered with row number then letter for the column or position. I.e. 11A, 11B, 11C, 11D, 11E, 11F. Where B and E are middle seats.

Strings are just primitives so why are we able to call a method such as slice on a primitive? Well behind the scenes, JavaScript is will convert a string primitive to a string object then call the method on the string object. This is called boxing.

```
console.log(new String('jonas'));
```

Here we can see that the string looks a little bit like an object.

```
▼ String {'jonas'} 1  
  0: "j"  
  1: "o"  
  2: "n"  
  3: "a"  
  4: "s"  
  length: 5  
  ► [[Prototype]]: String  
    [[PrimitiveValue]]: "jonas"
```

When JavaScript has finished performing a method on a primitive it will convert it back from an object to a string in the background

Working with Strings – part 2

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

We can change the case of a string to lowercase or uppercase.

```
const passenger = 'jOnAS';  
const passengerLower = passenger.toLowerCase();  
const passengerCorrect = passengerLower[0].toUpperCase() + passengerLower.slice(1);  
console.log(passengerCorrect);
```

Lets say that the passenger has entered their name in incorrectly at the time of booking. We can use a combination of `toLowerCase` and `splice` to correct this.

```
const email = 'hello@jonas.io';  
const loginEmail = ' Hello@jonas.io \n';  
  
const lowerEmail = loginEmail.toLowerCase();  
const trimmedEmail = lowerEmail.trim();  
console.log(trimmedEmail);  
  
const normalisedEmail = loginEmail.toLowerCase().trim();  
console.log(normalisedEmail);
```

This can be performed with one line of code by calling the `trim` method on the `toLowerCase`. Trim removes whitespace.

Lets say that the passenger has entered their login email incorrectly at the time of booking by adding a space at the beginning, a capital letter and a return at the end. We can use a combination of `toLowerCase` and `trim` to correct this.

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

Replace can be used to switch out characters or words in a string. In this example we are replacing the coma with a dot and the pound sign with a dollar sign.

```
const announcement = 'All passengers come to barding door 23. Boarding door 23!';  
console.log(announcement.replace('door', 'gate'));
```

All passengers come to barding gate 23. Boarding door 23!

Notice how replace only replaces the first instance. Coming soon to javascript will be a replaceAll method – Now working – added in 2022 release.

```
console.log(announcement.replaceAll('door', 'gate'));
```

All passengers come to barding gate 23. Boarding gate 23!

```
console.log(announcement.replace(/door/g, 'gate'));
```

We can also use a regular expression to replace door. In this example we put the word door between forward slashes instead of quotes and a g for global after the last forward slash.

All passengers come to barding gate 23. Boarding gate 23!


```
const airplane = 'A320neo';
console.log(airplane.includes('A320')); true
console.log(airplane.includes('Boeing')); false
```

There are several methods that work with Booleans such as includes, startsWith and endsWith.

```
const aircraft = 'Airbus A320neo';
console.log(aircraft.startsWith('Air')); true
console.log(aircraft.startsWith('Ab')); false
```

```
if (aircraft.startsWith('Airbus') && aircraft.endsWith('neo')) {
  console.log('Part of the NEW Airbus Family');
}
```

```
const checkBaggage = function (items) {
  const baggage = items.toLowerCase();
  if (baggage.includes('knife') || baggage.includes('gun')) {
    console.log('You are not allowed on board');
  } else {
    console.log('Welcome aboard');
  }
};
```

We can use includes in a function to check a string. Note that we put the string to lowercase first to eliminate the problem of user capitalisation errors.

```
checkBaggage('I have a laptop, some Food and a pocket Knife');
checkBaggage('I have socks and a camera');
checkBaggage('Got some snacks and a gun for protection');
```

You are not allowed on board

Welcome aboard

You are not allowed on board

Working with Strings – part 3

```
console.log('a+very+nice+string'.split('+'));  
console.log('Jonas Schmedtmann'.split(' '));
```

```
▶ (4) ['a', 'very', 'nice', 'string']
```

```
▶ (2) ['Jonas', 'Schmedtmann']
```

The split method can be used to split a string into an array of components when we specify what character we want to use to split.

```
const [firstName, LastName] = 'Jonas Schmedtmann'.split(' ');  
const newName = ['Mr.', firstName,  
LastName.toUpperCase()].join(' ');  
console.log(newName);
```

```
Mr. Jonas SCHMEDTMANN
```

We can use de-structuring to put the split string into an array then build a new string based on the values in that array. The opposite of split is join where we again specify what we want to use in the join.

```
const capitalizedName = function (name) {  
  const names = name.split(' ');  
  const namesUpper = [];
```

We can also write a function to capitalise the first letter of a persons names.

```
  for (const n of names) {  
    namesUpper.push(n[0].toUpperCase() + n.slice(1));  
  }  
  console.log(namesUpper.join(' '));  
};
```

```
capitalizedName('jessica ann smith davis');  
capitalizedName('jonas schmedtmann');
```

```
Jessica Ann Smith Davies
```

```
Jonas Schmedtmann
```

```
const capitalizedName = function (name) {
  const names = name.split(' ');
  const namesUpper = [];

  for (const n of names) {
    // namesUpper.push(n[0].toUpperCase() + n.slice(1));
    namesUpper.push(n.replace(n[0], n[0].toUpperCase()));
  }
  console.log(namesUpper.join(' '));
};
```

We could use slice or replace to change the first letter of each name to upper case

```
capitalizedName('jessica ann smith davies');
capitalizedName('jonas schmedtmann');
```

```
Jessica Ann Smith Davies
Jonas Schmedtmann
```

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

```
+++++Go to Gate 23!
+++++Jonas
+++++Go to Gate 23!+++++
+++++Jonas+++++
```

padStart and **padEnd** can be used to add padding to the start and end of a string where the first number is the desired length of the string and the second parameter is the characters we want to pad with.

```
const maskCreditCard = function (number) {
  const str = number + '';
  const lastFour = str.slice(-4);
  return lastFour.padStart(str.length, '*');
};
```

A real world application of padding would be to mask all but the last four digits of a credit card number.

```
console.log(maskCreditCard('4321765469762144'));
console.log(maskCreditCard('432176546976217886544'));
console.log(maskCreditCard('43217654697625675817886544'));
```

```
*****2144
```

```
*****6544
```

```
*****6544
```

```
const messageRepeat = 'Bad Wather... All departures delayed... ';
console.log(messageRepeat.repeat(5));
```

```
Bad Wather... All departures delayed... Bad Wather... All departures delayed... Bad Wather... All departures delayed... Bad Wather... All departures delayed... Bad Wather... All departures delayed...
```

The repeat method will just repeat a string as many times as we specify. In this case 5.

```
const planesInLine = function (n) {
  console.log(`There are ${n} planes in line ${'✈️'.repeat(n)}`);
};
```

```
planesInLine(4);    There are 4 planes in line ✈️✈️✈️✈️
```

```
planesInLine(10);   There are 10 planes in line ✈️✈️✈️✈️✈️✈️✈️✈️✈️✈️
```

The repeat method can be used in a function.

All the different methods for strings can be found in the left hand pane of this page

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/repeat

A Closer look at functions

```
const bookings = [];
```

```
const createBooking = function (  
  flightNum,  
  numPassengers = 1,  
  price = 199 * numPassengers  
) {  
  // ES5 method of setting default parameters  
  // numPassengers = numPassengers || 1;  
  // price = price || 199;
```

```
  const booking = {  
    flightNum,  
    numPassengers,  
    price,  
  };
```

```
  console.log(booking);  
  bookings.push(booking);  
};
```

```
createBooking('LH123');  
createBooking('LH123', 2, 800);  
createBooking('LH123', 2);  
createBooking('LH123', 5);  
createBooking('LH123', undefined, 1000);
```

Default Parameters

Here is a simple function that creates a booking object and pushes it into an array.

```
▼ (5) [{...}, {...}, {...}, {...}, {...}] ⓘ  
  ▶ 0: {flightNum: 'LH123', numPassengers: 1, price: 199}  
  ▶ 1: {flightNum: 'LH123', numPassengers: 2, price: 800}  
  ▶ 2: {flightNum: 'LH123', numPassengers: 2, price: 398}  
  ▶ 3: {flightNum: 'LH123', numPassengers: 5, price: 995}  
  ▶ 4: {flightNum: 'LH123', numPassengers: 1, price: 1000}  
    length: 5  
  ▶ [[Prototype]]: Array(0)
```

Note how the function create bookings takes three parameters of flight number, numPassengers and price. The NumPassengers has a default value defined as 1. The Price has a default value of 199 x numPassengers. This works because NumPassengers is defined before price in the parameter list of the function.

```
▶ {flightNum: 'LH123', numPassengers: 1, price: 199}  
▶ {flightNum: 'LH123', numPassengers: 2, price: 800}  
▶ {flightNum: 'LH123', numPassengers: 2, price: 398}  
▶ {flightNum: 'LH123', numPassengers: 5, price: 995}  
▶ {flightNum: 'LH123', numPassengers: 1, price: 1000}
```

How passing arguments works: Value vs Reference

```
const flight = 'LH234';  
const jonas = {  
  name: 'Jonas Schmedtmann',  
  passport: 2349346893,  
};  
  
const checkIn = function (flightNum, passenger) {  
  flightNum = 'LH999';  
  passenger.name = 'Mr. ' +  
    passenger.name;  
  
  if (passenger.passport === 2349346893) {  
    alert('Checked in');  
  } else {  
    alert('Wrong passport!');  
  }  
};  
  
checkIn(flight, jonas);  
console.log(flight);  
// LH234  
// {name: 'Mr. Jonas Schmedtmann', passport: 2349346893}
```

3) Now we have a flight number defined and a passenger object.

1) This is a checkIn function that takes flightNum and passenger as parameters.

2) If the passport is correct then the passenger is checked in.

4) Note how the flight number is LH234 and not LH999 as defined in the function.

This is because of the way JavaScript treats primitives. The flight is copied from the variable and passed into the flightNum parameter.

5) When we pass in an object (jonas) to a function it is also affected by this change because when we pass an object we just send a reference to the location in the memory heap.

6) When we are manipulating the passenger object we are also manipulating the jonas object and whatever we do to passenger will also be done to the Jonas object because they are the same thing in the memory heap.

7) We need to be careful with this JavaScript behaviour because it can have unforeseen consequences in large codebases.

```
const newPassport = function (person) {  
  person.passport = Math.trunc(Math.random() * 1000000000000);  
};
```

```
newPassport(jonas);  
checkIn(flight, jonas);
```

9) Now because we are manipulating the Jonas object the passport number is different to the one expected.

8) Now we have a newPassport function which is manipulating the same object of Jonas to change the passport

127.0.0.1:5500 says

Wrong passport!

OK

There are two terms here: passing by value or pass by reference.

When we pass a primitive such as a string we are passing a value. When we pass an object we are using a reference to a memory location in the stack.

However, JavaScript does NOT have pass by reference even though we are passing in the memory location of the object in the stack. We are actually passing in a value which contains the reference to that memory location.

First class and High-Order functions

First Class Functions

JavaScript treats functions as **first-class citizens**

This means 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 a function FROM a function

- Call method on a function

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

High-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.

- Functions that receive another function

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

High order
function

Callback
function

- Functions that return a function

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

Higher-order
function

Returned
function

Functions Accepting Callback Functions

```
// This function converts a string to one word all lowercase
const oneWord = function (str) {
  return str.replace(/ /g, '').toLowerCase();
};
```

← **First Class function**

```
// This function converts the first word of a string to upper case
const upperFirstWord = function (str) {
  const [first, ...others] = str.split(' ');
  return [first.toUpperCase(), ...others].join(' ');
};
```

← **First Class function**

```
// Transformer function
const transformer = function (str, fn) {
  console.log(`Original string: ${str}`);
  console.log(`Transformed string: ${fn(str)}`);
  console.log(`Transformed by: ${fn.name}`);
};
```

← **High order function**

← **Callback function**

2) In the transformer function we are calling the function that was passed into it.

```
transformer('JavaScript is the best!', upperFirstWord);
transformer('JavaScript is the best!', oneWord);
```

1) Here we are calling the transformer function which accepts two parameters, a string and the function which we want to apply to the string. At this point the upperFirstWord function is not doing anything we are just simply passing the name of the function as a parameter.

Original string: JavaScript is the best!
Transformed string: JAVASCRIPT is the best!
Transformed by: upperFirstWord
Original string: JavaScript is the best!
Transformed string: javascriptisthebest!
Transformed by: oneWord

```
// An event listener that calls the high5 function
const high5 = function () {
  console.log('👋');
};
```

← **First Class
function**



The document.body is listening for a click and calls the high5 function when it detects that click.

```
document.body.addEventListener('click', high5);
```

↑
**High order built in
JavaScript function**

↑
**Callback
function**

```
['jonas', 'Marta', 'Adam'].forEach(high5);
```



The JavaScript built it function forEach is calling the high5 function for each item in the array.

- 1) Call back functions are used all the time in JavaScript, they help us follow the DRY principle of do not Repeat yourself! Because we can split the code up into reusable parts.
- 2) Callback functions allow us to create abstractions. Abstraction means we hide the detail about some code implementation, i.e convert to one word, because we do not care about that detail. We can now think about implementation at a higher level, i.e. transformer is just concerned with the console.log and delegating the string translation to other functions.

Functions Returning other Functions

```
const greet = function (greeting) {  
  return function (name) {  
    console.log(` ${greeting} ${name}`);  
  };  
};
```

```
const greeterHey = greet('hey');  
greeterHey('Jonas');  
greeterHey('Steven');
```

```
greet('hello')('Jonas');
```

```
const greetArr = greeting => name =>  
  console.log(` ${greeting} ${name}`);
```

```
greetArr('Hi')('Jonas');
```

```
Hi Jonas
```

A very simple greet function that takes a parameter of greeting such as hi or hello etc. It returns a function that receives an argument of name and console logs the results.

we can call the function from a variable and pass in a value of 'hey' to the greet function.

we can then call this variable and pass in a parameter which gets passed into the returned function. This is because of JavaScript closures. see later lessons.

We can do this all in one go by calling the function of greet and pass in a parameter of 'hello', Because this is a function we can pass in another parameter for the return function i.e. 'Jonas'.

This can be written as an arrow function. The greetArr function takes a parameter of greeting. this function returns a function so we have an arrow function within an arrow function. This is a shorter but more confusing way of writing a function that returns a function.

The call and Apply Methods

```
const lufthansa = {  
  airline: 'Lufthansa',  
  iataCode: 'LH',  
  bookings: [],
```

```
  // book: function() {}  
  book(flightNum, name) {  
    console.log(  
      `${name} booked a flight 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');  
console.log(lufthansa);
```

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

The object has a function called book which compiles a string and pushes it onto the bookings array.

Jonas Schmedtmann booked a flight on Lufthansa flight LH239
John Smith booked a flight on Lufthansa flight LH635



```
▼ {airline: 'Lufthansa', iataCode: 'LH', bookings: Array(2), book: f} ⓘ  
  airline: "Lufthansa"  
  ▶ book: f book(flightNum, name)  
  ▼ bookings: Array(2)  
    ▶ 0: {flight: 'LH239', name: 'Jonas Schmedtmann'}  
    ▶ 1: {flight: 'LH635', name: 'John Smith'}  
    length: 2  
    ▶ [[Prototype]]: Array(0)  
    iataCode: "LH"  
    ▶ [[Prototype]]: Object
```

Now we want the eurowings object to use the same function of book. We could copy it and put it in the eurowings object but that would be bad practice.

We will take the method of book and store it in an external variable. However it does not work.

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

```
book(23, 'Sarah Williams');
```

```
✖ ▶ Uncaught TypeError: Cannot read properties of undefined (reading 'airline')  
  at book (script.js:134:42)  
  at script.js:152:1
```

This is because the book function here is just a regular function call and it does not know what to do with the this keyword which now points to undefined.

```
book.call(eurowings, 23, 'Sarah  
Williams');
```

```
console.log(eurowings);
```

```
book.call(lufthansa, 239, 'Mary Cooper');  
console.log(lufthansa);
```

```
const swiss = {  
  airline: 'Swiss Air Lines',  
  iataCode: 'LX',  
  bookings: [],  
};
```

```
book.call(swiss, 583, 'Mary Cooper');
```

The solution is to use the call method on book function.

```
▶ {airline: 'Lufthansa', iataCode: 'LH', bookings: Array(2), book: f}  
Sarah Williams booked a seat on Eurowings flight EW23  
▶ {airline: 'Eurowings', iataCode: 'EW', bookings: Array(1)}  
Mary Cooper booked a seat on Lufthansa flight LH239  
▼ {airline: 'Lufthansa', iataCode: 'LH', bookings: Array(3), book: f} ⓘ  
  airline: "Lufthansa"  
  ▶ book: f book(flightNum, name)  
  ▼ bookings: Array(3)  
    ▶ 0: {flight: 'LH239', name: 'Jonas Schmedtmann'}  
    ▶ 1: {flight: 'LH635', name: 'John Smith'}  
    ▶ 2: {flight: 'LH239', name: 'Mary Cooper'}  
    length: 3  
    ▶ [[Prototype]]: Array(0)  
  iataCode: "LH"  
  ▶ [[Prototype]]: Object
```

We can add another airline that can call the book method withing the Lufthansa object. Note that the object properties must all be the same for this to work. i.e. airline, iataCode and bookings array.

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

The apply method does exactly the same as the call method but apply does not take a list of parameters but an array.

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

The call method does not get used much because we can now use the spread operator to break out the values from the array before using the call method.

George Cooper booked a seat on Swiss Air Lines flight LX583

```
▼ {airline: 'Swiss Air Lines', iataCode: 'LX', bookings: Array(2)} ⓘ  
  airline: "Swiss Air Lines"  
  ▼ bookings: Array(3)  
    ► 0: {flight: 'LX583', name: 'Mary Cooper'}  
    ► 1: {flight: 'LX583', name: 'George Cooper'}  
    ► 2: {flight: 'LX583', name: 'George Cooper'}  
    length: 3  
    ► [[Prototype]]: Array(0)  
  iataCode: "LX"  
  ► [[Prototype]]: Object
```

George Cooper booked a seat on Swiss Air Lines flight LX583

The bind Method

```
const bookEW = book.bind(eurowings);
const bookLH = book.bind(lufthansa);
const bookLX = book.bind(swiss);
```

```
bookEW(23, 'Steven Williams'); // Steven Williams booked a seat on Eurowings flight EW23
```

Note that unlike the call method we do not need to pass in the this keyword, i.e. lufthansa, eurowings, swiss etc. We just pass the flight number and passenger name.

```
const bookEW23 = book.bind(eurowings, 23);
bookEW23('Jonas Schmedtmann');
bookEW23('Martha Cooper');
```

But we can take this a step further and not just bind the airline but also bind the flight number so that only the passenger name argument is needed.

This is a common pattern called partial application where we are pre-defining some of the parameters we need to pass.

Bind allows us to manually set the this keyword for any function call. However bind does not immediately call the method, instead returning a new function where the this keyword is found so it is set to whatever value we pass into bind.

```
▼ {airline: 'Eurowings', iataCode: 'EW', bookings: Array(1)} ⓘ
  airline: "Eurowings"
  ▼ bookings: Array(4)
    ► 0: {flight: 'EW23', name: 'Sarah Williams'}
    ► 1: {flight: 'EW23', name: 'Steven Williams'}
    ► 2: {flight: 'EW23', name: 'Jonas Schmedtmann'}
    ► 3: {flight: 'EW23', name: 'Martha Cooper'}
    length: 4
    ► [[Prototype]]: Array(0)
    iataCode: "EW"
    ► [[Prototype]]: Object

Mary Cooper booked a seat on Lufthansa flight LH239
► {airline: 'Lufthansa', iataCode: 'LH', bookings: Array(3), book: f}

Mary Cooper booked a seat on Swiss Air Lines flight LX583
George Cooper booked a seat on Swiss Air Lines flight LX583
► {airline: 'Swiss Air Lines', iataCode: 'LX', bookings: Array(2)}

George Cooper booked a seat on Swiss Air Lines flight LX583
Steven Williams booked a seat on Eurowings flight EW23
Jonas Schmedtmann booked a seat on Eurowings flight EW23
Martha Cooper booked a seat on Eurowings flight EW23
```



```
lufthansa.planes = 300;
lufthansa.buyPlane = function () {
  this.planes++;
  console.log(this.plnaes);
};
```

Here we can add a new property to the Lufthansa object called planes and set the value to 300. We can also create a new function within the Lufthansa object to buy new planes. This function will increment the number of planes by 1 and console.log the planes value.

```
document.querySelector('.buy').addEventListener('click', lufthansa.buyPlane);
```

undefined

An event listener is added to the class of buy and on click it will call the Lufthansa function of buy planes. However it does not work and returns undefined.

This is because the this keyword is set dynamically so in the case of an event listener it is set to the button which is the object calling the function.

```
lufthansa.planes = 300;
lufthansa.buyPlane = function () {
  console.log(this);

  this.planes++;
  console.log(this.planes);
};
```

But we need the this keyword to point to the function so we will have to manually set it.

```
document.querySelector('.buy').addEventListener('click',
lufthansa.buyPlane.bind(lufthansa));
```

To solve this problem we bind the this keyword to the Lufthansa object.



```
const addTax = (rate, value) => value + value * rate;  
console.log(addTax(0.1, 200));
```

We have a basic arrow function that adds a tax to some value

220

```
const addVAT = addTax.bind(null, 0.23);
```

we can use the bind function to preset the value of tax. We always bind to the this keyword but in this function the this keyword is not even present so we set this to null. the second parameter of 0.23 is binding the tax rate to 23%. The order of the arguments is important because the tax is the first argument in the addTax function.

```
console.log(addVAT(100));  
console.log(addVAT(23));
```

123

28.29

```
const addTaxRate = function (rate) {  
  return function (value) {  
    return value + value * rate;  
  };  
};
```

The above can be rewritten as a function that returns a function.

```
const addVAT2 = addTaxRate(0.23);  
console.log(addVAT2(100));  
console.log(addVAT2(23));
```

Immediately Invoked Function Expressions (IIFE)

```
const runOnce = function () {  
  console.log('This will never run again!');  
};  
  
runOnce();
```

an IIFE is a function that is immediately executed only once.

We could write this as a normal function but this function can be called multiple times.

```
function () {  
  console.log('This will never run again!');  
}
```

We can do this by writing the expression without assigning it to any variable. However JavaScript will throw an error.

✖ Uncaught SyntaxError: Function statements require a function name (at [script.js:323:1](#)) [script.js:323](#)

```
(function () {  
  console.log('This function will run only once!');  
})();  
    This function will run only once!
```

But we can trick JavaScript by wrapping the function in parenthesis **()** then calling it at the end with **()**;

```
((() => console.log('This will ALSO never run again!'))());
```

This can be written as an arrow function again wrapping the function in parenthesis **()** then calling it at the end with **()**;

```
(function () {  
  console.log('This function has private variables!');  
  const isPrivate = 23;  
})();
```

But why was IIFE invented?
Anything inside a function is private with local scope to that function.

```
console.log(isPrivate);
```

✖ ▶ Uncaught ReferenceError: isPrivate is not defined
at script.js:334:13

```
{  
  const isPrivate = 23;  
  var notPrivate = 46;  
}
```

Scope also depends if a variable is defined as var or let/const. See eariler lessons about scope.

```
console.log(notPrivate);  
console.log(isPrivate);
```

46

✖ ▶ Uncaught ReferenceError: isPrivate is not defined
at script.js:340:13

Closures

```
const secureBooking = function () {  
  let passengerCount = 0;  
  
  return function () {  
    passengerCount++;  
    console.log(` ${passengerCount} passengers` );  
  };  
};
```

```
const booker = secureBooking();
```

```
booker();  
booker();  
booker();
```

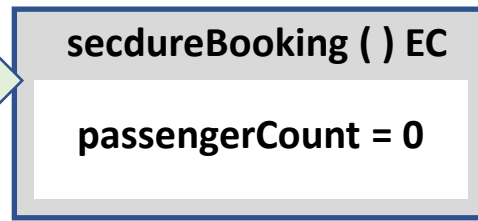
1 passengers
2 passengers
3 passengers

The function `secureBooking` creates the closure. Closure is something that we do not explicitly use. It is called `secureBooking` because the `passengerCount` variable is something that cannot be manipulated from the outside.

The `secureBooking` function returns a function that increments the `passengerCount` by 1.

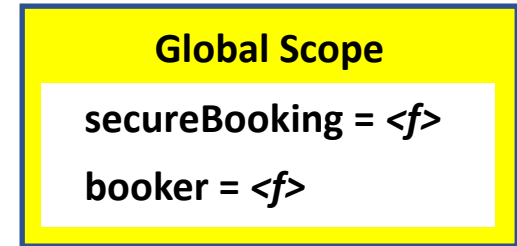
Now we have a `booker` variable that calls `secureBooking` function which returns a new function that is stored in the `booker` variable.

2) When secureBooking is run it is added to the call stack with all of its variables

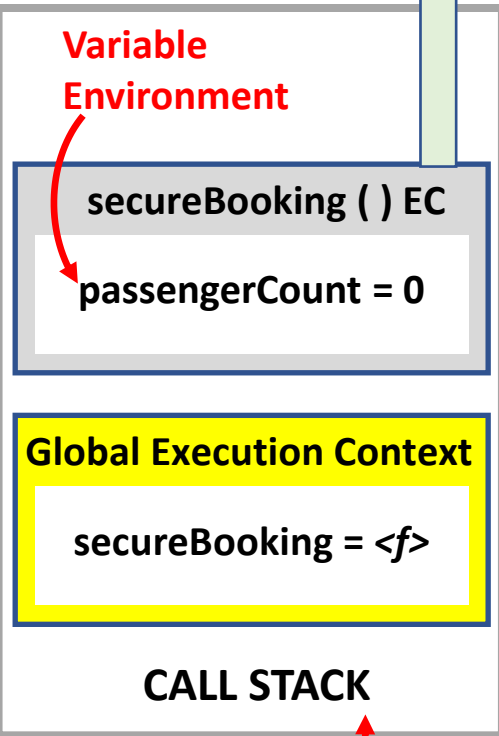
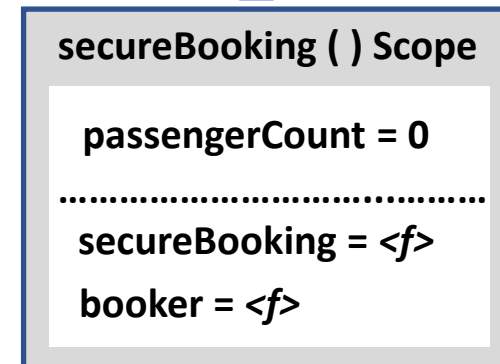


4) The SecureBooking execution context disappears of the stack because it's job is done.

3) Now the global scope has the booker variable which contains a function.



```
const secureBooking = function () {  
  let passengerCount = 0;  
  
  return function () {  
    passengerCount++;  
    console.log(`${passengerCount} passengers`);  
  };  
};  
  
const booker = secureBooking();  
  
booker();  
booker();  
booker();
```



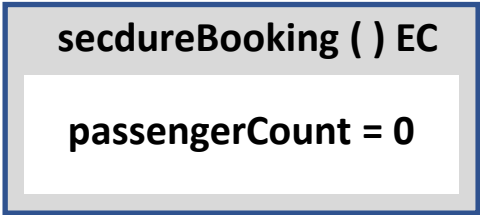
1) The code is running in the global execution context and currently only has the secureBooking Function.

Order in which functions are written in the code (from top to bottom)

Order in which functions were called (from bottom to top)

SCOPE CHAIN

6) When booker is run it is added to the call stack with no variables



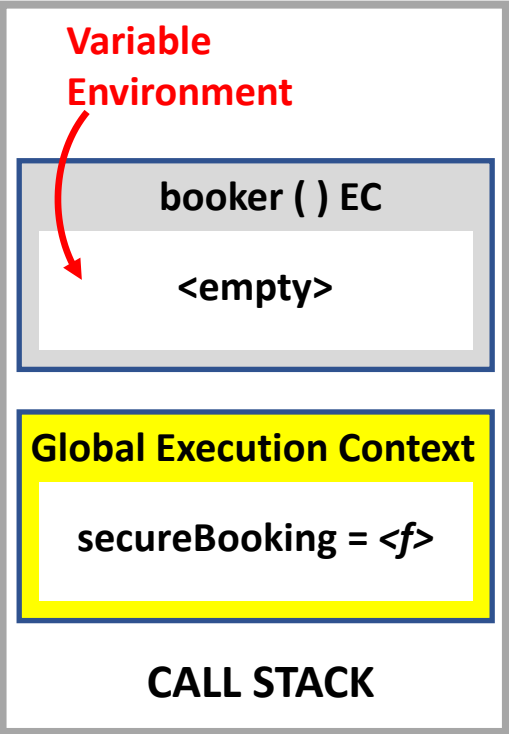
8) How will the booker function access the passenger count variable that is not in the call stack?

```
const secureBooking = function () {
  let passengerCount = 0;

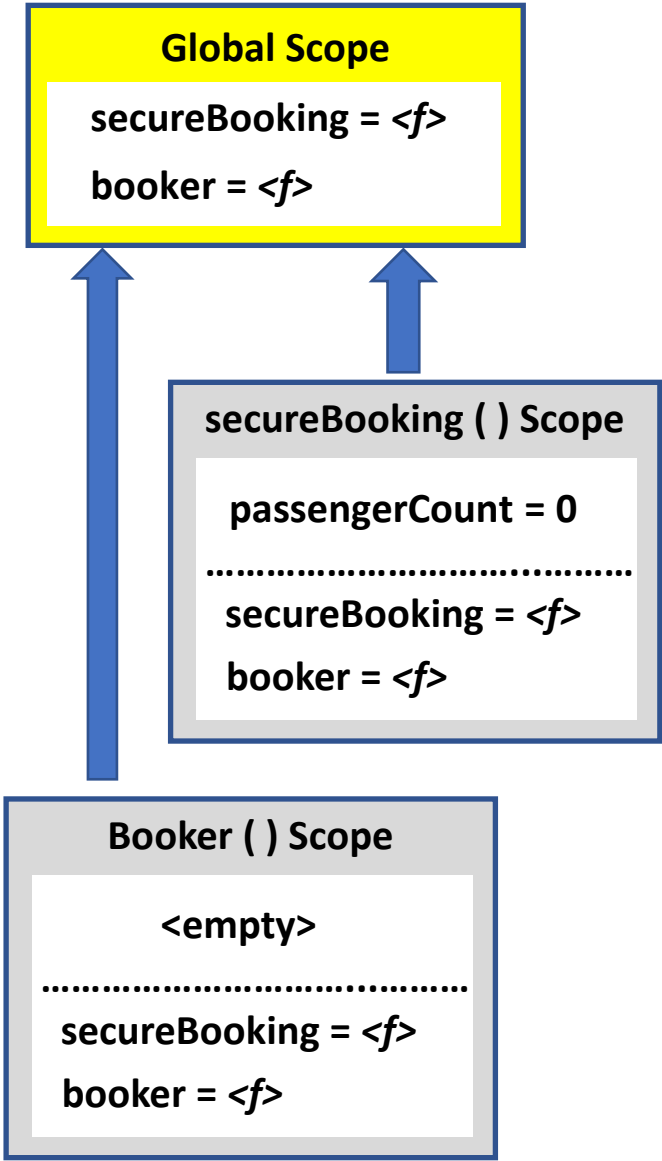
  return function () {
    passengerCount++;
    console.log(`${passengerCount} passengers`);
  };
};

const booker = secureBooking();

booker();
booker();
booker();
```

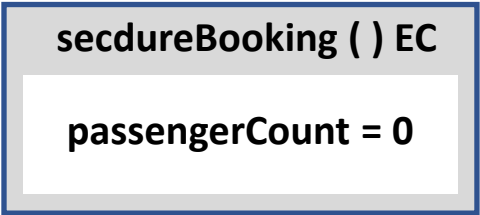


7) Booker is simply a child scope of the global scope

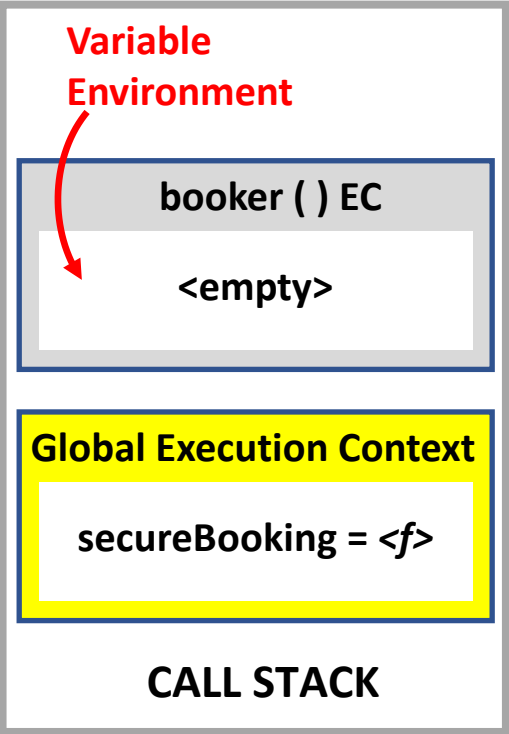


5) Now we can call the booker function three times but how does it execute secureBooking when it has gone from the call stack?

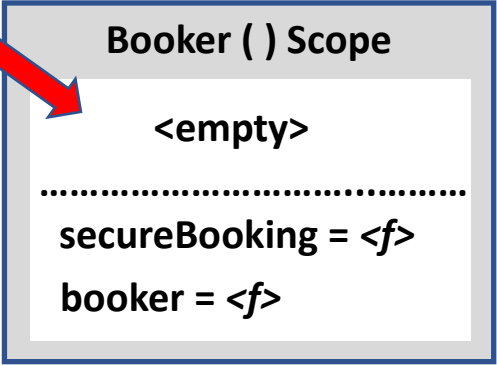
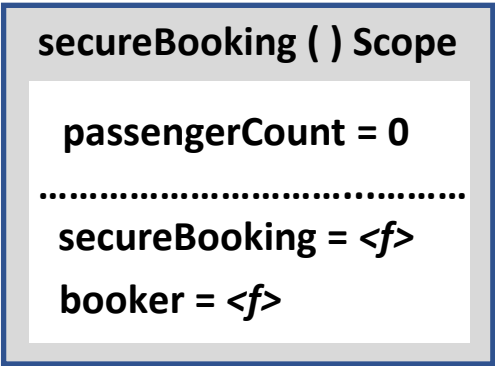
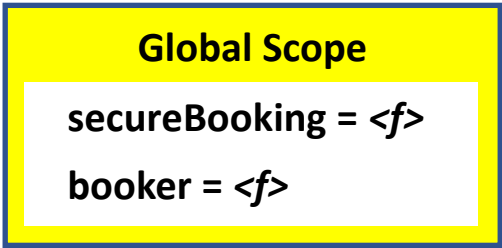
10) booker was created in the EC of secureBooking which was popped of the call stack previously.



```
const secureBooking = function () {  
  let passengerCount = 0;  
  
  return function () {  
    passengerCount++;  
    console.log(`${passengerCount} passengers`);  
  };  
};  
  
const booker = secureBooking();  
booker();  
booker();  
booker();
```



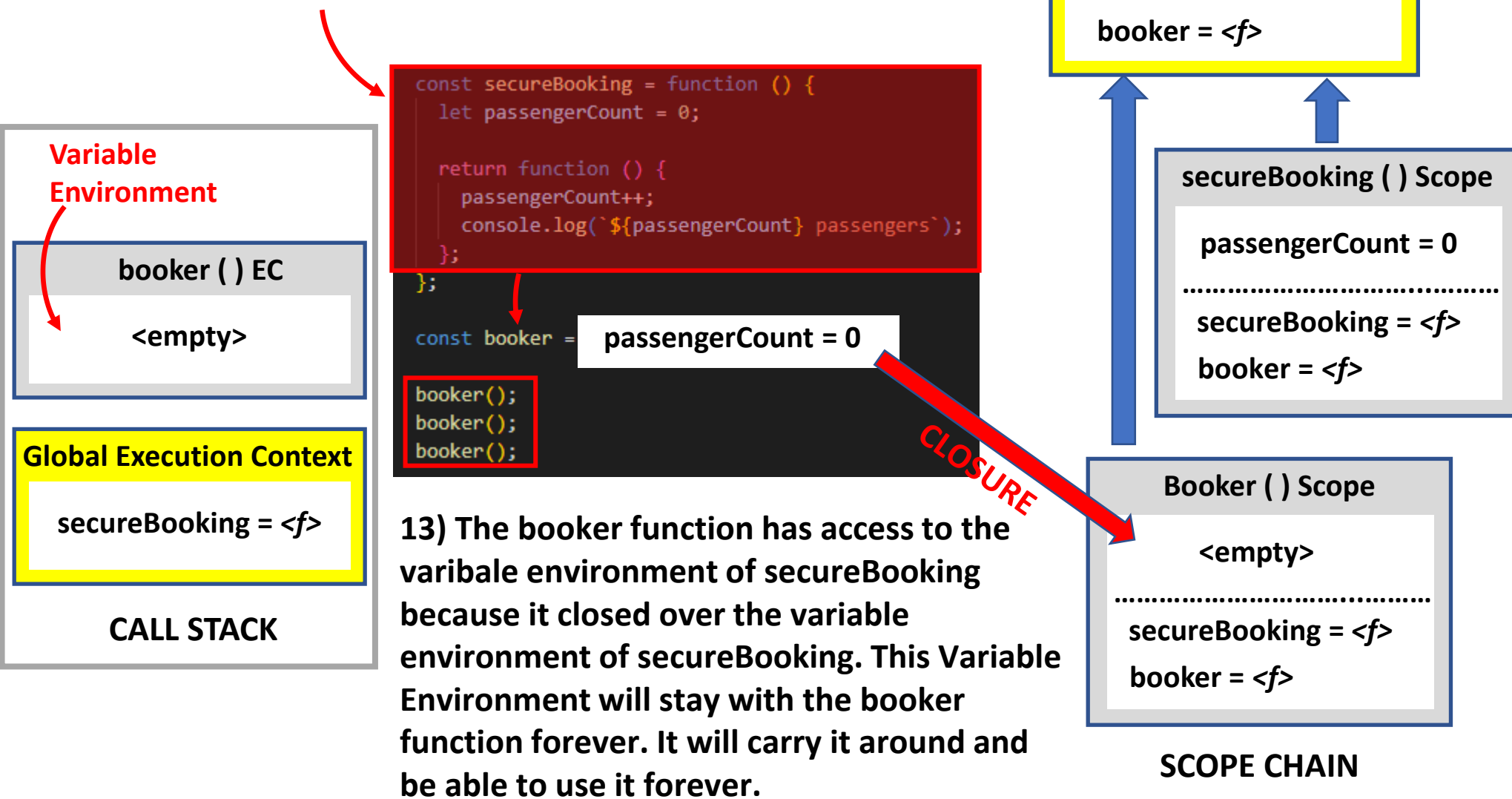
11) The closure is the variable environment attached to the function exactly as it was at the time and place the function was created. The scope chain is preserved through the closure.



9) Closure: Any function always has access the variable environment to which the function was created.

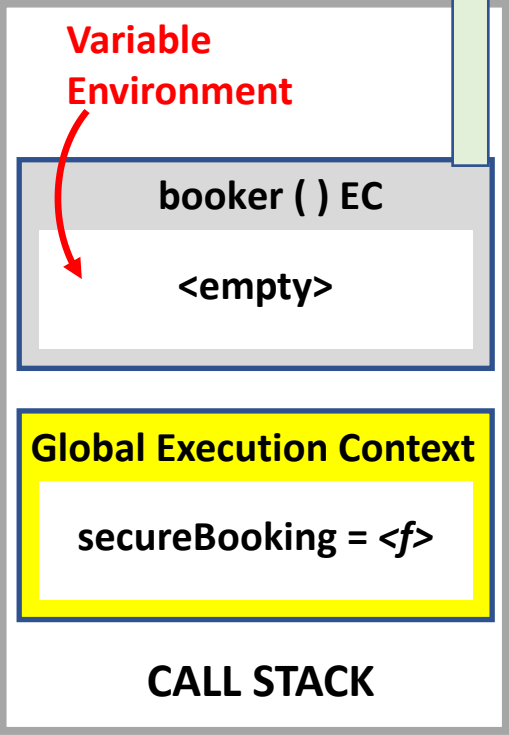
SCOPE CHAIN

12) We can say that the booker function closed over its parent scope.



14) The closure is the variable environment attached to the execution context at the time of execution. The scope chain is preserved through the closure. The variable environment keeps living in the JS engine even though the execution context has been destroyed.

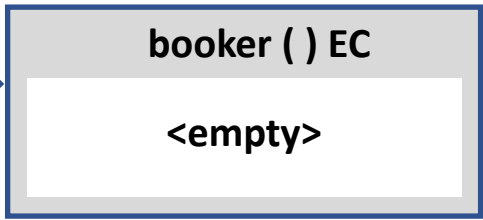
16) passengerCount is incremented by 1 and the booker drops off the execution context



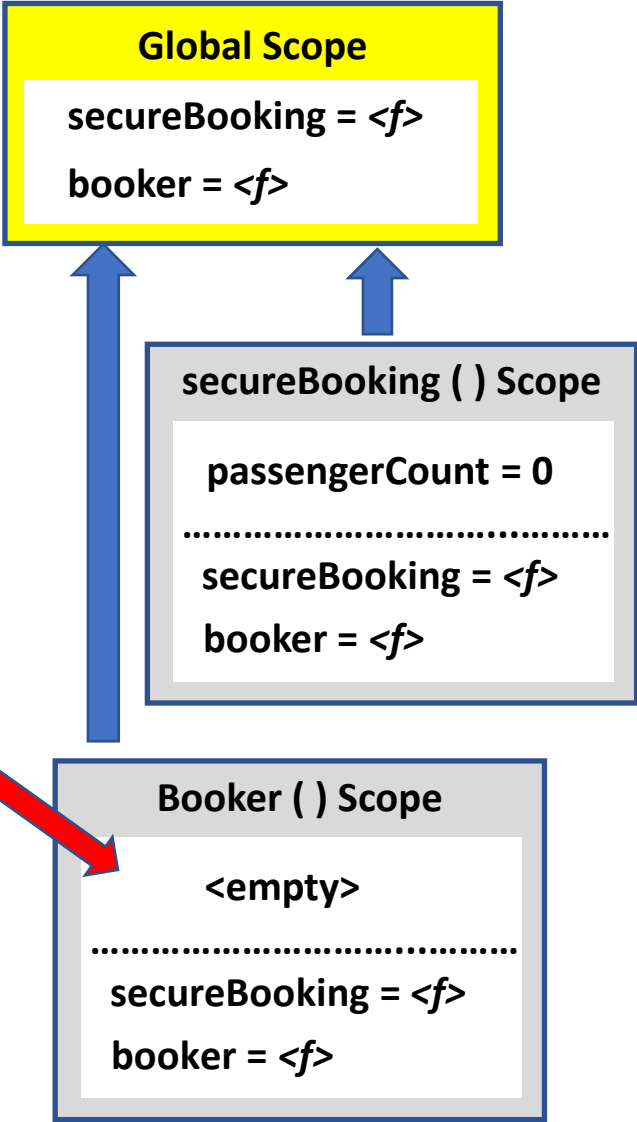
```
const secureBooking = function () {  
  let passengerCount = 0;  
  
  return function () {  
    passengerCount++;  
    console.log(`${passengerCount} passengers`);  
  };  
};  
  
const booker =
```

passengerCount = 1

```
booker();  
booker();  
booker();
```



17) this cycle repeat for each time the booker function is called.



Priority over scope chain

CLOSURE

15) The booker function attempts to increase the passengerCount variable but it is not in the current scope. It looks in the closure to find the variable. In fact the closure has priority over the scope chain, it looks at the closure first.

A closure is the closed-over **variable environment** of the execution context **in which a function was created**, *even after* that execution context is gone.

A closure gives a function access to all the variables of **its parent function**, *even after* that parent function has returned. The function keeps a **reference** to its outer scope, which **preserves** the scope chain throughout time.

A closure makes sure that a function does not lose connection to **variables that existed at the function's birthplace**.

A closure is like a **backpack** that a function carries around wherever it goes. This backpack contains all the **variables that were present in the environment where the function was created**.

NOTE: We do **not** have to create closures manually. This is a JavaScript feature that happens automatically. WE cannot even access closed over variables explicitly. a closure is **not** a tangible JavaScript object.

```
console.dir(booker);
```

script.js:36

```
▼ f anonymous() ⓘ  
  length: 0  
  name: ""  
  ► prototype: {constructor: f}  
  arguments: (...)  
  caller: (...)  
  [[FunctionLocation]]: script.js:353  
  ► [[Prototype]]: f ()  
  ▼ [[Scopes]]: Scopes[3]  
    ► 0: Closure (secureBooking) {passengerCount: 3}  
    ► 1: Script {secureBooking: f, booker: f}  
    ► 2: Global {window: Window, self: Window, document: document, name: '', location: Location, ...}
```

Console.dir will show us the closure of a function. Here we see that under scopes the passengerCount from secureBooking is the closure. Note that what appears here in double square brackets is not accessible but purely for diagnostic purposes.

More Closures Examples

```
let f;
```

46

```
const g = function () {  
  const a = 23;  
  f = function () {  
    console.log(a * 2);  
  };  
};
```

1554

```
const h = function () {  
  const b = 777;  
  f = function () {  
    console.log(b * 2);  
  };  
};
```

```
g();  
f();  
console.dir(f);  
// re-assigning f function  
h();  
f();  
console.dir(f);
```

```
▼ f f() ⓘ  
  length: 0  
  name: "f"  
  ▶ prototype: {constructor: f}  
  arguments: (...)  
  caller: (...)  
  [[FunctionLocation]]: script.js:375  
  ▶ [[Prototype]]: f ()  
  ▼ [[Scopes]]: Scopes[3]  
    ▶ 0: Closure (g) {a: 23}  
    ▶ 1: Script {f: f, g: f, h: f}  
    ▶ 2: Global {window: Window, self: Window, document: document, name: '', location: Location, ...}
```

f function closes over the execution context of the g function to pick up its variable environment to access the a variable a.

```
▼ f f() ⓘ  
  length: 0  
  name: "f"  
  ▶ prototype: {constructor: f}  
  arguments: (...)  
  caller: (...)  
  [[FunctionLocation]]: script.js:382  
  ▶ [[Prototype]]: f ()  
  ▼ [[Scopes]]: Scopes[3]  
    ▶ 0: Closure (h) {b: 777}  
    ▶ 1: Script {f: f, g: f, h: f}  
    ▶ 2: Global {window: Window, self: Window, document: document, name: '', location: Location, ...}
```

f function now closes over the execution context of the h function to pick up its variable environment to access the a variable b.

```
const boardPassengers = function (n, wait) {  
  const perGroup = n / 3;
```

A timer is another great example where we do not need to return a function to see an example of closure.

```
  setTimeout(function () {  
    console.log(`We are now boarding all ${n} passengers`);  
    console.log(`There are 3 groups, each with ${perGroup} passengers`);  
  }, wait * 1000);
```

```
  console.log(`Will start boarding in ${wait} seconds`);  
};
```

The boardPassenger function takes n number of passengers who have to wait before boarding.

```
const perGroup = 1000;  
boardPassengers(180, 3);
```

The timeout function is accessing n, wait and perGroup as a closure.

Note how the closure is taking priority over the global scope. The const perGroup is defined within the boardPassengers and is calculated from n which is passed in (deriving a value of 60 passengers.) The Global scope for perGroup is defined outside of the function and set to 1000.

```
Will start boarding in 3 seconds
```

```
We are now boarding all 180 passengers
```

```
There are 3 groups, each with 60 passengers
```

Working with Arrays

Simple Array Methods

```
let arr = ['a', 'b', 'c', 'd', 'e'];
```

```
console.log(arr.slice(2)); ▶ (3) ['c', 'd', 'e']
```

Slice: We can take a slice of an array without mutating the original

```
console.log(arr.slice(2, 4)); ▶ (2) ['c', 'd']
```

We can specify the end position of where we want the slice to stop. Position 4 corresponds to e because an array counts from zero. The end position is not included in the slice.

```
console.log(arr.slice(-2)); ▶ (2) ['d', 'e']
```

A negative number will start splicing from the end of the array, i.e. 2 positions back.

```
console.log(arr.slice(-1)); ▶ ['e']
```

-1 will always get us the last item in the array.

```
console.log(arr.slice(1, -2)); ▶ (2) ['b', 'c']
```

We can also slice from the middle of the array choosing starting position and ending position as a negative number.

```
console.log(arr.slice()); ▶ (5) ['a', 'b', 'c', 'd', 'e']
```

We can use slice to create a shallow copy of the array by not specifying a starting or ending position.


```
console.log(arr.splice(2));  
console.log(arr);
```

```
▶ (3) ['c', 'd', 'e']  
▶ (2) ['a', 'b']
```

Splice will mutate the original array and is destructive. We can see after the splice that everything after position 2 has been removed.

```
console.log(arr.splice(-1));  
console.log(arr);
```

```
▶ ['e']  
▶ (4) ['a', 'b', 'c', 'd']
```

-1 will remove the last element from the array.

```
console.log(arr.splice(1, 2));  
console.log(arr);
```

```
▶ (2) ['b', 'c']  
▶ (2) ['a', 'd']
```

We can specify a starting and ending position of elements to remove from the array.

```
const arr2 = ['j', 'i', 'h', 'g', 'f'];  
console.log(arr2.reverse());  
console.log(arr2);
```

```
▶ (5) ['f', 'g', 'h', 'i', 'j']  
▶ (5) ['f', 'g', 'h', 'i', 'j']
```

Reverse will reorder the array in reverse

Note that reverse does mutate the original array.

```
let arr1 = ['a', 'b', 'c', 'd', 'e'];  
let arr3 = ['j', 'i', 'h', 'g', 'f'];  
const letters = arr1.concat(arr3);  
console.log(letters);
```

```
▶ (10) ['a', 'b', 'c', 'd', 'e', 'j', 'i', 'h', 'g', 'f']
```

```
console.log([...arr1, ...arr3]);
```

```
▶ (10) ['a', 'b', 'c', 'd', 'e', 'j', 'i', 'h', 'g', 'f']
```

Concat method will join two arrays where we specify the array that we want to concatenate and the array we want to pass in.

Equally, the spread operator can be used to achieve the same result as concat.

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

```
a - b - c - d - e - j - i - h - g - f
```

```
a---b---c---d---e---j---i---h---g---f
```

The join method will join all the elements in an array into a string and we can specify what we want between each element.

The new at method (from ES22)

```
const arr4 = [23, 11, 64];  
console.log(arr4[0]);  
console.log(arr4.at(0));
```

```
23
```

```
23
```

```
console.log(arr4[arr4.length - 1]);
```

```
console.log(arr4.slice(-1)[0]);
```

```
64
```

```
console.log(arr4.at(-1));
```

```
64
```

```
console.log(arr4.at(-2));
```

```
64
```

```
console.log('jonas'.at(0));
```

```
j
```

```
console.log('jonas'.at(-1));
```

```
s
```

The at method is the same as using a square bracket to denote the position in the array.

We can calculate the last element of the array when we do not know how long the array is using array length or slicing at -1 and reading position zero of this new array.

At is useful now because we just take the element at position -1. at(-2) will get us the second from last element.

The at method also works for strings. We can use it to get the first or last letter of a string.

Looping arrays: forEach method

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

for (const movement of movements) {
  if (movement > 0) {
    console.log(`you deposited ${movement}`);
  } else {
    console.log(`You withdrew ${Math.abs(movement)}`);
  }
}
```

```
console.log('----- FOREACH -----');
movements.forEach(function (movement) {
  if (movement > 0) {
    console.log(`you deposited ${movement}`);
  } else {
    console.log(`You withdrew ${Math.abs(movement)}`);
  }
});
```

```
you deposited 200
you deposited 450
You withdrew 400
you deposited 3000
You withdrew 650
You withdrew 130
you deposited 70
you deposited 1300
```

the `forEach` method is a high order function that has a callback function as an anonymous function.

each iteration through the array the anonymous function is called with a the value corresponding to the position in the array.

```
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 (movement, index, array) {
  if (movement > 0) {
    console.log(`Movement ${index + 1}: you deposited ${movement}`);
  } else {
    console.log(`Movement ${index + 1}: you withdrew ${Math.abs(movement)}`);
  }
});
```

The forEach method passes the element, index and array so we can use that to construct our strings.

Note that in the real world we would abbreviate index to i and array to arr.

```
Movement 1: you deposited 200
Movement 2: you deposited 450
Movement 3: you withdrew 400
Movement 4: you deposited 3000
Movement 5: you withdrew 650
Movement 6: you withdrew 130
Movement 7: you deposited 70
Movement 8: you deposited 1300
```

```
----- FOREACH -----
Movement 1: you deposited 200
Movement 2: you deposited 450
Movement 3: you withdrew 400
Movement 4: you deposited 3000
Movement 5: you withdrew 650
Movement 6: you withdrew 130
Movement 7: you deposited 70
Movement 8: you deposited 1300
```

forEach method with Sets and maps

```
const currencies = new Map([
  ['USD', 'United States dollar'],
  ['EUR', 'Euro'],
  ['GBP', 'Pound sterling'],
]);
```

The forEach method on a map passes the value, key and map, so we can use that to construct our strings.

```
currencies.forEach(function (value, key, map) {
  console.log(`${key}: ${value}`);
});
```

USD: United States dollar

EUR: Euro

GBP: Pound sterling

```
const currenciesUnique = new Set(['USD', 'GBP', 'EUR', 'USD', 'EUR', 'EUR']);
console.log(currenciesUnique);
```

▶ Set(3) {'USD', 'GBP', 'EUR'}

```
currenciesUnique.forEach(function (value, key, map) {
  console.log(`${key}: ${value}`);
});
```

USD: USD

GBP: GBP

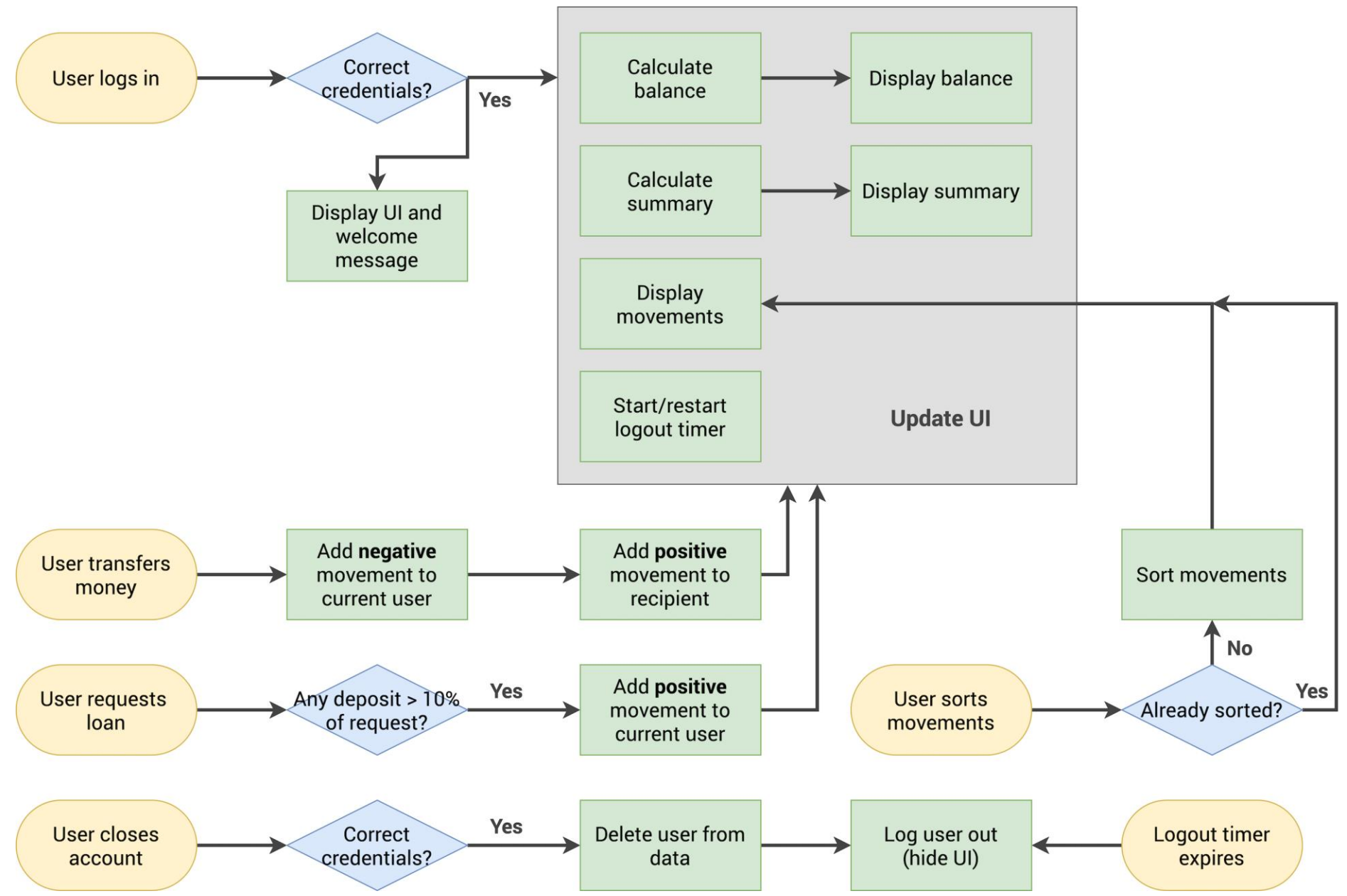
EUR: EUR

The first console log shows that the set has taken out duplicate values because a set is an array that does not have duplicate values. Notice in the forEach loop we are using the same three parameters of value, key and map even though sets do not have keys. This is to allow convention in the forEach.

```
currenciesUnique.forEach(function (value, _, map) {
  console.log(`${value}: ${value}`);
});
```

We can remove the key and replace it with underscore. Underscore is a throwaway variable which is a variable that is unnecessary.

Because there is no key we only have value in the string.



User objects for people
that have a bankist account

```
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 account3 = {  
  owner: 'Steven Thomas Williams',  
  movements: [200, -200, 340, -300, -20, 50, 400, -460],  
  interestRate: 0.7,  
  pin: 3333,  
};
```

```
const account4 = {  
  owner: 'Sarah Smith',  
  movements: [430, 1000, 700, 50, 90],  
  interestRate: 1,  
  pin: 4444,  
};
```

```
const accounts = [account1, account2, account3, account4];
```

```
const labelWelcome = document.querySelector('.welcome');
const labelDate = document.querySelector('.date');
const labelBalance = document.querySelector('.balance__value');
const labelSumIn = document.querySelector('.summary__value--in');
const labelSumOut = document.querySelector('.summary__value--out');
const labelSumInterest = document.querySelector('.summary__value--interest');
const labelTimer = document.querySelector('.timer');

const containerApp = document.querySelector('.app');
const containerMovements = document.querySelector('.movements');

const btnLogin = document.querySelector('.login__btn');
const btnTransfer = document.querySelector('.form__btn--transfer');
const btnLoan = document.querySelector('.form__btn--loan');
const btnClose = document.querySelector('.form__btn--close');
const btnSort = document.querySelector('.btn--sort');

const inputLoginUsername = document.querySelector('.login__input--user');
const inputLoginPin = document.querySelector('.login__input--pin');
const inputTransferTo = document.querySelector('.form__input--to');
const inputTransferAmount = document.querySelector('.form__input--amount');
const inputLoanAmount = document.querySelector('.form__input--loan-amount');
const inputCloseUsername = document.querySelector('.form__input--user');
const inputClosePin = document.querySelector('.form__input--pin');
```


Creating DOM elements

<https://developer.mozilla.org/en-US/docs/Web/API/Element/insertAdjacentHTML>

```
const displayMovements = function (movements) {  
  containerMovements.innerHTML = '';   
  movements.forEach(function (mov, i) {  
    const type = mov > 0 ? 'deposit' : 'withdrawal';
```

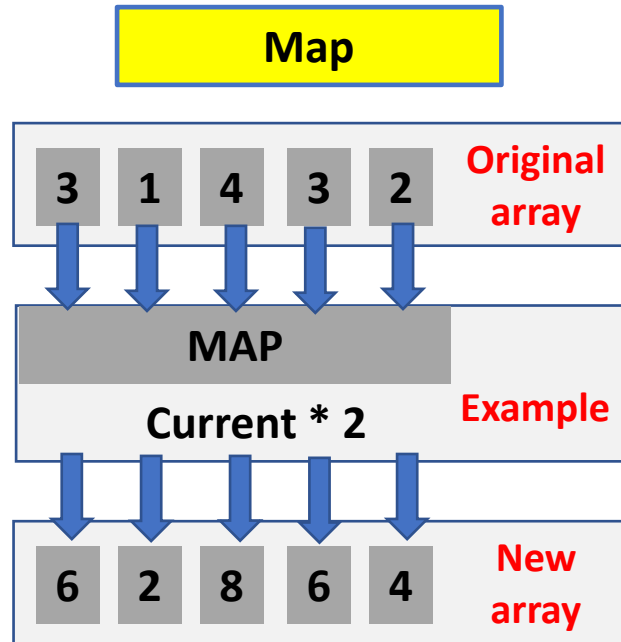
Using innerHTML means that any JavaScript references to the descendants of element will be removed.

Using Template literals to build a block of HTML for each row of movements

```
    <div class="movements__row">  
      <div class="movements__type movements__type--${type}">${i + 1} ${type}</div>  
      <div class="movements__date">3 days ago</div>  
      <div class="movements__value">${mov}</div>  
    </div>  
  `;  
  
  containerMovements.insertAdjacentHTML('afterBegin', html);  
});  
};  
  
displayMovements(account1.movements);
```

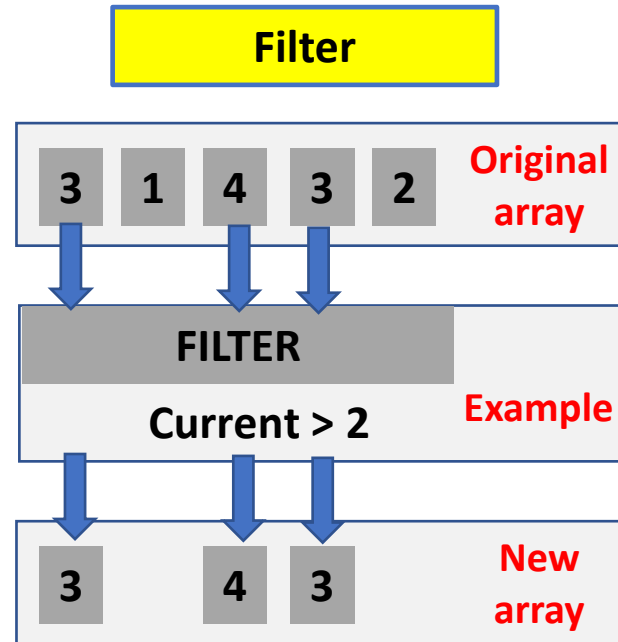
When you use insertAdjacentHTML, adding additional content will not corrupt the existing JS references and the existing nodes are not altered. We are using it here to forEach loop over the array and add a html block 'movements__row' for each iteration.

Data Transformations: Map, Filter, Reduce



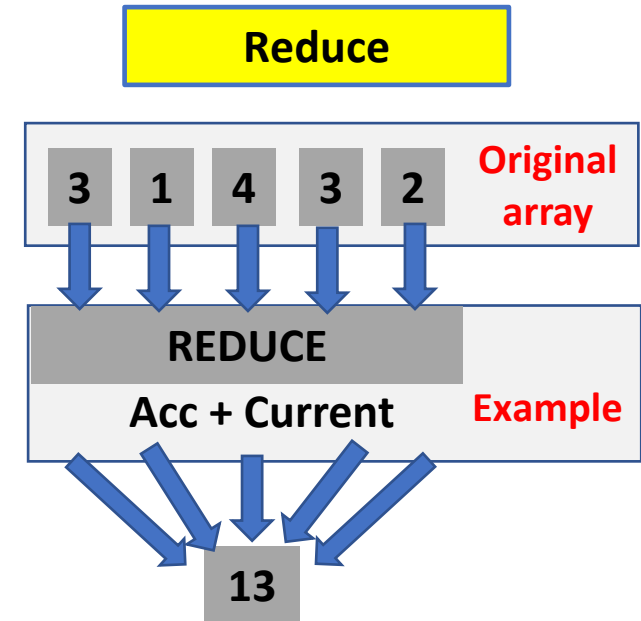
map is similar to the `forEach` method but creates a brand new array. IT maps the original values of an array to a new array.

Map returns a **new array** containing the results of applying an operation on all original array elements.



Filter is used to filter for certain elements in the array.

filter returns a new array containing the array elements that passed a specific test condition.



We have an accumulator variable and as we loop through the array we add on the values.

reduce boils ("reduces") all array elements down to one single value (e.g adding all elements together)

Reduce returns a single value, not an array.

```
const eurToUsd = 1.1;
```

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

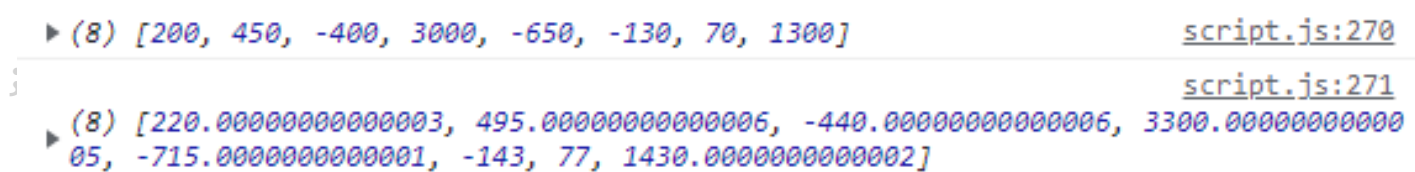
```
const movementsUSD = movements.map(function (mov) {  
  return mov * eurToUsd;  
});
```

The MAP method

We use an anonymous call back function to return a new array of values of movements * eurToUsd.

Note how the original movements array was not mutated but conserved.

```
console.log(movements);  
console.log(movementsUSD);
```

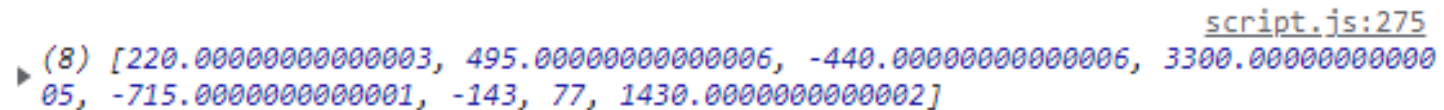


```
▶ (8) [200, 450, -400, 3000, -650, -130, 70, 1300] script.js:270  
▶ (8) [220.00000000000003, 495.00000000000006, -440.00000000000006, 3300.00000000000005, -715.0000000000001, -143, 77, 1430.0000000000002] script.js:271
```

```
const movementsUSDfor = [];  
for (const mov of movements) movementsUSDfor.push(mov * eurToUsd);
```

```
console.log(movementsUSDfor);
```

The same could be achieved by a for loop but it has more steps: 1) initialise blank array, 2) loop over array and perform calculation, 3) push new value into array.



```
▶ (8) [220.00000000000003, 495.00000000000006, -440.00000000000006, 3300.00000000000005, -715.0000000000001, -143, 77, 1430.0000000000002] script.js:275
```

```
const movementsUSDarr = movements.map(mov => mov * eurToUsd);  
console.log(movementsUSDarr);
```

This can be further reduced to an arrow function.

```
const createUsernames = function (accs) {
  accs.forEach(function (acc) {
    acc.username = acc.owner
      .toLowerCase()
      .split(' ')
      .map(name => name[0])
      .join('');
  });
};

createUsernames(accounts);
console.log(accounts);
```

▼ (4) [{...}, {...}, {...}, {...}] ⓘ

- ▶ 0: {owner: 'Jonas Schmedtmann', movements: Array(8), interestRate: 1.2, pin: 1111}
- ▶ 1: {owner: 'Jessica Davis', movements: Array(8), interestRate: 1.5, pin: 2222, us
- ▼ 2:
 - interestRate: 0.7
 - ▶ movements: (8) [200, -200, 340, -300, -20, 50, 400, -460]
 - owner: "Steven Thomas Williams"
 - pin: 3333
 - username: "stw"
 - ▶ [[Prototype]]: Object
- ▶ 3: {owner: 'Sarah Smith', movements: Array(5), interestRate: 1, pin: 4444, usern
- length: 4
- ▶ [[Prototype]]: Array(0)

We use the map method within a forEach loop to create a new array of usernames by taking the initial letter of owner from the accounts objects.

The FILTER method

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

```
const deposits = movements.filter(function (mov) {  
  return mov > 0;  
});
```

We use the filter method to select only the values from movements that are positive i.e. deposits.

```
console.log(movements); ▶ (8) [200, 450, -400, 3000, -650, -130, 70, 1300]  
console.log(deposits);   ▶ (5) [200, 450, 3000, 70, 1300]
```

```
const depositsFor = [];  
for (const mov of movements) if (mov > 0) depositsFor.push(mov);
```

```
console.log(depositsFor); ▶ (5) [200, 450, 3000, 70, 1300]
```

The same could be achieved by a for loop but it has more steps:

- 1) initialise blank array,
- 2) loop over array and perform calculation,
- 3) push new value into array.

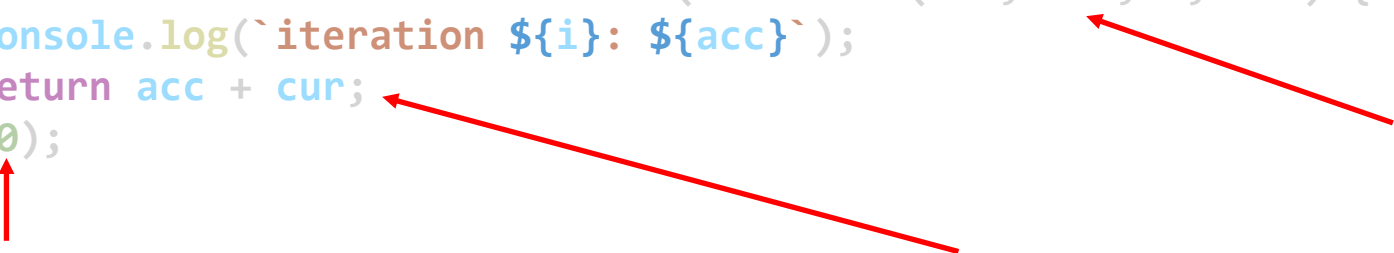
```
const withdrawals = movements.filter(mov => mov < 0);
```

```
console.log(movements);  
console.log(withdrawals);  
▶ (8) [200, 450, -400, 3000, -650, -130, 70, 1300]  
▶ (3) [-400, -650, -130]
```

Same can be done to get a withdrawals array, This time using an arrow function.

The REDUCE method

```
const balance = movements.reduce(function (acc, cur, i, arr) {  
  console.log(`iteration ${i}: ${acc}`);  
  return acc + cur;  
}, 0);
```



1) The reduce method takes the usual 3 parameters of current, iteration and array but also has the additional value of the accumulator (acc)

3) The accumulator value we set here. In this case we want to start from a value of zero.

```
console.log(balance);
```

2) For each iteration through the loop we want to increment the accumulator by the current value (cur) and return that new value so that the acc is updated for the next iteration in the loop.

iteration 0:	0
iteration 1:	200
iteration 2:	650
iteration 3:	250
iteration 4:	3250
iteration 5:	2600
iteration 6:	2470
iteration 7:	2540
	3840

```
let balance2 = 0;  
for (const mov of movements) balance2 += mov;  
console.log(balance2);
```

This could also be achieved with a for loop but it is a bit more cumbersome because we have to first define a balance variable outside of the loop and increment it with each iteration.

```
const balance3 = movements.reduce((acc, cur) => acc + cur, 0);  
console.log(balance3);
```

Same can be done to get balance, this time using an arrow function.

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

```
const max = movements.reduce((acc, mov) => {  
  if (acc > mov) return acc;  
  else return mov;  
}, movements[0]);
```

```
console.log(max);
```

The reduce method can also be used to find the maximum value in an array.

The accumulator starts at the first value in the array. As we iterate over it we are performing the logic of updating the accumulator if the current value is greater than the accumulator. In this example it finds the maximum value of 3000.

```
const min = movements.reduce((acc, mov) => {  
  if (acc < mov) return acc;  
  else return mov;  
}, movements[0]);
```

```
console.log(min);
```

The reduce method can also be used to find the minimum value in an array.

The Magic of Chaining Methods

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

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

We can change any methods as long as they return an array. Note how reduce returns a single value so it must be last in the chain.

```
const totalDepositsUSD = movements
  .filter(mov => mov < 0)
  .map((mov, i, arr) => {
    console.log(arr);
    return mov * eurToUsd;
  })
  .reduce((acc, mov) => acc + mov, 0);
console.log(totalDepositsUSD);
```

Now the error is seen, the filter is selecting values less than zero not greater than zero!

We are taking an array of movements and selecting the deposits (positive values) then converting them to USD.

This could be done as a series of individual functions but that would be very lengthy verbose code.

We can chain methods. First we use filter method to pick from the movements array all values greater than zero. Next in the chain we use map method to transform the values into USD. Finally we use reduce to calculate the total. All in one function.

Chaining methods presents diagnostic challenges if we get an unexpected result. In this scenario there is a code error which returns a negative value when deposits should be positive values.

To diagnose it we can break it in half and console log the results of the filter operation which is the array being used for the map operation.

```
► (3) [-400, -650, -130]
► (3) [-400, -650, -130]
► (3) [-400, -650, -130]
-1298.0000000000002
```



```

const calcDisplaySummary = function (movements) {
  const incomes = movements
    .filter(mov => mov > 0)
    .reduce((acc, mov) => acc + mov, 0);
  labelSumIn.textContent = `${incomes}€`;

  const out = movements
    .filter(mov => mov < 0)
    .reduce((acc, mov) => acc + mov, 0);
  labelSumOut.textContent = `${Math.abs(out)}€`;

  const interest = movements
    .filter(mov => mov > 0)
    .map(deposit => (deposit * 1.2) / 100)
    .filter((int, i, arr) => {
      console.log(arr);
      return int >= 1;
    })
    .reduce((acc, int) => acc + int, 0);
  labelSumInterest.textContent = `${interest}€`;
};

calcDisplaySummary(account1.movements);

```

It is bad practice to chain methods that mutate the array such as slice.

Here we can chain multiple methods in a function to create three constants for total incomes, total outgoings and total interest.

The Math.abs method takes the absolute number rather than the negative number.

The second filter method only takes interest values of 1 or more to be added into the array for the reduce method. A decimal value of less than 1 will not get added to the interest.

Chaining methods is very powerful but should only be used where absolutely necessary because a large array of values could cause a large memory overhead when working with arrays.

The Find Method

We can use the find method to find one element in the array based on a condition.

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

```
const firstWithdrawal = movements.find(mov => mov < 0);
```

```
console.log(movements);  
console.log(firstWithdrawal);
```

The find method is similar to the filter method but does not return an array, it returns a value based on the first instance where the condition is met in the array.

In this case -400 is the first instance in the array where `mov < 0`.

```
▶ (8) [200, 450, -400, 3000, -650, -130, 70, 1300]  
-400
```

```
console.log(accounts);
```

We can use the find method to find an object in our accounts array

```
▼ (4) [{...}, {...}, {...}, {...}] 1  
▶ 0: {owner: 'Jonas Schmedtmann', movements: Array(8), interestRate: 1.2, pin: 1111, username: 'js'}  
▶ 1: {owner: 'Jessica Davis', movements: Array(8), interestRate: 1.5, pin: 2222, username: 'jd'}  
▶ 2: {owner: 'Steven Thomas Williams', movements: Array(8), interestRate: 0.7, pin: 3333, username: 'stw'}  
▶ 3: {owner: 'Sarah Smith', movements: Array(5), interestRate: 1, pin: 4444, username: 'ss'}  
    length: 4  
▶ [[Prototype]]: Array(0)
```

```
const account = accounts.find(acc => acc.owner === 'Jessica Davis');
```

```
console.log(account);
```

```
let currentAccount;
```

The login system is triggered with an event handler click event. Note that the function has (e) in it.

```
btnLogin.addEventListener('click', function (e) {  
  // Prevent form from submitting  
  e.preventDefault();  
  currentAccount = accounts.find(  
    acc => acc.username === inputLoginUsername.value  
  );  
  console.log(currentAccount);
```

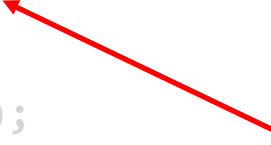
This is used to override html default behaviour. The username and pin are part of a form element and when a button is pressed in form the default behaviour is to reload the page. We disable this default behaviour.

```
  if (currentAccount?.pin === Number(inputLoginPin.value)) {  
    // Display UI and welcom message  
    labelWelcome.textContent = `Welcome back, ${currentAccount.owner.split(' ')[0]}`;  
    // clear input fields  
    inputLoginUsername.value = inputLoginPin.value = '';  
    inputLoginPin.blur();  
    // Display Movements  
    containerApp.style.opacity = 100;  
    displayMovements(currentAccount.movements);  
    // Display balance  
    calcDisplayBalance(currentAccount.movements);  
    // Display Summary  
    calcDisplaySummary(currentAccount.movements);  
  }  
});
```

If the pin is equal to the pin stored in the accounts object then the below functions are called to update the screen contents. Note the optional chaining.

```
const calcDisplaySummary = function (acc) {  
  const incomes = acc.movements  
    .filter(mov => mov > 0)  
    .reduce((acc, mov) => acc + mov, 0);  
  labelSumIn.textContent = `${incomes}€`;
```

Because each account has a different interest rate we want to use that value from the object rather than a default value so we pass in the entire account object here.



```
  const out = acc.movements  
    .filter(mov => mov < 0)  
    .reduce((acc, mov) => acc + mov, 0);  
  labelSumOut.textContent = `${Math.abs(out)}€`;
```

Now we change this from a default value to the account interest rate.



```
  const interest = acc.movements  
    .filter(mov => mov > 0)  
    .map(deposit => (deposit * acc.interestRate) / 100)  
    .filter((int, i, arr) => {  
      // console.log(arr);  
      return int >= 1;  
    })  
    .reduce((acc, int) => acc + int, 0);  
  labelSumInterest.textContent = `${interest}€`;  
};
```

When we call the calcDisplaySummary function as part of the login event listener we now pass in the CurrentAccount.



```
// Display Summary  
calcDisplaySummary(currentAccount);
```

Implementing Transfers storing the balance

To perform a transfer from an account we would like to be able to get the balance value from the account object but before we only calculated the balance.

```
const calcDisplayBalance = function (movements) {  
  const balance = movements.reduce((acc, mov) => acc + mov, 0);  
  labelBalance.textContent = `${balance}€`;  
};
```

To perform a transfer from an account we would like to be able to get the balance value from the account object but we did not store this value when performing the calcDisplayBalance function.

```
const calcDisplayBalance = function (acc) {  
  acc.balance = acc.movements.reduce((acc, mov) => acc + mov, 0);  
  labelBalance.textContent = `${acc.balance}€`;  
};
```

We can modify this function to now take account as a parameter then add a new acc.balance to the account object. Now balance will be stored in the object rather than a variable.

```
calcDisplayBalance(currentAccount.movements);
```

When we now call this function we only have to pass in the currentAccount, not the movements.

```
calcDisplayBalance(currentAccount);  
console.log(currentAccount);
```

Now balance is stored in the accounts object.

```
{owner: 'Jonas Schmedtmann', movements: Array(8),  
  1.2, pin: 1111, username: 'js', ...} ⓘ  
  balance: 3840  
  interestRate: 1.2  
  ▶ movements: (8) [200, 450, -400, 3000, -650, -130, 70, 1300]  
    owner: "Jonas Schmedtmann"  
    pin: 1111  
    username: "js"  
    ▶ [[Prototype]]: Object
```

Implementing Transfers event Listener

```
btnTransfer.addEventListener('click', function (e) {  
  e.preventDefault();  
  const amount = Number(inputTransferAmount.value);  
  const recieverAcc = accounts.find(acc => acc.username === inputTransferTo.value  
);  
  //console.log(amount, recieverAcc);  
  // validity conditions for transfer  
  if (  
    amount > 0 &&  
    recieverAcc &&  
    currentAccount.balance >= amount &&  
    recieverAcc.username !== currentAccount.username  
  ) {  
    // console.log('Transfer valid');  
    // Add a negative movement to the currentAccount  
    currentAccount.movements.push(-amount);  
    // Add a positive movement to the recieverAcc  
    recieverAcc.movements.push(amount);  
  }  
});
```

1) We add an event listener to the transfer button and store the amount and receiver account as variables.

2) We validate the transfer with a series of conditions. A) Is the amount greater than zero? B) Does the receiver account exist? C) Is the current Account balance greater than the transfer amount? D) Is the receiver account different from the current account?

3) push the transfer amount to the current account movements array as a negative number.
Push the transfer amount to the receiver account movements array as a positive number.

Implementing Transfers Refresh values

```
// Display Movements
displayMovements(currentAccount.movements);
// Display balance
calcDisplayBalance(currentAccount);
// Display Summary
calcDisplaySummary(currentAccount);

updateUI(currentAccount);
```

```
const updateUI = function (acc) {
  // Display Movements
  displayMovements(currentAccount.movements);
  // Display balance
  calcDisplayBalance(currentAccount);
  // Display Summary
  calcDisplaySummary(currentAccount);
};
```

The last action to perform for the transfer event listener is to update the UI.

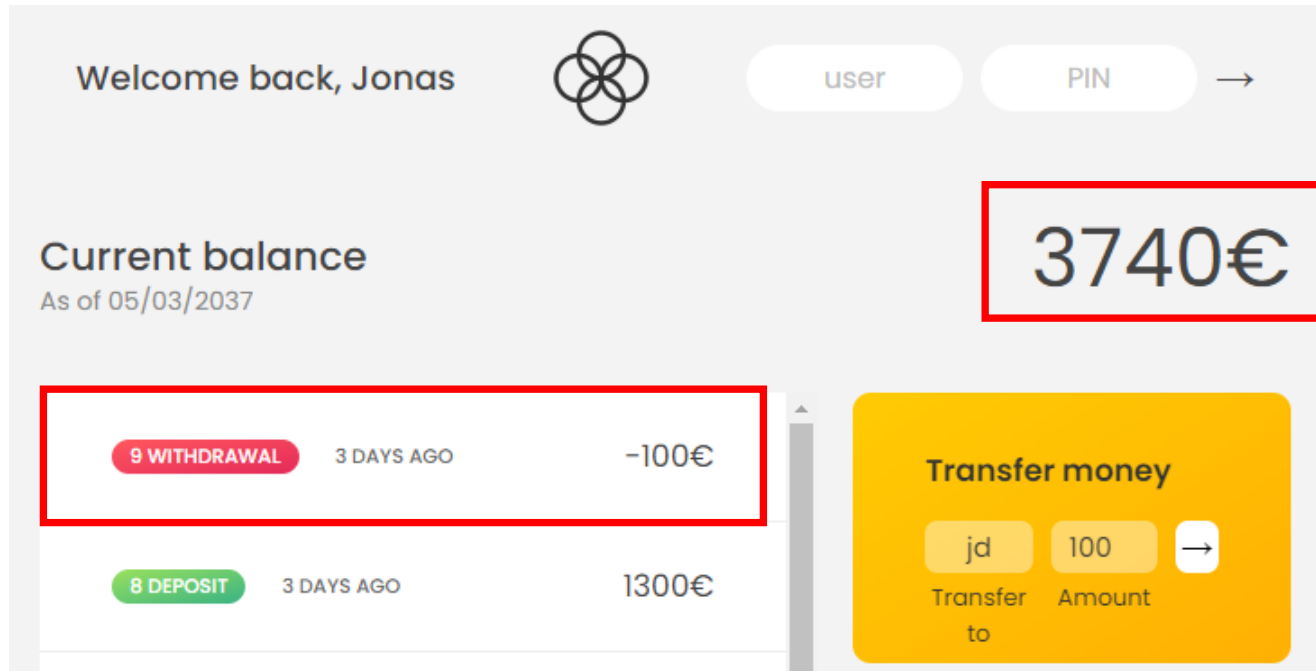
We want to refresh the values for the current account on the screen after performing the transfer. WE could just call these functions again but that would be bad practice. What we need to do is re-factor these into a separate function.

We will do this by replacing these three function calls with one call to a function called updateUI.

Now we have an updateUI function that we can call anywhere in our code when we want to refresh the screen values for the account.

```
.....
// Add a positive movement to the recieverAcc
recieverAcc.movements.push(amount);
// Update UI
updateUI(currentAccount);
}
});
```

Implementing Transfers check values



We see that if a transfer of 100 to jd is performed the `updateUI` function works.

Also the `account.movements` array shows a -100 new amount. The Jessica Davis `account.movements` shows a positive 100 new amount.

Final step is to clear the values in the transfer money block.

```
inputTransferAmount.value = inputTransferTo.value = '';
```

```
▼ (4) [{...}, {...}, {...}, {...}] ⓘ
  ▼ 0:
    balance: 3740
    interestRate: 1.2
    ▼ movements: Array(9)
      0: 200
      1: 450
      2: -400
      3: 3000
      4: -650
      5: -130
      6: 70
      7: 1300
      8: -100
      length: 9
    ▶ [[Prototype]]: Array(0)
    owner: "Jonas Schmedtmann"
    pin: 1111
    username: "js"
    ▶ [[Prototype]]: Object
  ▼ 1:
    interestRate: 1.5
    ▼ movements: Array(9)
      0: 5000
      1: 3400
      2: -150
      3: -790
      4: -3210
      5: -1000
      6: 8500
      7: -30
      8: 100
      length: 9
    ▶ [[Prototype]]: Array(0)
    owner: "Jessica Davis"
    pin: 2222
```


The findIndex method

```
const account4 = {  
  owner: 'Sarah Smith',  
  movements: [430, 1000, 700, 50, 90],  
  interestRate: 1,  
  pin: 4444,  
};
```

```
const accounts = [account1, account2, account3, account4];
```

```
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);  
  }  
});
```

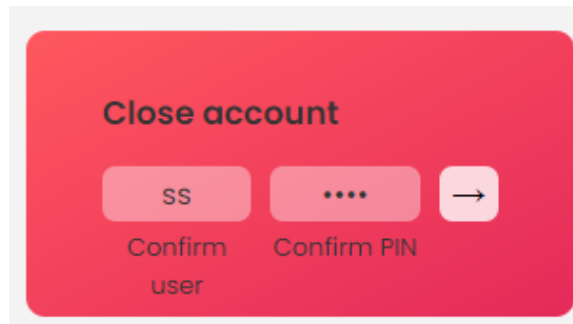
findIndex returns a value of 3 which corresponds to the Sarah Smith Object.

The find index method is almost identical to the find method except that the findIndex returns the index whereas the find returns the value.

A good use for the findIndex method is for account deletion where we can use the splice method to remove account4 from the array but we need to index – findIndex!

An event listener is created for the close button and the inputs are validated with an if statement.

Index variable is defined and the findIndex method is used to find the object in the accounts array.



```
▼ Object 1  
  balance: 2270  
  interestRate: 1  
  ► movements: (5) [430, 1000, 700, 50, 90]  
  owner: "Sarah Smith"  
  pin: 4444  
  username: "ss"  
  ► [[Prototype]]: Object
```

```
const index = accounts.findIndex(condition
  acc => acc.username === currentAccount.username
);
```

The find index method returns the **index** of the **first** element in the array which matches the condition.

```
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);
    // Delete Account
    accounts.splice(index, 1);
    // Hide UI
    containerApp.style.opacity = 0;
    console.log(accounts);
  }
  // clear input values
  inputCloseUsername.value = inputClosePin.value = '';
});
```

Now we just add the actions to perform to delete the account from the accounts array.

- 1) Use splice to remove that && index from the array.
- 2) Hide UI by setting containerApp opacity to zero.

```
▼ (3) [{...}, {...}, {...}] ⓘ
  ▶ 0: {owner: 'Jonas Schmedtmann', movements: Array(8), interestRate: 1.2, pin: 1111,
  ▶ 1: {owner: 'Jessica Davis', movements: Array(8), interestRate: 1.5, pin: 2222, user
  ▶ 2: {owner: 'Steven Thomas Williams', movements: Array(8), interestRate: 0.7, pin: :
    length: 3
  ▶ [[Prototype]]: Array(0)
```

Note how the accounts array now only has 3 users in it and Sarah Smith has been removed.

Last action is to clear the input values from the close form.

The Some and Every methods (ES6)

We can use includes metho to check if a value is in an array

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

```
console.log(movements);  
console.log(movements.includes(-130));
```

► (8) [200, 450, -400, 3000, -650, -130, 70, 1300]
true

But includes can only really test for equality so includes will return true if any value in the array is exactly equal to -130. Some can be used to test for a condition.

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

true

```
const depositsFifteen = movements.some(mov => mov > 1500);  
console.log(depositsFifteen);
```

true

With some we can check for a condition such as greater than.

WE can use the some method in out Bankist project to implement the loan functionality where our virtual bank will only grant a loan if there is at least one deposit in the account that is equal to or greater than 10% of the user requested loan amount.

```
btnLoan.addEventListener('click', function (e) {  
  e.preventDefault();
```

We create an event listener for the loan button and put the amount into a variable.

```
    const amount = Number(inputLoanAmount.value);  
    if (amount > 0 && currentAccount.movements.some(mov => mov >= amount * 0.1));  
    // add loan  
    currentAccount.movements.push(amount);  
    // update UI  
    updateUI(currentAccount);  
    // clear input value  
    inputLoanAmount.value = '';  
    console.log(currentAccount.movements);  
  });
```

We validate the amount in an if statement checking it is greater than zero. Also the some method is used to check if the loan amount is greater than 10% of any existing value in the movements array.

We perform three actions if amount passes validation: a) push loan amount to movements array. b) updateUI. c) clear input value.

A loan amount of 1000 was requested.
10% of 1000 is 100 so there must be an existing movement of 100 or more for some method condition to be met.

```
▼ (9) [200, 450, -400, 3000, -650, -130, 70, 1300, 1000] ⓘ  
  0: 200  
  1: 450  
  2: -400  
  3: 3000  
  4: -650  
  5: -130  
  6: 70  
  7: 1300  
  8: 1000  
  length: 9  
  ► [[Prototype]]: Array(0)
```

```
console.log(movements.every(mov => mov > 0));           false
console.log(account4.movements);                         ► (5) [430, 1000, 700, 50, 90]
console.log(account4.movements.every(mov => mov > 0));    true
```

The every method returns true if every element in the array meets the condition.

For example in movements we have negative and positive values so these do not meet the condition of being greater than zero.

Account4.movements only has positive values above zero so does meet the condition.

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

We can also write the function separately and pass it as a function.

In this example the console.log functions are callbacks to the deposit function. They all reuse the same function of deposit.

The flat and flatMap methods (ES2019)

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

The flat method will flatten the contents of nested arrays into one array.

```
▼ (4) [Array(2), Array(2), 7, 8] ⓘ  
  ▶ 0: (2) [Array(2), 3]  
  ▶ 1: (2) [4, Array(2)]  
    2: 7  
    3: 8  
    length: 4  
  ▶ [[Prototype]]: Array(0)
```

```
▼ (6) [Array(2), 3, 4, Array(2), 7, 8] ⓘ  
  ▶ 0: (2) [1, 2]  
    1: 3  
    2: 4  
  ▶ 3: (2) [5, 6]  
    4: 7  
    5: 8  
    length: 6  
  ▶ [[Prototype]]: Array(0)
```

```
const arrDeep2 = [[[1, 2], 3], [4, [5, 6]], 7, 8];  
console.log(arrDeep2.flat(2));
```

```
▶ (8) [1, 2, 3, 4, 5, 6, 7, 8]
```

```
▼ (4) [Array(3), Array(3), 7, 8] ⓘ  
  ▶ 0: (3) [1, 2, 3]  
  ▶ 1: (3) [4, 5, 6]  
    2: 7  
    3: 8  
    length: 4  
  ▶ [[Prototype]]: Array(0)
```

```
▼ (8) [1, 2, 3, 4, 5, 6, 7, 8] ⓘ  
  0: 1  
  1: 2  
  2: 3  
  3: 4  
  4: 5  
  5: 6  
  6: 7  
  7: 8  
  length: 8  
  ▶ [[Prototype]]: Array(0)
```

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

The flat method only goes one level deep when flattening an array. Note how the nested arrays two levels down are still inside an array.

We can specify the depth to flatten to reduce deep nested arrays to one array.

```
console.log(accounts);
const accountMovements = accounts.map(acc => acc.movements);
console.log(accountMovements);
```

We have an array of accounts which contains user objects. Within each user object is an array called movements. We want to extract the movements from each object and put it in a single nested array. Array.map will do this.

▼ (4) [{...}, {...}, {...}, {...}] ⓘ

```
▶ 0: {owner: 'Jonas Schmedtmann', movements: Array(8), interestRate: 1.2, pin: 1111, username: 'js'}
▶ 1: {owner: 'Jessica Davis', movements: Array(8), interestRate: 1.5, pin: 2222, username: 'jd'}
▶ 2: {owner: 'Steven Thomas Williams', movements: Array(8), interestRate: 0.7, pin: 3333, username: 'stw'}
▶ 3: {owner: 'Sarah Smith', movements: Array(5), interestRate: 1, pin: 4444, username: 'ss'}
    length: 4
    [[Prototype]]: Array(0)
```

accounts

▼ (4) [Array(8), Array(8), Array(8), Array(5)] ⓘ

```
▶ 0: (8) [200, 450, -400, 3000, -650, -130, 70, 1300]
▶ 1: (8) [5000, 3400, -150, -790, -3210, -1000, 8500, -30]
▶ 2: (8) [200, -200, 340, -300, -20, 50, 400, -460]
▶ 3: (5) [430, 1000, 700, 50, 90]
    length: 4
    [[Prototype]]: Array(0)
```

accountMovements

```
const allMovements = accountMovements.flat();
console.log(allMovements);
```

```
const overallBalance = allMovements.reduce((acc, mov) => acc + mov, 0);
console.log(overallBalance);
```

Array.flat can be used to flatten all the nested arrays into one and array.reduce can be used to total array values.

```
(29) [200, 450, -400, 3000, -650, -130, 70, 1300, 5000,
▶ 3400, -150, -790, -3210, -1000, 8500, -30, 200, -200, 34
0, -300, -20, 50, 400, -460, 430, 1000, 700, 50, 90]
```

17840

```
const totalMovments = accounts
  .map(acc => acc.movements)
  .flat()
  .reduce((acc, mov) => acc + mov, 0);

console.log(totalMovments);
```

```
const totalMovments2 = accounts
  .flatMap(acc => acc.movements)
  .reduce((acc, mov) => acc + mov, 0);

console.log(totalMovments2);
```

The methods of `array.map`, `array.flat` and `array.reduce` can be chained together to produce a single function that calculates the total of all the movements in all the accounts.

This is a pretty common use of `map` then `flat`.

The methods of `array.map` and `array.flat` can be combined into one by using the method `array.flatMap`. Note however that `flatMap` only goes one level deep in nested arrays so if we need to flatten deep levels then we need to use the `array.flat(n)` and specify `n` as the number of layers.

Sorting Arrays

```
// strings
```

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

Array.sort will sort an array of strings alphabetically but it will mutate the original array.

```
▶ (4) ['Adam', 'Jonas', 'Martha', 'Zach']
```

```
▶ (4) ['Adam', 'Jonas', 'Martha', 'Zach']
```

```
// Numbers
```

```
const movements = [200, 450, -400, 3000, -650, -130, 70, 1300];  
console.log(movements.sort());  
console.log(movements);
```

Array.sort performed on an array of numbers will also sort the values alphabetically and mutate the original array.

```
▶ (8) [-130, -400, -650, 1300, 200, 3000, 450, 70]
```

```
▶ (8) [-130, -400, -650, 1300, 200, 3000, 450, 70]
```

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

There is a workaround for sorting numerically using a compare callback function that returns 1 or -1 within a loop depending if a is greater than b.

```
▶ (8) [-650, -400, -130, 70, 200, 450, 1300, 3000]
```

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

And to reverse the sorting order we can simply swap the 1 and -1.

```
▶ (8) [3000, 1300, 450, 200, 70, -130, -400, -650]
```

```
movements.sort((a, b) => a - b);  
console.log(movements);
```

Sorting numerically can be simplified because if a is greater than b it will return a positive number. If a is less than b then it will return a negative number.

```
► (8) [-650, -400, -130, 70, 200, 450, 1300, 3000]
```

```
movements.sort((a, b) => b - a);  
console.log(movements);
```

And to sort numerically from highest to lowest value we just invert a and b.

```
► (8) [3000, 1300, 450, 200, 70, -130, -400, -650]
```

The array.sort method will not work on a mixed array of strings and numbers.

```
const displayMovements = function (movements) {  
  containerMovements.innerHTML = '';  
  movements.forEach(function (mov, i) {  
    const type = mov > 0 ? 'deposit' : 'withdrawal';  
  
    const html = `  
      <div class="movements__row">  
        <div class="movements__type movements__type--${type}">${i + 1} ${type}</div>  
        <div class="movements__date">3 days ago</div>  
        <div class="movements__value">${mov}€</div>  
      </div>  
    `;  
    containerMovements.insertAdjacentHTML('afterBegin', html);  
  });  
};
```

The array.sort method is useful on our bankist project to sort deposits and withdrawals by value so we will modify the displayMovements function.

```
const displayMovements = function (movements, sort = false) {  
  containerMovements.innerHTML = '';
```

DisplayMovements function is modified to take an additional parameter called sort which is set to false.

```
const movs = sort ? movements.slice().sort((a, b) => a - b) : movements;
```

Variable 'movs' is created so that if sort = true then it will make a copy of the movements array with slice then sort the order numerically.

```
movs.forEach(function (mov, i) {  
  const type = mov > 0 ? 'deposit' : 'withdrawal';
```

Now we need to use the movs array that we just created rather than the movements array in the forEach loop.

```
const html = `  
  <div class="movements__row">  
    <div class="movements__type movements__type--${type}">${i + 1} ${type}</div>  
    <div class="movements__date">3 days ago</div>  
    <div class="movements__value">${mov}€</div>  
  </div>  
`;  
  
containerMovements.insertAdjacentHTML('afterBegin', html);  
});  
};
```

```
btnSort.addEventListener('click', function (e) {  
  e.preventDefault();  
  displayMovements(currentAccount.movements, true);  
});
```

We add an event listener to the sort button that calls the displayMovements function with the parameter of sort set to true.

Now the sort button at the bottom of the movements window performs a sort of values when clicked but if we click it again it does not undo the sort.

We can do this by defining a state variable outside of the sort button event listener And then inverting it after calling the sort function.

```
let sorted = false;  
btnSort.addEventListener('click', function (e) {  
  e.preventDefault();  
  displayMovements(currentAccount.movements, !sorted);  
  sorted = !sorted;  
});
```

This acts like a toggle where the sorted value is switched between false and true before and after calling the displayMovements function.

More ways of creating and filling Arrays

We can create new arrays by using one of the top two methods.

```
const arr = [1, 2, 3, 4, 5, 6, 7];  
console.log(new Array(1, 2, 3, 4, 5, 6, 7));
```

► (7) [1, 2, 3, 4, 5, 6, 7]

► (7) [1, 2, 3, 4, 5, 6, 7]

► (7) [empty × 7]

If we create a new array with just one value of let's say 7 we would expect an array one item containing value seven but what we actually get is an array of seven empty items. This is a peculiarity of JavaScript.

Because we have an empty array we cannot use any methods on it except one – the fill method.

```
x.fill(1);  
console.log(x);  
Or
```

► (7) [1, 1, 1, 1, 1, 1, 1]

The fill method will populate all positions in the array with the value specified.

```
x.fill('cats');  
console.log(x);
```

► (7) ['cats', 'cats', 'cats', 'cats', 'cats', 'cats', 'cats']

```
x.fill('cats', 1, 3);  
console.log(x);
```

► (7) [empty, 'cats', 'cats', empty × 4]

We can also add in two optional parameters like slice to specify where to start the fill and where to end the fill.

```
arr.fill(23, 2, 6);  
console.log(arr);
```

► (7) [1, 2, 23, 23, 23, 23, 7]

Fill will mutate an existing array, for example to overwrite 23 to position 2 to 6.

```
const y = Array.from({ length: 7 }, () => 1);  
console.log(y);
```

```
▶ (7) [1, 1, 1, 1, 1, 1, 1]
```

We can use the Array function with the from method and pass in an length object and a blank call back function that returns 1.

```
const z = Array.from({ length: 7 }, (cur, i) => i + 1);  
console.log(z);
```

```
▶ (7) [1, 2, 3, 4, 5, 6, 7]
```

This time we can use the call back function to increment I by one on each iteration.

```
const z = Array.from({ length: 7 }, (_, i) => i + 1);  
console.log(z);
```

```
▶ (7) [1, 2, 3, 4, 5, 6, 7]
```

Because we are not using the cur parameter it is convention that we mark it with an underscore.

Array.from was created to be able to build arrays from array like structures, i.e. itterables.

```
labelBalance.addEventListener('click', function () {  
  const movementsUI = Array.from(  
    document.querySelectorAll('.movements__value')  
  );  
  console.log(movementsUI);  
});
```

We can add an event listener to any html element on a page.

On click we will create an array of html elements with all their properties.

```
(8) [div.movements__value, div.movements__value, div.movements__value,  
▶ div.movements__value, div.movements__value, div.movements__value,  
div.movements__value, div.movements__value]
```

```
labelBalance.addEventListener('click', function () {  
  const movementsUI = Array.from(  
    document.querySelectorAll('.movements__value'),  
    el => Number(el.textContent.replace('€', ''))  
  );  
  console.log(movementsUI);  
});
```

```
► (8) [1300, 70, -130, -650, 3000, -400, 450, 200]
```

This can further be modified to chain in a function that will convert the html element to a number and remove the euro symbol.

In conclusion we used `Array.from` to convert the nodelist of html elements for `movements__value` into an array. We then used the `replace` method on this array to remove the euro symbols.

```
const movementsUI2 =  
[...document.querySelectorAll('.movements__value')];
```

The spread operator could also be used in place of the `Array.from` but we then could not chain on the `replace` method.

Summary: Which Array Method to use?

To mutate original Array

- Add to original

.push

(end)

.unshift

(start)

- Remove from original

.Pop

(end)

.shift

(start)

.splice

(any)

- Others

.reverse

.sort

.fill

A new Array

- Compute from original

.map

(loop)

- Filter using Condition

.filter

- Portion of original

.slice

- Adding original to other

.concat

- Flattening the original

.flat

.flatMap

An array Index

- Based on Value

.indexOf

- Based on test Condition

.findIndex

An array Element

- Based on test Condition

.find

Know if an array includes

- Based on Value

.includes

- Based on test Condition

.some

.every

An new string

- Based on a separator string

.join

To Transform to value

- Based on accumulator

.reduce

(boil down an array to a single value of any type: number, string, Boolean or even new array or object)

To just loop array

- Based on callback

.forEach

(Does not create a new array, just loops over it)

Array Methods practice

```
const bankdespositSum = accounts.map(acc => acc.movements).flat();
console.log(bankdespositSum);
```

```
const bankdespositSum2 = accounts.flatMap(acc => acc.movements);
console.log(bankdespositSum2);
```

```
const bankdespositSum3 = accounts
  .flatMap(acc => acc.movements)
  .filter(mov => mov > 0)
  .reduce((sum, cur) => sum + cur, 0);
console.log(bankdespositSum3);
```

```
// method 1
const numDesposits1000 = accounts
  .flatMap(acc => acc.movements)
  .filter(mov => mov > 1000).length;
console.log(numDesposits1000);
```

```
// method 2
const numDesposits10002 = accounts
  .flatMap(acc => acc.movements)
  .reduce((count, cur) => (cur >= 1000 ? count + 1 : count), 0);
console.log(numDesposits10002);
```

1. FlatMap, Filter and reduce methods to get a total sum of deposits.

2. Two methods to Count how many desposits in the bank over 1000

```
const { deposits, withdrawals } = accounts
  .flatMap(acc => acc.movements)
  .reduce(
    (sums, cur) => {
      sums[cur > 0 ? 'deposits' : 'withdrawals'] += cur;
      return sums;
    },
    { deposits: 0, withdrawals: 0 }
  );
console.log(deposits, withdrawals);
```

3. use reduce method to return an object.

```
const convertTitleCase = function (title) {
  const capitalize = str => str[0].toUpperCase() + str.slice(1);

  const exceptions = [ 'a', 'an', 'and', 'the', 'but', 'or', 'on', 'in', 'with', 'is' ];

  const titleCase = title
    .toLowerCase()
    .split(' ')
    .map(word => (exceptions.includes(word) ? word : capitalize(word)))
    .join(' ');

  return capitalize(titleCase);
};
console.log(convertTitleCase('this is a nice title'));
console.log(convertTitleCase('this is a LONG title but not too long'));
console.log(convertTitleCase('and here is another title with an EXAMPLE'));
```

4. Capitalise only important words in a string leaving the exceptions in lowercase.

Numbers, Dates, Intl and
Timers

Converting and Checking numbers

```
console.log(23 === 23.0); true
```

```
console.log(0.1 + 0.2); 0.30000000000000004
```

0.1 plus 0.2 should equal 0.3 but the binary conversion produces a small inaccuracy. This is normal in coding languages like php, JavaScript, ruby etc. Sometimes JavaScript will do some behind the scenes correction by rounding but not in this case so it makes it an inappropriate language for really precise scientific or financial calculations.

```
console.log(Number('23')); 23
```

```
console.log(+ '23'); 23
```

Converting a string to a number is easy in JavaScript. Also when JavaScript sees a plus symbol it will do type coercion of a string into a number. This can be used to make code neater by substituting the inbuilt function of `Number()` with the plus symbol.

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

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

The second parameter by default is set to base 10 i.e. 0-9 but we can also set it to binary base 2. `parseInt` will parse an integer, i.e. whole number.

In JavaScript all numbers are represented as floating point numbers, i.e. Decimal.

In JavaScript numbers are represented as binary i.e. 64 base 2. This produces weird results sometimes with fractions.

JavaScript can be used to parse a string into a number and can remove unnecessary letters. However the number in the string must come before the letters or it will return not a number.

```
console.log(Number.parseInt('2.5rem '));
```

2

parseInt will only return an integer. for decimal numbers we can use parseFloat.

```
console.log(Number.parseFloat('2.5rem '));
```

2.5

Note that The parsing function will also remove blank spaces.

```
console.log(Number.isNaN(20));
```

false

isNaN will check if it is a NOT A NUMBER and return a Boolean response.

```
console.log(Number.isNaN('20'));
```

false

```
console.log(Number.isNaN(+ '20px'));
```

true

isNaN returns true because 20px is not a number.

```
console.log(Number.isNaN(23 / 0));
```

false

Dividing by zero gives us infinity which JavaScript considers a number so isNaN is false.

```
console.log(Number.isFinite(23 / 0));
```

false

A better way to check if it is a number: isFinite which will return false for infinity.

```
console.log(Number.isInteger(23));
```

true

```
console.log(Number.isInteger(23.0));
```

true

Finally, we can check if a value is a whole number using the isInteger method.

```
console.log(Number.isInteger(23 / 0));
```

false

Math and Rounding

```
console.log(Math.sqrt(25));
```

5

Math.Sqrt will return the square root of a number.

```
console.log(25 ** (1 / 2));
```

5

It can also be written as $n ** (1 / 2)$. There is not inbuilt method to get the cube root of a number but it can be written as $n ** (1 / 3)$.

```
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

Math.max will find the maximum number and also perform automatic type coercion of strings to numbers, except where a string contains letters

```
console.log(Math.min(5, 18, 23, 11, 2));
```

2

Math.min will find the minimum number.

```
console.log(Math.PI);
```

3.141592653589793

```
console.log(Math.PI + parseFloat('10px') ** 2);
```

103.1415926535898

JavaScript also has PI built in. The formula for an area is Pi radius squared so we can calculate the area of an element in pixels.

```
console.log(Math.trunc(Math.random() * 6) + 1);
```

6

```
const randomInt = (min, max) =>  
  Math.floor(Math.random() * (max - min) + 1) + min;
```

```
console.log(randomInt(1, 6));
```

4

```
console.log(randomInt(10, 20));
```

12

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

23

```
console.log(Math.round(23.9));
```

24

```
console.log(Math.round(23.5));
```

24

```
console.log(Math.round(23.4));
```

23

```
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

Math.random will generate a random decimal number between 0 and 1.

Math.trunc() function returns the integer part of a number by removing any fractional digits.

The above can be written as a function that returns a random number between a max and min.

Math.trunc will just remove the decimal part whereas **Math.round** will round up or down to the nearest integer.

Math.ceil (ceiling) will always round up to the nearest integer.

Math.floor will always round down to the nearest integer. This is similar to **math.trunc** except for negative numbers.

All these functions do automatic type coercion

```
console.log(Math.ceil(-23.3)); 23  
console.log(Math.ceil(-23.9)); 23
```

Math.ceil (ceiling) will always round up to the nearest integer and in the case of a negative number the highest integer is the same as `math.trunc`.

```
console.log(Math.floor(-23.3)); 24  
console.log(Math.floor(-23.9)); 24
```

Math.floor will always round up to the nearest integer and in the case of a negative number the highest integer is the same as `math.trunc + 1`.

```
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));
```

toFixed is used to round decimal parts or fractions. We call the method on the number, specifying the number of places to round to. Note that the output is a string and not a number so we would have to use `plus` to return an actual number.


```
btnLoan.addEventListener('click', function (e) {  
  e.preventDefault();
```

```
  const amount = Math.floor(inputLoanAmount.value);
```

```
  if (amount > 0 && currentAccount.movements.some(mov => mov >= amount * 0.1)) {  
    // Add movement  
    currentAccount.movements.push(amount);
```

```
    // Update UI
```

```
    updateUI(currentAccount);
```

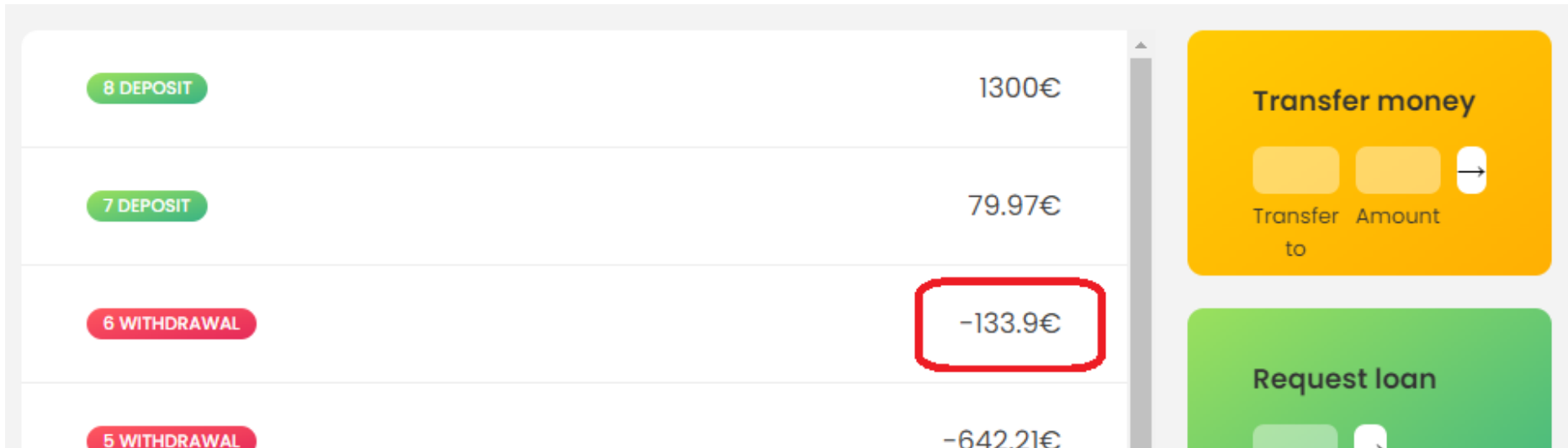
```
  }
```

```
  inputLoanAmount.value = '';
```

```
});
```

We can use `math.Floor` in our Bankist app to round down any loan request to an integer.

We can use `toFixed` to display currencies to always two decimal places.



```
const displayMovements = function (movements, sort = false) {
  containerMovements.innerHTML = '';

  const movs = sort ? movements.slice().sort((a, b) => a - b) : movements;

  movs.forEach(function (mov, i) {
    const type = mov > 0 ? 'deposit' : 'withdrawal';

    const html = `
      <div class="movements__row">
        <div class="movements__type movements__type--${type}">${i + 1} ${type}</div>
        <div class="movements__value">${mov.toFixed(2)}€</div>
      </div>
    `;

    containerMovements.insertAdjacentHTML('afterbegin', html);
  });
};
```

```
const calcDisplayBalance = function (acc) {  
  acc.balance = acc.movements.reduce((acc, mov) => acc + mov, 0);  
  labelBalance.textContent = `${acc.balance.toFixed(2)}€`;  
};
```

```
const calcDisplaySummary = function (acc) {  
  const incomes = acc.movements  
    .filter(mov => mov > 0)  
    .reduce((acc, mov) => acc + mov, 0);  
  labelSumIn.textContent = ``${incomes.toFixed(2)}€`;
```

```
const out = acc.movements  
  .filter(mov => mov < 0)  
  .reduce((acc, mov) => acc + mov, 0);  
labelSumOut.textContent = ``${Math.abs(out).toFixed(2)}€`;
```

```
const interest = acc.movements  
  .filter(mov => mov > 0)  
  .map(deposit => (deposit * acc.interestRate) / 100)  
  .filter((int, i, arr) => {  
    // console.log(arr);  
    return int >= 1;  
  })  
  .reduce((acc, int) => acc + int, 0);  
labelSumInterest.textContent = ``${interest.toFixed(2)}€`;
```

The Remainder Operator

<code>console.log(5 % 2);</code>	<code>1</code>	5 divided by two is 2.5. but if we did this as an integer division then 5 / 2 would be two with 2 remaining.
<code>console.log(5 / 2);</code>	<code>2.5</code>	

<code>console.log(8 % 3);</code>	<code>2</code>	8 divided by 3 is 2.6666 but if we divide 8 by 3 getting only whole number results then it would be 3 x 2 = 6 with 2 remaining.
<code>console.log(8 / 3);</code>	<code>2.6666666666666665</code>	

<code>console.log(6 % 2);</code>	<code>0</code>	An even number is divisible by two with no remainder i.e. the remainder function returns zero. An odd number is not divisible by two and will always have a remainder of 1. We can use this knowledge to create a function.
<code>console.log(6 / 2);</code>	<code>3</code>	
<code>console.log(7 % 2);</code>	<code>1</code>	
<code>console.log(7 / 2);</code>	<code>3.5</code>	

```
const isEven = n => n % 2 === 0;
console.log(isEven(8));    true
console.log(isEven(23));   false
console.log(isEven(514));  true
```

This function returns true if the number is even and false if the number is odd.

We can also use this to set even rows background color in the bankist app. When we click on the balance value if the background colour for each row (i). orangered (even), blue (every third row).

```
labelBalance.addEventListener('click', function () {
  [...document.querySelectorAll('.movements__row')].forEach(function (row, i) {
    if (i % 2 === 0) row.style.backgroundColor = 'orangered';
    if (i % 3 === 0) row.style.backgroundColor = 'blue';

  });
});
```

Welcome back, Jonas



user

PIN



Current balance

As of 05/03/2037

25952.59€

8 DEPOSIT

1300.00€

7 DEPOSIT

79.97€

6 WITHDRAWAL

-133.90€

5 WITHDRAWAL

-642.21€

Transfer money

Transfer to

Amount



Request loan

Amount



For even's (orangered) the row (i) values are 0,2,4,8,10 etc.... For every third row (blue) the row (i) values are 0,3,6,9,12 etc... the remainder operator is good for doing something every nth time.

Numeric Separators (ES2021)

We can represent really large numbers in JavaScript using numeric Separators, For example the diameter of the solar system is 287,460,000,000km. Written in human readable coma separated format it is easy to understand compared with written as a plain. We can break a long number up using the underscore to perhaps represent thousands, like a coma would in human readable form. Note that JavaScript interprets underscores in numbers as a visual aid and removes them for calculations, output etc.

```
const diameter = 287460000000;  
const diameter2 = 287_460_000_000;  
console.log(diameter2);  
287460000000
```

```
const priceCents = 345_99;  
console.log(priceCents);  
34599
```

```
const transferFee = 15_00;  
console.log(transferFee);  
1500
```

If we use a numeric separator to replace a decimal then the number is interpreted by JavaScript as an integer.

```
const PI = 3.141_592;  
console.log(PI);  
3.141592
```

```
const PI = 3_141592;  
console.log(PI);  
3141592
```

```
const PI = _3.141592;  
console.log(PI);
```

✖ Uncaught SyntaxError: Unexpected number (at script.js:371:14)

```
const PI = 3.141592_;  
console.log(PI);
```

✖ Uncaught SyntaxError: Numeric separators are not allowed at the end of numeric literals (at script.js:371:21)

```
const PI = 3.141__592;  
console.log(PI);
```

✖ Uncaught SyntaxError: Only one underscore is allowed as numeric separator (at script.js:371:18)

Numeric Separators cannot go at the beginning or end of a number and we cannot use more than one numeric separator such as a double underscore.

```
console.log(Number('230_000'));  
NaN
```

Using numeric separators in a string that is then converted to a number does not work.

Big Int (ES2020)

```
console.log(2 ** 53 - 1);  
console.log(Number.MAX_SAFE_INTEGER);
```

```
9007199254740991
```

```
9007199254740991
```

Javascript has a maximum integer number which is two to the power of 53 minus 1.

It is even stored in the Number namespace as MAX_SAFE_INTEGER.

A number above the MAX_SAFE_INTEGER may produce unexpected problems in JavaScript because it cannot process numbers above this value. An example might be that JavaScript is receiving data from an external API database with big integer values.

```
console.log(83834930339063793679367937693);  
console.log(83834930339063793679367937693n);  
console.log(BigInt(83834930339063793679367937693));
```

```
8.38349303390638e+28
```

```
83834930339063793679367937693n
```

```
83834930339063795180151767040n
```

To store really big integers above the max safe integer value we can use an n after the number to signify that it is the new primitive type of big integer. The bigint function does the same to specify that a number is a big integer above the MAX_SAFE_INTEGER.

```
console.log(10000n + 10000n);  
console.log(83834930339063793679367937693n * 100n);
```

```
20000n
```

```
8383493033906379367936793769300n
```

We can perform mathematical operations with big Integer's.

```
const huge = 83834930339063793679367937693n;  
const small = 23;  
console.log(huge - small);
```

```
✖ ▶ Uncaught TypeError: Cannot mix      script.js:390  
   BigInt and other types, use explicit conversions  
   at script.js:390:18
```

But we cannot mix big Integers with normal numbers.

```
console.log(20n > 15);      true  
console.log(20n === 20);    false  
console.log(typeof 20n);    bigint
```

We can compare bigint's with normal numbers because JavaScript does not do type coercion between the bigint and the normal number

```
console.log(20n == 20);      true  
console.log(huge + 'is REALLY big!!!'); 83834930339063793679367937693is  
                                         REALLY big!!!
```

If we do a loose comparison with two equals then it does do type coercion and recognises the bigint of 20n as just a normal number of 20. If we concatenate the huge number with a string then JavaScript does type coercion to turn the bigint into a string. Note how it is not displayed with an n at the end.

```
console.log(Math.sqrt(16n));
```

```
✖ ▶ Uncaught TypeError: Cannot convert script.js:399  
   a BigInt value to a number  
   at Math.sqrt (<anonymous>)  
   at script.js:399:18
```

We cannot use Math methods on bigint's

```
console.log(10n / 3n);      3n  
console.log(11n / 3n);      3n  
console.log(12n / 3n);      4n
```

We can do division with bigint's. Note that it simply cuts off the decimal part.

Creating Dates

Creating dates and times in Javascript can be a bit messy but there are 4 main methods for doing it.

```
const now = new Date(); Thu May 12 2022 17:14:38 GMT+0200  
console.log(now);      (Central European Summer Time)
```

1. Create a new variable and set it to the current date and time.

```
console.log(new Date('Aug 02 2020 18:05:41'));
```

```
Sun Aug 02 2020 18:05:41 GMT+0200  
(Central European Summer Time)
```

```
console.log(new Date('December 24, 2015'));
```

```
Thu Dec 24 2015 00:00:00 GMT+0100  
(Central European Standard Time)
```

```
console.log(new Date(account1.movementsDates[0])); Mon Nov 18 2019 22:31:17 GMT+0100  
(Central European Standard Time)
```

2a. We can manually set the date and time by passing in a string to the date function.

2b. JavaScript will even calculate the missing data like day when we pass in a string manually.

3c. We can call the date and time from an object.

```
console.log(new Date(2037, 10, 19, 15, 23, 5));
```

```
Thu Nov 19 2037 15:23:05 GMT+0100  
(Central European Standard Time)
```

```
console.log(new Date(2037, 10, 31));
```

```
Tue Dec 01 2037 00:00:00 GMT+0100  
(Central European Standard Time)
```

```
console.log(new Date(2037, 10, 33));
```

```
Thu Dec 03 2037 00:00:00 GMT+0100  
(Central European Standard Time)
```

3a. We can pass in the date and time as comma separated values. Note that month is 10 = November, the eleventh month of the year. Months are zero based i.e. January is month zero.

4b & 4c. JavaScript will autocorrect. I.e. November does not have 31 days so JS assumes we mean 1st of December. 33rd of November will get auto corrected to 3rd of December.

```
console.log(new Date(0));
```

```
Thu Jan 01 1970 01:00:00 GMT+0100  
(Central European Standard Time)
```

```
console.log(new Date(3 * 24 * 60 * 60 * 1000));
```

```
Sun Jan 04 1970 01:00:00 GMT+0100  
(Central European Standard Time)
```

4a. Unix date time started on Jan 1st 1970 at 00:00:00 GMT or UTC.

4b. We can calculate a new date by making a time stamp of 3 days times 24 hours times 60 minutes times 60 seconds times 1000 milliseconds.

Dates are just another type of objects and have their own special methods

```
const future = new Date(2037, 10, 19, 15, 23);  
console.log(future);  
console.log(future.getFullYear());  
console.log(future.getMonth());  
console.log(future.getDate());  
console.log(future.getDay());  
console.log(future.getHours());  
console.log(future.getMinutes());  
console.log(future.getSeconds());
```

```
Thu Nov 19 2037 15:23:00 GMT+0100  
(Central European Standard Time)
```

```
2037
```

```
10
```

```
19
```

4 **getDay** is day of the week and is zero based

```
15
```

```
23
```

```
0
```

```
console.log(future.toISOString());
```

```
2037-11-19T14:23:00.000Z
```

toISOString converts the date time into a standard ISO international format.

```
console.log(future.getTime()); 2142253380000
```

The `getTime` method will produce a timestamp in milliseconds. A time stamp is the value of time that has passed since the start of Unix time (Jan 1st 1970).

```
console.log(new Date(2142253380000));
```

Thu Nov 19 2037 15:23:00 GMT+0100
(Central European Standard Time)

The timestamp can be used to find a date Time

```
console.log(Date.now()); 1652370630188
```

Time stamps are so important that there is even a method to get the current timestamp.

```
future.setFullYear(2040);  
console.log(future);
```

Mon Nov 19 2040 15:23:00 GMT+0100
(Central European Standard Time)

There are methods to set the year, month, day, hour, seconds, milliseconds.

```
// FAKE ALWAYS LOGGED IN  
currentAccount = account1;  
updateUI(currentAccount);  
containerApp.style.opacity = 100;
```

We can modify our bankist app to now show the correct dates and times but first for testing purposes we need to fake that we are always logged in using the code snippet in the event handlers section.

```
const now = new Date();  
const day = `${now.getDate()}`.padStart(2, 0);  
const month = `${now.getMonth() + 1}`.padStart(2, 0);  
const year = now.getFullYear();  
const hours = `${now.getHours()}`.padStart(2, 0);  
const mins = `${now.getMinutes()}`.padStart(2, 0);  
labelDate.textContent = `${day}/${month}/${year}, ${hours}:${mins}`;
```

The now variable is set to the current date and time.

From 'now' we can build the components of the date and time that we want then combine them together in a string to show date and time how we wish to see it rather than the ugly default settings. For day, month, hours and seconds we are using padding to add a leading zero if the value is only one digit long.

Note that this date and time is static and does not increment as time passes. For that we need a timer. See later lessons.

```
const account1 = {
  owner: 'Jonas Schmedtmann',
  movements: [200, 455.23, -306.5, 25000,
-642.21, -133.9, 79.97, 1300],
  interestRate: 1.2, // %
  pin: 1111,

  movementsDates: [
    '2019-11-18T21:31:17.178Z',
    '2019-12-23T07:42:02.383Z',
    '2020-01-28T09:15:04.904Z',
    '2020-04-01T10:17:24.185Z',
    '2020-05-08T14:11:59.604Z',
    '2020-05-27T17:01:17.194Z',
    '2020-07-11T23:36:17.929Z',
    '2020-07-12T10:51:36.790Z',
  ],
  currency: 'EUR',
  locale: 'pt-PT', // de-DE
};
```

The account objects have been modified to also include the property of movementsDates.

```

const displayMovements = function (acc, sort = false) {
  containerMovements.innerHTML = '';

  const movs = sort
    ? acc.movements.slice().sort((a, b) => a - b)
    : acc.movements;

  movs.forEach(function (mov, i) {
    const type = mov > 0 ? 'deposit' : 'withdrawal';

    const date = new Date(acc.movementsDates[i]);
    const day = `${date.getDate()}`.padStart(2, 0);
    const month = `${date.getMonth() + 1}`.padStart(2, 0);
    const year = date.getFullYear();
    const displayDate = `${day}/${month}/${year}`;

    const html = `
      <div class="movements__row">
        <div class="movements__type movements__type--${type}">${i + 1} ${type}</div>
        <div class="movements__date">${displayDate}</div>
        <div class="movements__value">${mov.toFixed(2)}€</div>
      </div>
    `;

    containerMovements.insertAdjacentHTML('afterbegin', html);
  });
};

```

The displayMovements function is modified to take the whole account object rather than just the movements array. This means the sort code has to be modified.

Acc.movementsDates is formatted into a Date and assigned to variable date.

A date string is made from the components of the date variable and assigned to variable display date.

Display date is added to the html.

The updateUI function is modified to call DisplayMovements with the acc rather than just movements.

```
const updateUI = function (acc) {  
  // Display movements  
  displayMovements(acc);  
  
  // Display balance  
  calcDisplayBalance(acc);  
  
  // Display summary  
  calcDisplaySummary(acc);  
};
```

```
btnLoan.addEventListener('click', function (e) {  
  e.preventDefault();  
  
  const amount = Math.floor(inputLoanAmount.value);  
  
  if (amount > 0 && currentAccount.movements.some(mov => mov >= amount * 0.1)) {  
    // Add movement  
    currentAccount.movements.push(amount);  
  
    // Add transfer Date  
    currentAccount.movementsDates.push(new Date().toISOString());  
  
    // Update UI  
    updateUI(currentAccount);  
  }  
  inputLoanAmount.value = '';  
});
```

The loan button has a new line added to push the current date into the movementsDates array in ISO string format.

```
btnTransfer.addEventListener('click', function (e) {  
  e.preventDefault();  
  const amount = +inputTransferAmount.value;  
  const receiverAcc = accounts.find(  
    acc => acc.username === inputTransferTo.value  
  );  
  inputTransferAmount.value = inputTransferTo.value = '';
```

```
  if (  
    amount > 0 &&  
    receiverAcc &&  
    currentAccount.balance >= amount &&  
    receiverAcc?.username !== currentAccount.username  
  ) {  
    // Doing the transfer  
    currentAccount.movements.push(-amount);  
    receiverAcc.movements.push(amount);  
  
    // Add transfer Date  
    currentAccount.movementsDates.push(new Date().toISOString());  
    receiverAcc.movementsDates.push(new Date().toISOString());
```

```
    // Update UI  
    updateUI(currentAccount);  
  }  
});
```

For the transfers button two new lines are added to add movementsDates values of now in an ISO string format to the array


```
let sorted = false;
btnSort.addEventListener('click', function (e) {
  e.preventDefault();
  displayMovements(currentAccount, !sorted);
  sorted = !sorted;
});
```

Now the sort button needs to be modified to sort the account rather than movements.

Operations with Dates

```
const future = new Date(2037, 10, 19, 15, 23);  
console.log(+future); 2142253380000
```

```
const calcDaysPassed = (date1, date2) =>  
  (date2 - date1) / (1000 * 60 * 60 * 24);
```

```
const days1 = calcDaysPassed(new Date(2037, 3, 14), new Date(2037, 3, 24));  
console.log(days1); 10
```

```
const days1 = calcDaysPassed(new Date(2037, 3, 14), new Date(2037, 3, 4));  
console.log(days1); -10
```

```
const calcDaysPassed = (date1, date2) =>  
  Math.abs(date2 - date1) / (1000 * 60 * 60 * 24);
```

```
const days1 = calcDaysPassed(new Date(2037, 3, 14), new Date(2037, 3, 4));  
console.log(days1); 10
```

```
const calcDaysPassed = (date1, date2) =>  
  Math.abs(date2 - date1) / (1000 * 60 * 60 * 24);
```

```
const days1 = calcDaysPassed(  
  new Date(2037, 3, 14, 10, 8), new Date(2037, 3, 4));  
console.log(days1); 10.422222222222222
```

Future variable is set to a date in the future. Console log shows a timestamp for this date which is a value of how many milliseconds from Jan 1st 1970 to the future date.

We can write a function to calculate the number of days between two dates, assuming date 2 is later than date 1.

We can use `Math.abs` to take the absolute value of `date1 - date2` which will always show a positive number of days even if `date2` is less than `date1`.

If one or both of the dates have hours and or minutes then we will get a result that contains a fraction. We can solve this by using a `Math.round` or `Math.floor` or `math.ceil` to round the value.

```

const displayMovements = function (acc, sort = false) {
  containerMovements.innerHTML = '';

  const movs = sort
    ? acc.movements.slice().sort((a, b) => a - b)
    : acc.movements;

  movs.forEach(function (mov, i) {
    const type = mov > 0 ? 'deposit' : 'withdrawal';

    const date = new Date(acc.movementsDates[i]);
    const displayDate = formatMovementDate(date);

```

We can modify our bankist app to display today, yesterday, x days ago up to a week or date in our movements section.

First we strip out the date code from the displayMovements function and put it in a separate function called formatMovementDate which takes date as a parameter.

```

const html = `
  <div class="movements__row">
    <div class="movements__type movements__type--${type}">${i + 1} ${type}</div>
    <div class="movements__date">${displayDate}</div>
    <div class="movements__value">${mov.toFixed(2)}€</div>
  </div>
`;

containerMovements.insertAdjacentHTML('afterbegin', html);
});
};

```

```

const formatMovementDate = function (date) {
  const calcDaysPassed = (date1, date2) =>
    Math.round(Math.abs(date2 - date1) / (1000 * 60 * 60 * 24));

  const daysPassed = calcDaysPassed(new Date(), date);
  console.log(daysPassed);

  if (daysPassed === 0) return 'Today';
  if (daysPassed === 1) return 'Yesterday';
  if (daysPassed <= 7) return `${daysPassed} days ago`;

  const day = `${date.getDate()}`.padStart(2, 0);
  const month = `${date.getMonth() + 1}`.padStart(2, 0);
  const year = date.getFullYear();
  return `${day}/${month}/${year}`;
};

```

Note that this works because JavaScript will read each if statement until true then it returns and does not look at the rest of the code. We do not need to end with an else or use complicated else if statements.

We use a function to calculate the difference between two dates (calcDaysPassed) rounding up the value.

We call the calcDaysPassed function passing in the current date and the date passed into the formatMovementDate function

Now we code the four returns based on the below logic:

- 1) If daysPassed is zero, return today.
- 2) If DaysPassed is 1, return yesterday,
- 3) If days passed is equal to or less than 7 return 'daysPassed' ago
- 4) If none of above format the date as dd/mm/yyyy

Welcome back, Jessica



user

PIN



Current balance

As of 13/05/2022, 17:05

11720.00€

8 WITHDRAWAL

YESTERDAY

-30.00€

7 DEPOSIT

TODAY

8500.00€

6 WITHDRAWAL

3 DAYS AGO

-1000.00€

5 WITHDRAWAL

5 DAYS AGO

-3210.00€

4 WITHDRAWAL

25/01/2020

-790.00€

Transfer money

Transfer to

Amount



Request loan

```
movementsDates: [  
  '2019-11-01T13:15:33.035Z',  
  '2019-11-30T09:48:16.867Z',  
  '2019-12-25T06:04:23.907Z',  
  '2020-01-25T14:18:46.235Z',  
  '2022-05-08T16:33:06.386Z',  
  '2022-05-10T14:43:26.374Z',  
  '2022-05-13T18:49:59.371Z',  
  '2022-05-12T12:01:20.894Z',  
],
```

For testing purposes the dates were modified in the movementDates object.

Internationalising Dates

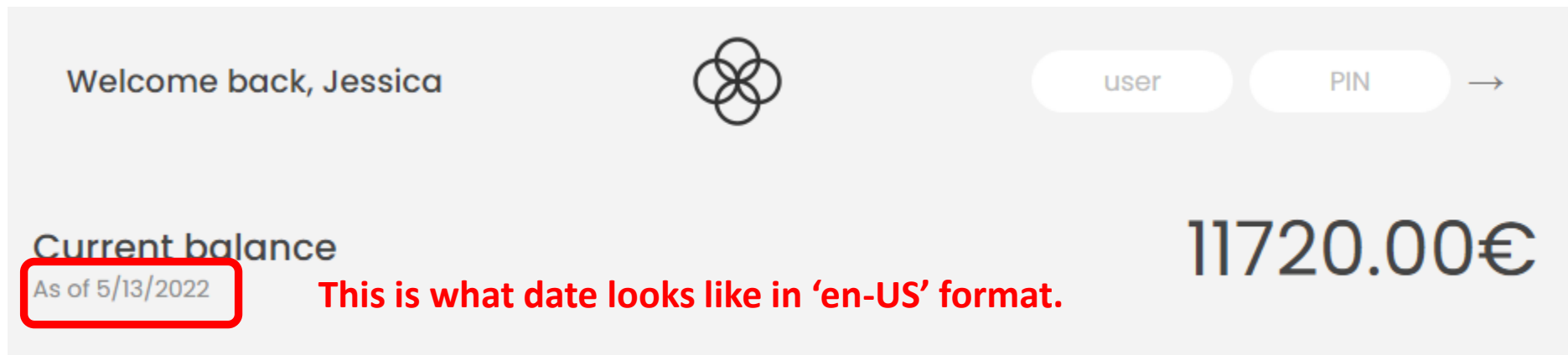
Internationalising date means we change it for the location of the user, i.e. in the USA date is mm/dd/yyyy, in Europe it is dd/mm/yyyy.

```
const now = new Date();
const day = `${now.getDate()}`.padStart(2, 0);
const month = `${now.getMonth() + 1}`.padStart(2, 0);
const year = now.getFullYear();
const hours = `${now.getHours()}`.padStart(2, 0);
const mins = `${now.getMinutes()}`.padStart(2, 0);
labelDate.textContent = `${day}/${month}/${year}, ${hours}:${mins}`;
```

This code snippet will format a date manually to European standard of dd/mm/yyyy but JavaScript has an automatic internationalisation API that can change dates, numbers and languages for us.

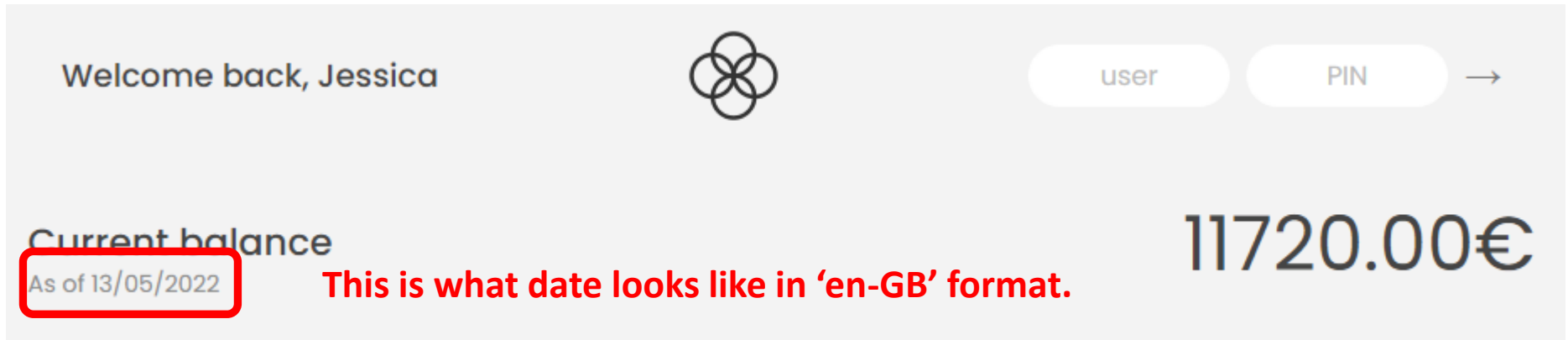
We can replace all of the above code with the less clunky Intl.DateTimeFormat function which takes a language as a parameter.

```
const now = new Date();
labelDate.textContent = new Intl.DateTimeFormat('en-US').format(now);
```

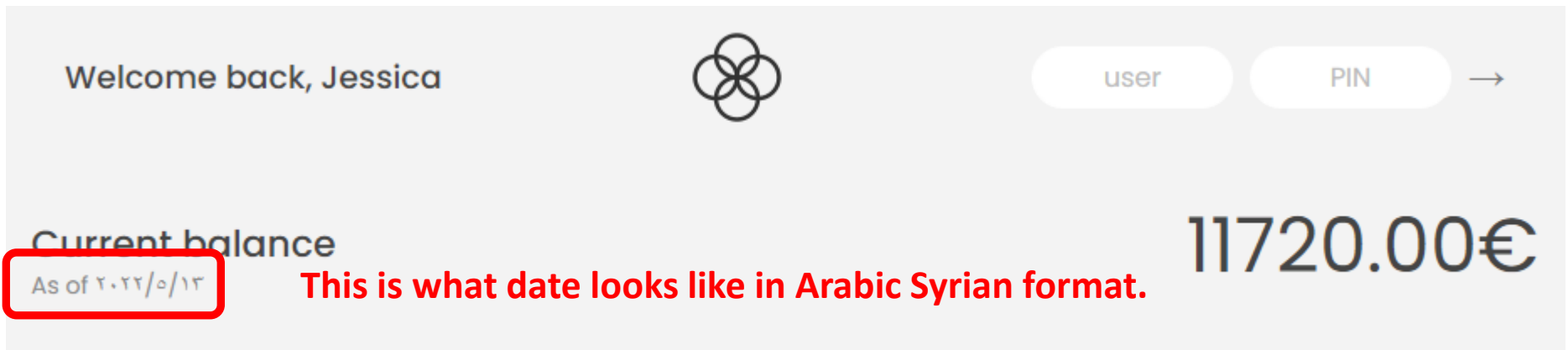


https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl

```
const now = new Date();  
labelDate.textContent = new Intl.DateTimeFormat('en-GB').format(now);
```



```
const now = new Date();  
labelDate.textContent = new Intl.DateTimeFormat('ar-SY').format(now);
```



For a complete list of language codes <http://www.lingoes.net/en/translator/langcode.html>

```
const options = {  
  hour: 'numeric',  
  minute: 'numeric',  
  day: 'numeric',  
  month: 'long',  
  year: 'numeric',  
  weekday: 'long',  
  // numeric, -2 digit, short, narrow  
};
```

We can create an external options object and specify all that we want in the date and time format and if we want long, short, numeric, -2 digit, narrow etc. month: long would be the full month i.e. November whereas month: short would be three letter month i.e. Nov.

The Intl.DateTimeFormat takes the language and an options object.

```
labelDate.textContent = new Intl.DateTimeFormat('es-ES', options).format(now);
```

Welcome back, Jessica



user

PIN



Current balance

As of viernes, 13 de mayo de 2022, 18:28

Date with options.

11720.00€


```
const options = {
  hour: 'numeric',
  minute: 'numeric',
  day: 'numeric',
  month: 'short',
  year: 'numeric',
  weekday: 'short', // numeric, -2 digit, short, narrow
};
```

Current balance

As of Fri, 13 May 2022, 18:36

```
labelDate.textContent = new Intl.DateTimeFormat('en-GB', options).format(now);
```

```
const now = new Date();
```

```
const options = {
  hour: 'numeric',
  minute: 'numeric',
  day: 'numeric',
  month: 'short',
  year: 'numeric',
  weekday: 'short',
};
```

Current balance

As of Fri, May 13, 2022, 6:40 PM

But currently we are manually defining the locale (en-GB) when it would be better to get it from the user's browser.

```
const locale = navigator.language;
console.log(locale);
```

en-US

```
labelDate.textContent = new Intl.DateTimeFormat(locale, options).format(now)
```

```

btnLogin.addEventListener('click', function (e) {
  // Prevent form from submitting
  e.preventDefault();

  currentAccount = accounts.find(
    acc => acc.username === inputLoginUsername.value);

  if (currentAccount?.pin === +inputLoginPin.value) {
    // Display UI and message
    labelWelcome.textContent = `Welcome back, ${currentAccount.owner.split(' ')[0]}`;
    containerApp.style.opacity = 100;

    // Create Current Date and time
    const now = new Date();

    const options = {
      hour: 'numeric',
      minute: 'numeric',
      day: 'numeric',
      month: 'short',
      year: 'numeric',
      weekday: 'short',
    };

    const locale = navigator.language;

    labelDate.textContent = new Intl.DateTimeFormat(locale, options).format(now);
  }
}

```

Now we cut the date time format functions and add them into the login function because in each account object we have a locale property which we can use to set the language.

```

const account2 = {
  ...
  currency: 'USD',
  locale: 'en-US',
};

```

```
// Create Current Date and time
const now = new Date();

const options = {
  hour: 'numeric',
  minute: 'numeric',
  day: 'numeric',
  month: 'short',
  year: 'numeric',
  weekday: 'short',
  // numeric, -2 digit, short, narrow
};

const locale = navigator.language;
```

```
labelDate.textContent = new Intl.DateTimeFormat(currentAccount.locale,
options).format(now);
```

```
const options = {
  hour: 'numeric',
  minute: 'numeric',
  day: 'numeric',
  month: 'numeric',
  year: 'numeric',
  // weekday: 'short',
  // numeric, -2 digit, short, narrow
};
```

Current balance

As of sexta, 13/05/2022, 19:08

We can call the locale property of the object using `currentAccount.locale`

But now we have a whole page in English with one element in the user language. We can make this a little bit more obvious by removing the weekday and setting the month to numeric.

```

const formatMovementDate = function (date, locale) {
  const calcDaysPassed = (date1, date2) =>
    Math.round(Math.abs(date2 - date1) / (1000 * 60 * 60 * 24));

  const daysPassed = calcDaysPassed(new Date(), date);

  if (daysPassed === 0) return 'Today';
  if (daysPassed === 1) return 'Yesterday';
  if (daysPassed <= 7) return `${daysPassed} days ago`;

  // const day = `${date.getDate()}`.padStart(2, 0);
  // const month = `${date.getMonth() + 1}`.padStart(2, 0);
  // const year = date.getFullYear();
  // return `${day}/${month}/${year}`;
  return new Intl.DateTimeFormat(locale).format(date);
};

```

We can also use the `Intl.DateTimeFormat` function to fix formatting of the movements dates by passing in an additional parameter of locale and substituting all the date manual formatting code.

```

const displayMovements = function (acc, sort = false) {
  ...

  const date = new Date(acc.movementsDates[i]);
  const displayDate = formatMovementDate(date, acc.locale);

```

Important to update the `formatMovementDate` function call to also pass the `acc.locale` parameter as well as the date!

Internationalising Numbers

```
const num = 3884764.23;
console.log('US:      ', new Intl.NumberFormat('en-US').format(num));
console.log('Germany:', new Intl.NumberFormat('de-DE').format(num));
console.log('Syria:   ', new Intl.NumberFormat('ar-SY').format(num));
```

US:	3,884,764.23
Germany:	3.884.764,23
Syria:	٣,٨٨٤,٧٦٤,٢٣

The JavaScript language API also supports numbers. For example in US English we separate long numbers with comas and use a dot for decimals. In German long numbers are separated by a dot with a coma used for decimals. We can also detect the language from the browser.

```
console.log('Browser:', new Intl.NumberFormat(navigator.language).format(num));
```

```
const num = 3884764.23;
const options = {
  style: 'unit',
  unit: 'mile-per-hour',
};
```

NumberFormat also supports options. We can use Unit to specify a standard unit.

```
console.log('US:      ', new Intl.NumberFormat('en-US', options).format(num));
console.log('Germany:', new Intl.NumberFormat('de-DE', options).format(num));
console.log('Syria:   ', new Intl.NumberFormat('ar-SY', options).format(num));
console.log('Browser:', new Intl.NumberFormat(navigator.language,
options).format(num));
```

US:	3,884,764.23 mph
Germany:	3.884.764,23 mi/h
Syria:	٣,٨٨٤,٧٦٤,٢٣ ميل/س
Browser:	3,884,764.23 mph

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl/NumberFormat

```
const num = 3884764.23;
const options = {
  style: 'unit',
  unit: 'celsius',
};
```

US:	3,884,764.23°C
Germany:	3.884.764,23 °C
Syria:	٣٨٨٤.٧٦٤,٢٣ °C
Browser:	3,884,764.23°C

NumberFormat also supports options. We can use Unit to specify a standard unit.

```
console.log('US: ', new Intl.NumberFormat('en-US', options).format(num));
console.log('Germany:', new Intl.NumberFormat('de-DE', options).format(num));
console.log('Syria: ', new Intl.NumberFormat('ar-SY', options).format(num));
console.log(
  'Browser:',
  new Intl.NumberFormat(navigator.language, options).format(num)
);
```

```
const num = 3884764.23;
const options = {
  style: 'percent',
  unit: 'celsius',
};
```

US:	388,476,423%
Germany:	388.476.423 %
Syria:	%٣٨٨,٤٧٦,٤٢٣
Browser:	388,476,423%

When style is set to percent the unit is completely ignored.

```
console.log('US: ', new Intl.NumberFormat('en-US', options).format(num));
console.log('Germany:', new Intl.NumberFormat('de-DE', options).format(num));
console.log('Syria: ', new Intl.NumberFormat('ar-SY', options).format(num));
console.log(
  'Browser:',
  new Intl.NumberFormat(navigator.language, options).format(num)
);
```

```
const num = 3884764.23;
const options = {
  style: 'currency',
  unit: 'celsius',
  currency: 'EUR',
};
```

US:	€3,884,764.23
Germany:	3.884.764,23 €
Syria:	٣,٨٨٤,٧٦٤,٢٣ €
Browser:	€3,884,764.23

When style is set to currency the unit is completely ignored, but we do have to define the currency. Note how in US format the euros sign is before whereas in Germany it is after. Currency is not determined by locale.

```
console.log('US: ', new Intl.NumberFormat('en-US', options).format(num));
console.log('Germany:', new Intl.NumberFormat('de-DE', options).format(num));
console.log('Syria: ', new Intl.NumberFormat('ar-SY', options).format(num));
console.log(
  'Browser:', new Intl.NumberFormat(navigator.language, options).format(num);
```

```
const num = 3884764.23;
const options = {
  style: 'currency',
  currency: 'EUR',
  useGrouping: false,
};
```

US:	€3884764.23
Germany:	3884764,23 €
Syria:	٣٨٨٤٧٦٤,٢٣ €
Browser:	€3884764.23

The separators for big numbers is called grouping and can be enabled or disabled. Default is enabled.

```
console.log('US: ', new Intl.NumberFormat('en-US', options).format(num));
console.log('Germany:', new Intl.NumberFormat('de-DE', options).format(num));
console.log('Syria: ', new Intl.NumberFormat('ar-SY', options).format(num));
console.log(
  'Browser:', new Intl.NumberFormat(navigator.language, options).format(num));
```

```

movs.forEach(function (mov, i) {
  const type = mov > 0 ? 'deposit' : 'withdrawal';

  const date = new Date(acc.movementsDates[i]);
  const displayDate = formatMovementDate(date, acc.locale);

```

```

const formattedMov = new Intl.NumberFormat(acc.locale, {
  style: 'currency',
  currency: 'USD',
}).format(mov);

```

We can use NumberFormat to format the movements in bankist project. Note that the options object is coded directly in the Intl.NumberFormat parameters.

```

const html = `
  <div class="movements__row">
    <div class="movements__type movements__type--${type}">${i + 1} ${type}</div>
    <div class="movements__date">${displayDate}</div>
    <div class="movements__value">${formattedMov}</div>
  </div>
`;

```

We change mov.fixed(-2) to formattedMov

```

containerMovements.insertAdjacentHTML('afterbegin', html);
});

```

But we also have a currency in the account object so we can call this rather than hard coding it.


```

movs.forEach(function (mov, i) {
  const type = mov > 0 ? 'deposit' : 'withdrawal';

  const date = new Date(acc.movementsDates[i]);
  const displayDate = formatMovementDate(date, acc.locale);

  const formattedMov = new Intl.NumberFormat(acc.locale, {
    style: 'currency',
    currency: acc.currency
  }).format(mov);

  const html = `
    <div class="movements__row">
      <div class="movements__type movements__type--${type}">${i + 1} ${type}</div>
      <div class="movements__date">${displayDate}</div>
      <div class="movements__value">${formattedMov}</div>
    </div>
  `;

  containerMovements.insertAdjacentHTML('afterbegin', html);
});

```

Now we are getting the currency from the `account.currency` property and formatting it according to `account.locale`. This means that someone with an account in Spain could have a currency of US dollars so it would be formatted as dots for separators, comas for decimal and US\$ as currency, i.e 4,000.29 US\$. Someone with currency as dollar and locale as US would be formatted \$4,000.29

Welcome back, Jessica



user

PIN



Current balance

As of 5/13/2022, 9:12 PM

11720.00€

8 WITHDRAWAL

YESTERDAY

-\$30.00

7 DEPOSIT

TODAY

\$8,500.00

6 WITHDRAWAL

3 DAYS AGO

-\$1,000.00

Transfer money

Transfer to

Amount



Request loan

IN 16900.00€ OUT 5180.00€ INTEREST 253.50€

↓
SORT

You will be logged out in 05:00

But there are other places where we need to implement Intl.NumberFormat to fix the currency.

```
const formatCur = function (value, locale, currency) {  
  return new Intl.NumberFormat(locale, {  
    style: 'currency',  
    currency: currency,  
  }).format(value);  
};
```

It would not be good practice to copy and paste code in multiple places to format the currency so a generic function that takes value, locale and currency would be much better. We can use this currency formatter multiple times in our code for the bankist project.

```
const formattedMov = new Intl.NumberFormat(acc.locale, {  
  style: 'currency',  
  currency: acc.currency,  
}).format(mov);
```

```
const formattedMov = formatCur(mov, acc.locale, acc.currency);
```

Now in the displayMovements function we can replace our previous account currency function and call our generic function.

```
const calcDisplayBalance = function (acc) {  
  acc.balance = acc.movements.reduce((acc, mov) => acc + mov, 0);  
  
  labelBalance.textContent = formatCur(acc.balance, acc.locale, acc.currency);  
};
```

We call our generic formatCur function to fix the displayAccountBalance function, passing in the value, locale and currency!

```
const calcDisplaySummary = function (acc) {  
  const incomes = acc.movements  
    .filter(mov => mov > 0)  
    .reduce((acc, mov) => acc + mov, 0);  
  labelSumIn.textContent = formatCur(incomes, acc.locale, acc.currency);  
  
  const out = acc.movements  
    .filter(mov => mov < 0)  
    .reduce((acc, mov) => acc + mov, 0);  
  labelSumOut.textContent = formatCur(Math.abs(out), acc.locale, acc.currency);  
  
  const interest = acc.movements  
    .filter(mov => mov > 0)  
    .map(deposit => (deposit * acc.interestRate) / 100)  
    .filter((int, i, arr) => {  
      return int >= 1;  
    })  
    .reduce((acc, int) => acc + int, 0);  
  labelSumInterest.textContent = formatCur(interest, acc.locale, acc.currency);  
};
```

We call our generic formatCur function to fix the calcDisplaySummary function, passing in the value, locale and currency!

Welcome back, Jonas



user

PIN



Now all the currency values are displayed in locale format with the correct currency based on the account object.

Current balance

As of 13/05/2022, 21:36

Note the number separators and the placement of the currency symbol

25 952,59 €

8 DEPOSIT

YESTERDAY

1300,00 €

7 DEPOSIT

2 DAYS AGO

79,97 €

6 WITHDRAWAL

TODAY

-133,90 €

5 WITHDRAWAL

5 DAYS AGO

-642,21 €

4 DEPOSIT

01/04/2020

25 000,00 €

3 WITHDRAWAL

28/01/2020

-306,50 €

Transfer money

Transfer to

Amount



Request loan

Amount



Close account

IN 27 035,20 € OUT 1082,61 € INTEREST 323,46 €

↓
SORT

You will be logged out in 05:00

Timers: set timeout and Set Interval

```
setTimeout(() => console.log('Here is your pizza 🍕'), 3000);
```

```
Here is your pizza 🍕
```

A simple timeout arrow function that displays 'here is your pizza' after 3 seconds (3000 ms). This is a call back where we wait 3000ms before calling the function.

```
setTimeout(() => console.log('Here is your pizza 🍕'), 3000);  
console.log('...waiting');
```

```
...waiting
```

```
Here is your pizza 🍕
```

We can see the timeout in action by logging '...waiting' till the timeout expires and the call back function is executed, in this case 'Here is your pizza'.

What is happening here is that the timeout timer starts counting then moves onto the next line of code (console.log ...waiting) when the timeout expires it returns to this line to execute the call back. This is called asynchronous JavaScript and is covered in future lessons.

Note that we are not calling the call back function ourselves, it is automatic in JavaScript so how do we pass in parameters to this call back function?

```
setTimeout(  
  (ing1, ing2) => console.log(`Here is your pizza with ${ing1} and ${ing2} 🍕`),  
  3000,  
  'olives',  
  'spinach'  
);  
console.log('pizza is cooking!');
```

We pass the parameters into the timeout function then use them in an object literal in the call back function.

```
pizza is cooking!
```

```
Here is your pizza with olives and spinach 🍕
```

`const ingredients = ['olives', 'spinach'];` Ingredients are now in a separate array.

```
const pizzaTimer = setTimeout(  
  (ing1, ing2) => console.log(`Here is your pizza with ${ing1} and ${ing2} 🍕`),  
  6000,  
  ...ingredients  
);
```

The spread operator is used to deconstruct ingredients array

```
console.log('pizza preperation in progress!');
```

```
if (ingredients.includes('spinach')) clearTimeout(pizzaTimer);
```

```
pizza preperation in progress!
```

If the ingredients array contains 'spinach' then the timeout timer is cleared. And the arrow function will never execute.

```
const ingredients = ['olives', 'tomatoes'];
```

```
const pizzaTimer = setTimeout(  
  (ing1, ing2) => console.log(`Here is your pizza with ${ing1} and ${ing2} 🍕`),  
  6000,  
  ...ingredients  
);
```

```
pizza preperation in progress!
```

```
Here is your pizza with olives and tomatoes 🍕
```

```
console.log('pizza preperation in progress!');
```

```
if (ingredients.includes('spinach')) clearTimeout(pizzaTimer);
```

However the timeout timer will now run down then display the message in the arrow function.

We can practice the use of a timeout timer in our bankist app. When a customer asks for a loan it is not usually approved straight away because the bank will do some checks and respond within a few minutes or hours or days.

```
btnLoan.addEventListener('click', function (e) {
  e.preventDefault();

  const amount = Math.floor(inputLoanAmount.value);

  if (amount > 0 && currentAccount.movements.some(mov => mov >= amount * 0.1)) {
    setTimeout(function () {
      // Add movement
      currentAccount.movements.push(amount);

      // Add transfer Date
      currentAccount.movementsDates.push(new Date().toISOString());

      // Update UI
      updateUI(currentAccount);
    }, 2500);
  }
  inputLoanAmount.value = '';
});
```

The add movement, add transfer date and updateUI function calls are wrapped in a timeout function with a 2.5 (2500ms) delay.

But what if we wanted to run a function over and over again according to a set time interval?

```
setInterval(function () {  
  const now = new Date();  
  console.log(now);  
}, 5000);
```

```
Fri May 13 2022 22:47:46 GMT+0200 (Central  
European Summer Time)  
Fri May 13 2022 22:47:51 GMT+0200 (Central  
European Summer Time)  
Fri May 13 2022 22:47:56 GMT+0200 (Central  
European Summer Time)  
Fri May 13 2022 22:48:02 GMT+0200 (Central  
European Summer Time)
```

The `setInterval` function will achieve this. Here we are logging the date and time to the console every five seconds.

Implementing a countdown Timer

We can use a countdown timer to log out the user of the bankist app automatically after 2 minutes.

```
// Clear input fields  
inputLoginUsername.value = inputLoginPin.value = '';  
inputLoginPin.blur();
```

```
// start logout timer  
startLogoutTimer();
```

In the login function we can call a logout timer function.

```
// Update UI  
updateUI(currentAccount);
```

```
const startLogoutTimer = function () {  
  // set time to 2 minute  
  let time = 120;
```

A time variable is set which is the countdown time.

```
  // call the timer every second We use the setInterval function to call the time every second.
```

```
  const timer = setInterval(function () {  
    const min = String(Math.trunc(time / 60)).padStart(2, 0);  
    const sec = String(time % 60).padStart(2, 0);
```

The time will show as 120 seconds rather than mm:ss so we use trunc, and padStart to build min and sec variables

```
  // in each call, print the remaining time to the UI  
  labelTimer.textContent = `${min}:${sec}`;
```

```
  // decrease time by 1 second  
  time--;
```

```
  // when 0 seconds, stop the timer and log out user
```

```
  if (time === 0) {  
    clearInterval(timer);  
    labelWelcome.textContent = 'Login to get started';  
    containerApp.style.opacity = 0;  
  }
```

The clearInterval function is used to stop the timer when time is zero. We do some DOM manipulation.

```
}, 1000);  
};
```

Note at the bottom that we want the time to update every 1 second (1000ms).

But this code has a bug in it. There is a delay of 1 second before the countdown timer starts ticking because the code within the startLogoutTimer only starts ticking once the 1000ms have passed. The call back is executed after the timer has expired!

```

const startLogoutTimer = function () {
  const tick = function () {
    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 2 minutes
  let time = 120;

  // Call the timer every second
  tick();
  const timer = setInterval(tick, 1000);
};

```

The solution to the bug is to wrap all the countdown actions in another function called tick. Then call the 'tick' from variable timer with a tick of 1000ms.

Note that the timer will log out at 1 second because the time was being decreased before the if time === 0 statement so we need to move the decrement to after this if statement.

However this has another bug in it. If we log in as one user then the timer starts but if we then subsequently login as another user then another timer is started while the other one is running. We need to stop any timers as part of the login process.

```
const startLogOutTimer = function () {  
  const tick = function () {  
    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 2 minutes  
let time = 120;  
// Call the timer every second  
tick();  
const timer = setInterval(tick, 1000);  
  
return timer;  
};
```

The simplest way is
to return the timer.

```
////////////////////////////////////  
//////////////// Event handlers  
let currentAccount, timer;
```

The timer variable also needs to be
in the parent scope with the
CurrentAccount.

```
// timer  
if (timer) clearInterval(timer);  
startLogOutTimer();
```

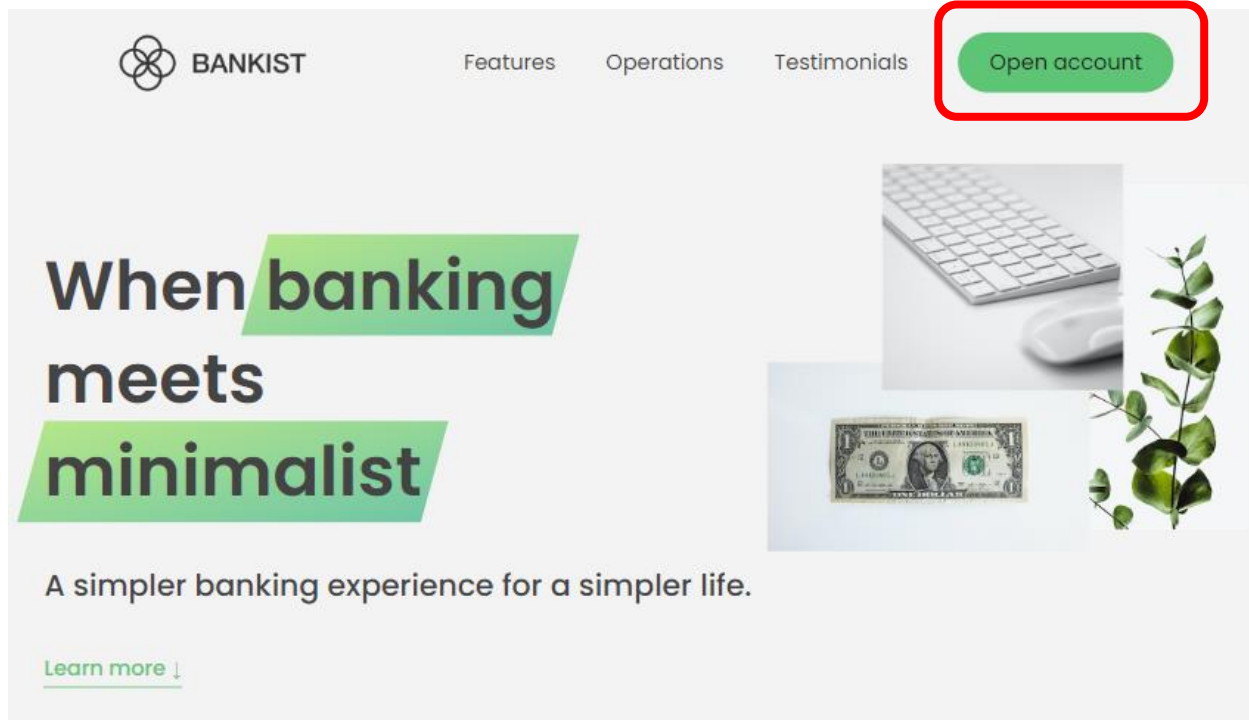
The login function needs to be
modified so that if there is a timer
running it is cleared before starting
a new timer.

The role of the timer is to countdown when there is no user activity so when a user performs a function such as a transfer or loan then the timer should be reset.

```
// Timer  
if (timer) clearInterval(timer);  
timer = startLogoutTimer();
```

This can simply be achieved by adding the code snippet to the transferBtn function and the loanBtn.

Advanced DOM and Events



We continue with our bvankist app this time using a lot of DOM manipulation to change the front end screen shown above.

We start with the open account buttons. There are two on the index page which both open the same modal window. See html code below.

We use a foreach loop to call back the openModal function.

```
<li class="nav__item">
  <a class="nav__link nav__link--btn btn--show-modal" href="#">Open account</a>
</li>
```

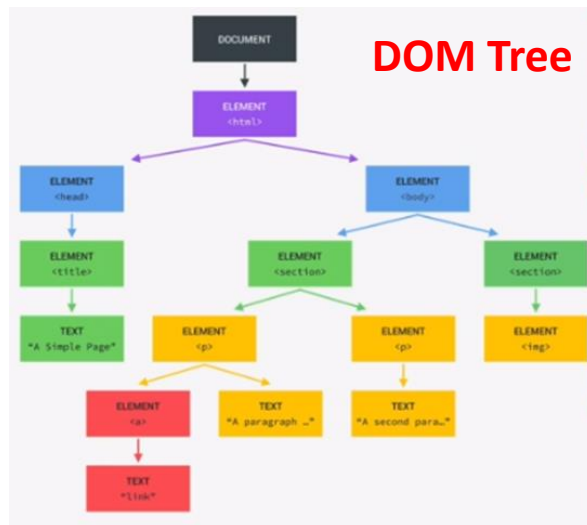
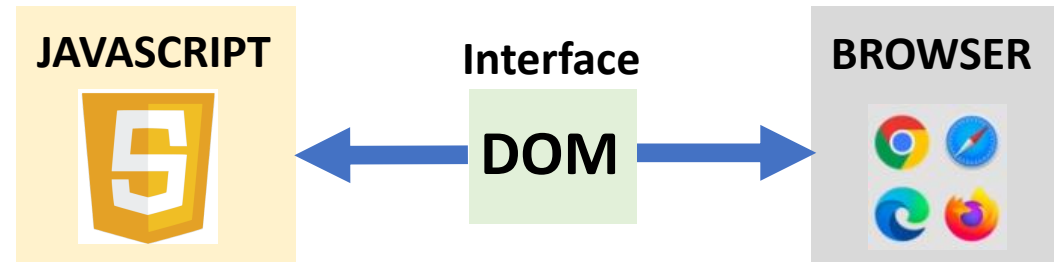
The default behaviour for html element href=# is to scroll to the top of the page. We do not want this but want the modal to open wherever the page is so we use preventDefault()

```
const openModal = function (e) {
  e.preventDefault();
  modal.classList.remove('hidden');
  overlay.classList.remove('hidden');
};
```

```
// Open account modal window buttons
btnsOpenModal.forEach(btn => btn.addEventListener('click', openModal));
```

How the DOM really works

The DOM is simply the interface between our JavaScript and the browser. more specifically HTML documents that are rendered by the browser.



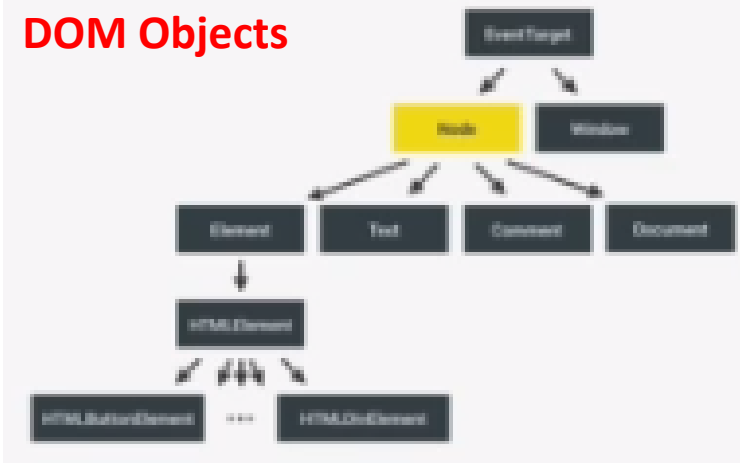
DOM Tree

- 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 Objects



- DOM is a very complex API that contains lots of methods and properties to interact with the DOM tree.

```
.querySelector() / .addEventListener() / .createElement() /  
.innerHTML / .textContent / .children / etc ...
```


How the DOM API is organised behind the scenes

Each node is represented in JavaScript by an object, with each object getting access to certain properties.

Every single node in the DOM tree is of the type Node.

Whenever there is text in an element it will be of type text.

Each element type has a child type, one for each type of HTML element because each HTML element type has different properties and attributes.

Document type is just another kind of node and has methods. Note how .querySelector is in the element type and document type.

Window is the global object and has lots of methods and properties, many unrelated to DOM.

Event target type methods can be called on window and all other node types because of inheritance.

INHERITANCE: Each child type will get access to the properties and methods of its parent type. And the node.

```
.innerHTML  
.classList  
.children  
.parentElement  
.append()  
.remove()  
.insertAdjacentHTML()  
.querySelector() ←  
.closest()  
.matches()  
.scrollIntoView()  
.SetAttributr()
```

```
.textContent  
.childNodes  
.parentNode  
.cloneNode()
```

EventTarget

```
.addEventListener()  
.removeEventListener()
```

node

Window

<p>Paragraph</p>

Element

Text

Comment

Document

Child type

HTMLElement

<p>Paragraph</p>

<!-- Comment -->

```
.querySelector() ←  
.createElement()  
.getElementById()
```

HTMLbuttonElement

...

HTMLdivElement

Selecting Elements

```
console.log(document.documentElement);
```

```
console.log(document.head);
```

```
console.log(document.body);
```

```
<html lang="en">  
  <head>...</head>  
  <body>...</body>  
</html>
```

We can select the whole page in JavaScript, for example. To apply a common style. The whole html page is called `documentElement`.

```
▶ <head>...</head>
```

Or Just the header with `.head`

```
▶ <body>...</body>
```

Or the body with `.body`

```
document.querySelector('.header');
```

We can select a specific element using `querySelector` where it will select the first element that meets the criteria reading the html document from top to bottom.

```
const allSections = document.querySelectorAll('.section');  
console.log(allSections);
```

To select multiple elements we can use `querySelectorAll` which will return a node list.

```
NodeList(4) [  
  section#section--1.section,  
  ▶ section#section--2.section,  
  section#section--3.section,  
  section.section.section--sign-up]
```

```
document.getElementById('section--1');
```

We can select a specific element by it's ID.

```
const allButtons = document.getElementsByTagName('button');  
console.log(allButtons);
```

We can select an element by its HTML tag i.e. button using `getElementsByTagName`

```
HTMLCollection(9) [button.btn--text.btn--scroll-to,  
button.btn.operations__tab.operations__tab--1.operations__tab--active,  
button.btn.operations__tab.operations__tab--2,  
▶ button.btn.operations__tab.operations__tab--3,  
button.slider__btn.slider__btn--left,  
button.slider__btn.slider__btn--right, button.btn.btn--show-modal,  
button.btn--close-modal, button.btn]
```

This will return a html collection, which is different to a node list because it is a **live list**. The **live list will automatically update if the DOM changes**, for example a button is deleted.

```
console.log(document.getElementsByClassName('btn'));
```

We can also select elements by class name. This will also return a html collection - a **live list**.

```
HTMLCollection(5) [  
button.btn.operations__tab.operations__tab--1.operations__tab--active,  
▶ button.btn.operations__tab.operations__tab--2,  
button.btn.operations__tab.operations__tab--3,  
button.btn.btn--show-modal, button.btn]
```

Creating and Inserting Elements

```
const html = `  <div class="movements__type movements__type--${type}">${i + 1} ${type}</div>  
  <div class="movements__date">${displayDate}</div>  
  <div class="movements__value">${formattedMov}</div>  
</div>  
`;  
  
containerMovements.insertAdjacentHTML('afterbegin', html);
```

insertAdjacentHTML
can be used to insert
text where the
parameters are **position**
and **text**.

<https://developer.mozilla.org/en-US/docs/Web/API/Element/insertAdjacentHTML>

Position:

‘beforebegin’ - Before the element. Only valid if the element is in the DOM tree and has a parent element.

‘afterbegin’ - Just inside the element, before its first child.

‘beforeend’ - Just inside the element, after its last child.

‘afterend’ - After the element. Only valid if the element is in the DOM tree and has a parent element.

Text:

The string to be parsed as HTML or XML and inserted into the tree.

```
const message = document.createElement('div')
```

We can create an empty div element as a variable. Note that it is not on the DOM yet (in the HTML document). It is only a DOM element that we can use by inserting onto the page with more code.

```
const message = document.createElement('div').message.classList.add('cookie-message');
```

We can add class to a newly created DOM element in the same line of code by using `classList.add` method.

```
const message = document.createElement('div');  
message.classList.add('cookie-message');  
message.textContent =  
  'We use cookies for improved functionality and analytics.';
```

We chain in the `textContent` method to add text.

```
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>');
```

Or use `innerHTML` method to add a block of html including html elements such as button..

```
const header = document.querySelector('.header');
```

```
const message = document.createElement('div');  
message.classList.add('cookie-message');  
message.innerHTML =
```

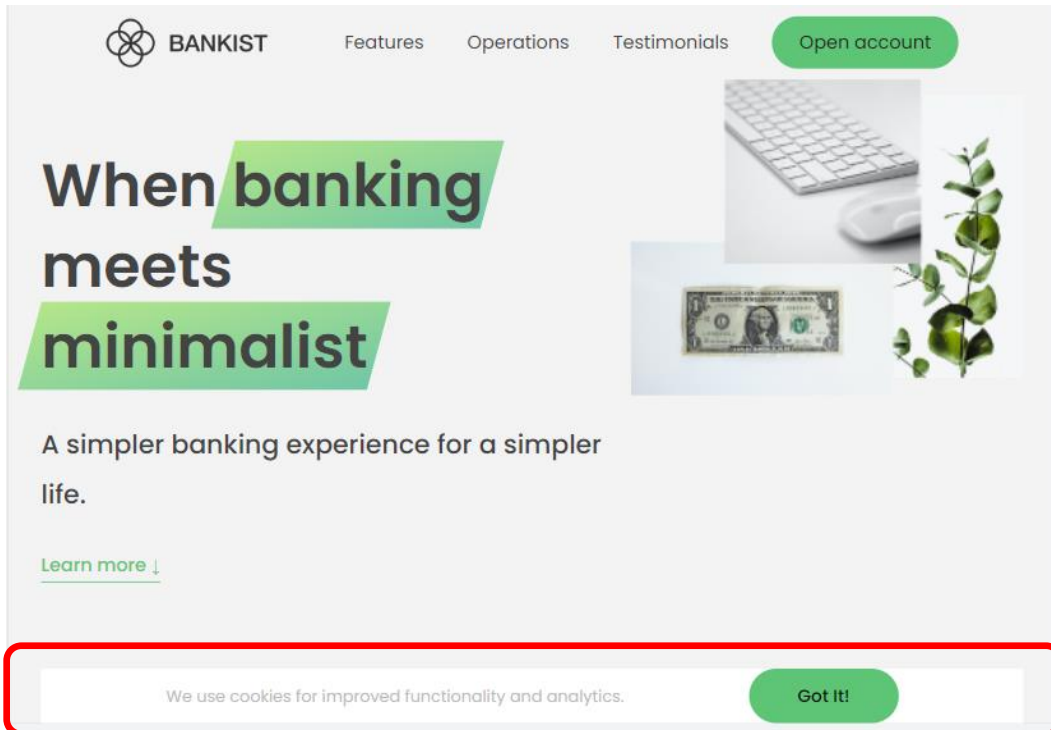
```
'We use cookies for improved functionality and analytics. <button class="btn btn--close--cookie">Got It!</button>';
```

```
header.prepend(message);  
header.append(message);
```

The `Element.prepend()` method is used to insert a Node object before the first child of the header Element. `Append()` will add it as the last child.

1) Header variable is defined and header element is selected with `querySelector`.

2) A div is created with `createElement` and `classList.add` and `innerHTML` method is used to fill the div element

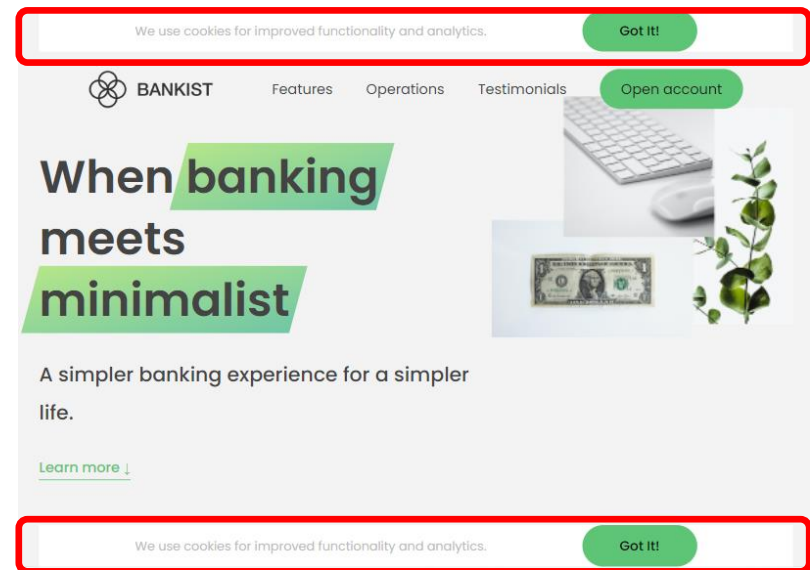


Note that there is only one cookie-message element on the page. That is because the DOM element is unique and cannot exist in two places at the same time so first the cookie-message was added as the first child of the header element then it was immediately removed and added as the last child of the header element.

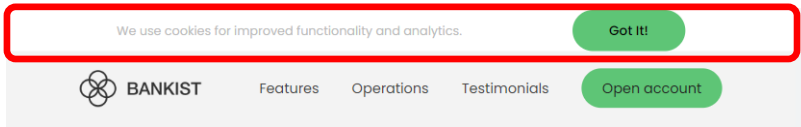
```
header.prepend(message);
header.append(message.cloneNode(true));
```

If we want the same element in two places then we would have to clone it using the cloneNode method with the options of true set (to clone all child elements also)

For this exercise will will append it only (add it as the last child element of header.)

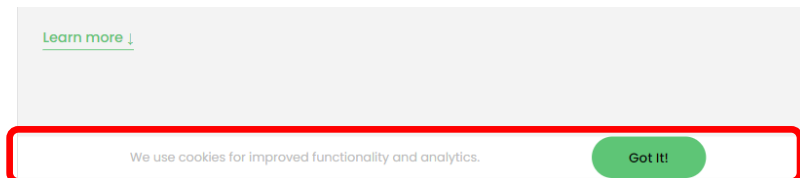


```
header.before(message);
```



The .before method will add it before the header element as a sibling.

```
header.after(message);
```



The .after method will add it after the header element as a sibling.

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    ... <div class="cookie-message">...</div> flex == $0
    <header class="header">...</header> flex
    <section class="section" id="section--1">...</section>
```

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <header class="header">...</header> flex
    ... <div class="cookie-message">...</div> flex == $0
    <section class="section" id="section--1">...</section>
```

```
const header = document.querySelector('.header');

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.prepend(message);
```

```
// Delete element
```

```
document
    .querySelector('.btn--close--cookie')
    .addEventListener('click', function () {
        message.remove();
    });
```

```
message.parentElement.removeChild(message);
```

We can remove an element from the DOM using the remove method.

In this example we have attached an eventListener to the button within the innerHTML of the cookie-message element that removes cookie-message when clicked.

The remove method is a recent addition to JavaScript. Before we would move up one layer to the parent element then remove the child, naming the child again! A lot more cumbersome than just remove.

This is called DOM traversing and is covered in a future lecture.

Styles

```
message.style.backgroundColor = '#37383d';  
message.style.width = '120%';
```

```
console.log(message.style.height);  
console.log(message.style.backgroundColor); #37383d
```

```
console.log(getComputedStyle(message));
```

This returns a long and complicated object with all of the style properties. What we do in practice is select the property we want, for example, color.

```
console.log(getComputedStyle(message).color); rgb(187, 187, 187)
```

These are the computed style properties. Even if they are not specified in the CSS file the browser has to compute them to render them on the screen.

```
message.style.height =  
    Number.parseFloat(getComputedStyle(message).height, 10) + 30 + 'px';
```

We can add 30px to the computed height but when we do + 40 + px gets converted into a string so we have to use Number.parseFloat() so that the numbers are treated as numbers by JavaScript.

We can set the a style property of an element in JavaScript. This is called inline style. If the style property is two words in css with a hyphen then we use camel case. i.e. background-color becomes backgroundColor.

We cannot read any style properties that are set in the CSS, only those that are set in JavaScript as inline style.

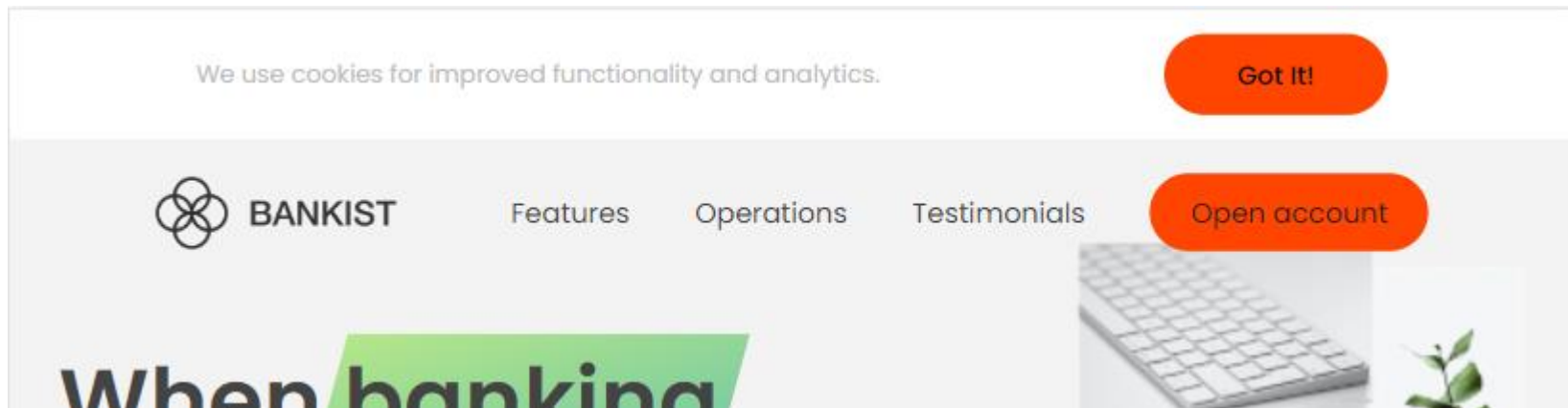
```
CSSStyleDeclaration {0: 'accent-color', 1: 'align-content',  
2: 'align-items', 3: 'align-self', 4: 'alignment-baseline',  
5: 'animation-delay', 6: 'animation-direction', 7: 'animation-duration', 8: 'animation-fill-mode', 9: 'animation-iteration-count', 10: 'animation-name', 11: 'animation-play-state', 12: 'animation-timing-function', 13: 'app-region', 14: 'appearance', 15: 'backdrop-filter', 16: 'backface-visibility', 17: 'background-attachment', 18: 'background-blend-mode', 19:
```

```
:root {  
  --color-primary: #5ec576;  
  --color-secondary: #ffcb03;  
  --color-tertiary: #ff585f;  
  --color-primary-darker: #4bbb7d;  
  --color-secondary-darker: #ffbb00;  
  --color-tertiary-darker: #fd424b;  
  --color-primary-opacity: #5ec5763a;  
  --color-secondary-opacity: #ffcd0331;  
  --color-tertiary-opacity: #ff58602d;  
  --gradient-primary: linear-gradient(to top left, #39b385, #9be15d);  
  --gradient-secondary: linear-gradient(to top left, #ffb003, #ffcb03);  
}
```

In the css file we have something called CSS custom properties, usually called css variables. The point of this is we can change a value of colour in many places by referencing the variable.

```
document.documentElement.style.setProperty('--color-primary', 'orangered');
```

We can use `documentElement.style` to set the property of and of the defined color variables in the css file to instantly change color on multiple elements simultaneously.



Attributes

In the html file we have an img element that has all the normal properties plus a designer property which is non standard and does not actually exist.

```

```

Bankist logo
http://127.0.0.1:5500/13-Advanced-DOM-and-Events/img/logo.png
undefined
nav__logo

Note how the designer attribute is undefined because it is non standard. Note also that the class attribute in JavaScript is className.

```
console.log(logo.getAttribute('designer')); Jonas
```

We can use `getAttribute()` to read non standard attributes.

```
logo.alt = 'Beutiful minamalist logo';
```

Just as we can read the attributes, we can also set them from JavaScript.

```
logo.setAttribute('company', 'Bankist');
```

The `setAttribute` method can be used to set or add new attributes to an element.

```
<nav class="nav">
  
  <ul class="nav__links">...</ul>
```

```
<nav class="nav">
  
  <ul class="nav__links">...</ul>
```

<pre></pre>	<pre>console.log(logo.src); console.log(logo.getAttribute('src')); http://127.0.0.1:5500/13-Advanced-DOM-and-Events/img/logo.png img/logo.png</pre>

In the html file the img has a relative path/filename. Relative being its location from the current index file, where as the browser has it's absolute path. I.e. the full path including the IP. GetAttribute will show the absolute path whereas element.src will show the relative path.

<pre><li class="nav__item"> Open account </pre> <pre>const link = document.querySelector('.nav__link--btn'); console.log(link.href); console.log(link.getAttribute('href'));</pre>	<pre>http://127.0.0.1:5500/13-Advanced-DOM-and-Events/# #</pre>

With href attribute getAttribute will show relative path and element.href will show absolute path.

```

```

```
const logo = document.querySelector('.nav__logo');
console.log(logo.dataset.versionNumber);
```

3.0

There is a special attribute called data where we can assign a data value to a html element. The name must start with 'data-' then whatever we want to call it. We can then get that attribute in Javascript with the element.dataset method. Note that if the name of the data attribute in html has a hyphen (version-number) then we have to use camel case in Javascript (versionNumber).

```
const logo = document.querySelector('.nav__logo');
logo.classList.add('c', 'd');
logo.classList.remove('c');
logo.classList.toggle('c');
logo.classList.contains('c')
```

Do not use logo.className = 'something' because this will overwrite all the existing classes on that element. Use instead one of the four methods above to manipulate classes on elements.

The four methods above allow us to add, remove or toggle multiple classes on an element, simply separated by a coma. classList.contains can be used to check if an element has a class.

Smooth Scrolling

Smooth scrolling is the feature where we click on a menu item or button on a page and we are automatically scrolled to the section for that button.

```
const btnScrollTo = document.querySelector('.btn--scroll-to');
const section1 = document.querySelector('#section--1');
```

```
btnScrollTo.addEventListener('click', function (e) {
  const s1coords = section1.getBoundingClientRect();
  console.log(s1coords);
```

`getBoundingClientRect()` will return values of the DOMRect object which is the smallest rectangle that contains the entire element, including its padding and border-width.

```
console.log(e.target.getBoundingClientRect());
```

```
console.log('current scroll (X/Y)', window.pageXOffset, pageYOffset);
```

```
console.log(
  'Height/Width viewport',
  document.documentElement.clientHeight,
  document.documentElement.clientWidth
);
});
```

The `e.target`, event target, the scroll to button has y of 320 and section 1 has y of 413. The difference is the amount of vertical scroll between button and section 1

We need to define the button and the section as variables and add an event listener to the button.

We can see the x and y coordinates of the section 1 element and the button.

```
DOMRect {x: 0, y: 413, width: 966, height: 1567.1875,
  top: 413, ...}
```

```
DOMRect {x: 30, y: 320.46875, width: 110, height: 29,
  top: 320.46875, ...}
```

```
current scroll (X/Y) 0 257
```

```
Height/Width viewport 670 966
```

```
// Scrolling
window.scrollTo(
  s1coords.left + window.pageXOffset,
  s1coords.top + window.pageYOffset
);
});
```

To perform the page scrolling we can use the `scrollTo` method where we calculate the scroll to position based on the coordinates of section 1 plus the the x and y page offset.

I.e. the current position plus the current scroll

```
// Scrolling
window.scrollTo({
  left: s1coords.left + window.pageXOffset,
  top: s1coords.top + window.pageYOffset,
  behaviour: 'smooth',
});
});
```

But the scrolling is clunky. We can improve it by implementing smooth scrolling. This is where the browser calculates the speed at which it should scroll rather than just jumping to that point in the document. To do this we pass in an object to the `scrollTo` method.

```
section1.scrollToView({ behaviour: 'smooth' });
```

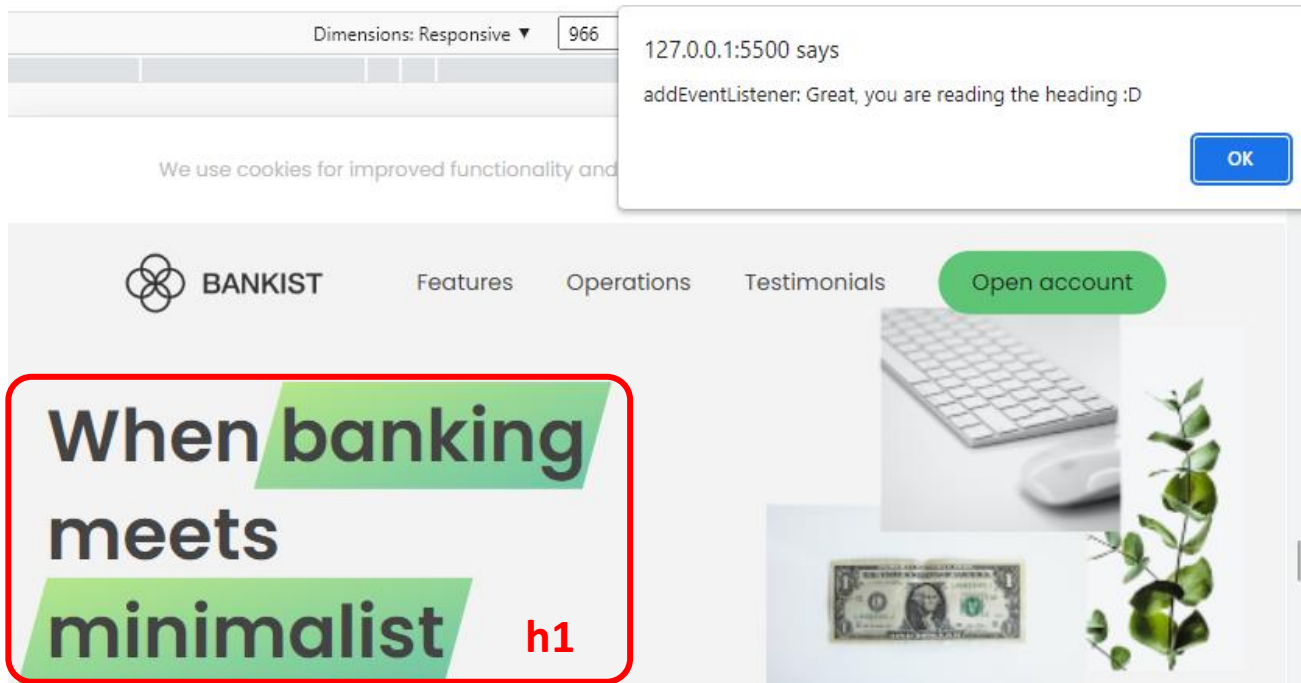
But there is a more modern method to passing in an object into `scrollTo`. We can use the modern `scrollIntoView` method which takes an object containing only the behaviour.

Types of Events and Event Handlers

An event is basically a signal that is generated by a certain DOM node. A signal means that something has happened, for example a click or mouse move or the user triggering full screen mode. We can then listen for these events in our code by adding event listeners to handle them. But these events will always happen regardless of if we are listening for them.

```
const h1 = document.querySelector('h1');
```

```
h1.addEventListener('mouseenter', function (e) {  
    alert('addEventListener: Great, you are reading the heading :D');  
});
```



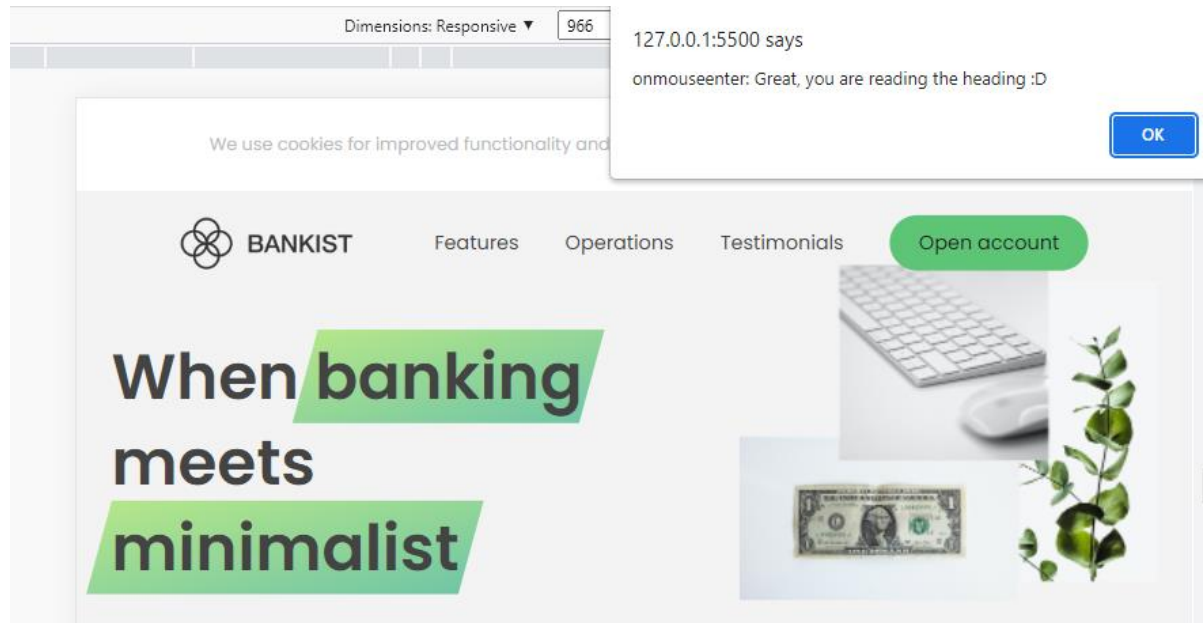
Mouseenter events will be fired when the mouse enters the element


```
const h1 = document.querySelector('h1');
```

```
h1.addEventListener('mouseenter', function (e) {  
    alert('addEventListener: Great, you are reading the heading :D');  
});
```

```
h1.onmouseenter = function (e) {  
    alert('onmouseenter: Great, you are reading the heading :D');  
};
```

The onmouseenter property can be added directly to the element rather than using an event listener. For each event in the table there is an event property i.e. onclick, but this is a bit old fashioned.



Event listeners are better because they allow multiple events to be added to an element whereas adding an event handler as a property would only allow one event handler, (the second event handler would overwrite the first).

```
const alertH1 = function (e) {  
  alert('addEventListener: Great, you are reading the heading :D');  
};
```

We can add a function as the action to be performed on an event listener.

```
h1.addEventListener('mouseenter', alertH1);
```

```
const alertH1 = function (e) {  
  alert('addEventListener: Great, you are reading the heading :D');  
  h1.removeEventListener('mouseenter', alertH1);  
};
```

Within the function we can then remove the event listener.

```
h1.addEventListener('mouseenter', alertH1);
```

```
const alertH1 = function (e) {  
  alert('addEventListener: Great, you are reading the heading :D');  
};
```

A timeout can also be used to remove an event listener.

```
h1.addEventListener('mouseenter', alertH1);
```

```
setTimeout(() => h1.removeEventListener('mouseenter', alertH1), 3000);
```

```
<h1 onclick="alert('HTML alert')">
```

A very old way is to add the event directly into the HTML code. This is obsolete but may still show up in old code.

Event Propagation: Capture and bubbling

JavaScript events have a very important property: they have a capturing phase and a bubbling phase.

Here we can see all the parent elements of the anchor tag.



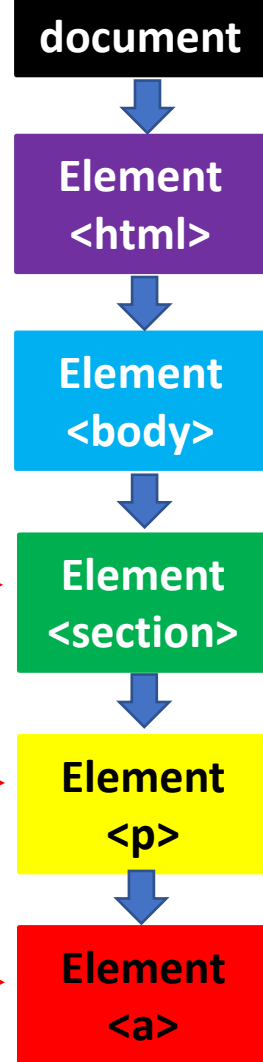
Events are handled right at the target with event listeners. Event listeners wait for the event to occur then execute the call back function.

1 CAPTURING PHASE

As the click event is actually generated at the root of the document at the very top of the DOM tree then travels down the tree to the target element passing through every parent element till it reaches the target element.

1 TARGET PHASE

Click



BUBBLING PHASE 3

After the event has happened the event travels all the way up the document to the document root at the top.

As an event passes down and up the tree it passes through all the parent elements and it is as if the event happen in each of the parent elements. If we attach the same event listener to another element then it will also be handled there.

Bubbling is when an event passes through all parent elements and is handled there as well as the target phase.

Events by default can only be handled in the target phase and bubbling phase.

However we can set up event listeners so they happen in the capturing phase.

Note that some types of events do not have a capturing and bubbling phase but most do.

Capturing and bubbling is events propagating from one place to another.

Event Propagation in practice

```
<header class="header">
  <nav class="nav">
    
    <ul class="nav__links">
      <li class="nav__item">
        <a class="nav__link" href="#">Features</a>
      </li>
```

```
const randomInt = (min, max) =>
  Math.floor(Math.random() * (max - min + 1) + min);
const randomColor = () =>
  `rgb(${randomInt(0, 255)}, ${randomInt(0, 255)}, ${randomInt(0, 255)})`;
console.log(randomColor(0, 255));
```

We have a html block with DOM tree of Document(root) -> html -> head -> header -> nav -> ul -> li -> a. The JavaScript code is to generate a random color.

```
document.querySelector('.nav__link').addEventListener('click', function (e) {  
  this.style.backgroundColor = randomColor();  
});
```



BANKIST

Features

Operations

Testimonials

Open account

Now when a 'nav—link' element is clicked a random color is added to the element.

```
document.querySelector('.nav__link').addEventListener('click', function (e) {  
  this.style.backgroundColor = randomColor();  
});
```

```
document.querySelector('.nav__links').addEventListener('click', function (e) {  
  this.style.backgroundColor = randomColor();  
});
```

Now the parent element of navLinks also has an event handler to add a random background color.



BANKIST

Features

Operations

Testimonials

Open account

When the features 'nav—link' is clicked on its parent element of 'nav—link' also gets its own background color because the click happens at the document root then it bubbles up to the parent elements and 'nav—link' parent element also has an event handler for a click. If the 'nav-links' element is clicked then the child element of 'nav-link' does not change because bubbling only happens up the DOM tree.



BANKIST

Features

Operations

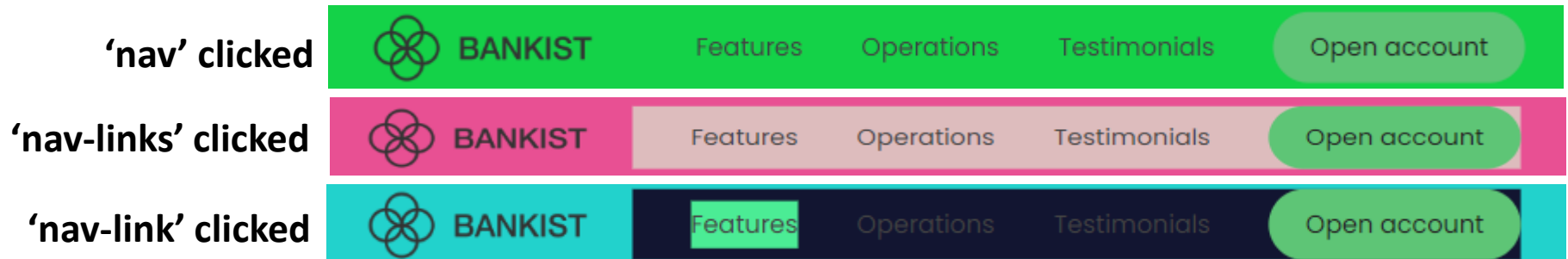
Testimonials

Open account

```
document.querySelector('.nav__link').addEventListener('click', function (e) {  
  this.style.backgroundColor = randomColor();  
});
```

```
document.querySelector('.nav__links').addEventListener('click', function (e) {  
  this.style.backgroundColor = randomColor();  
});
```

```
document.querySelector('.nav').addEventListener('click', function (e) {  
  this.style.backgroundColor = randomColor();  
});
```



The event can now bubble in three layers of parent elements. Depending where is clicked will determine where the click activates the event listener.

```
document.querySelector('.nav__link').addEventListener('click', function (e) {  
  this.style.backgroundColor = randomColor();  
  console.log('LINK', e.target);  
});
```

```
document.querySelector('.nav__links').addEventListener('click', function (e) {  
  this.style.backgroundColor = randomColor();  
  console.log('CONTAINER', e.target);  
});
```

```
document.querySelector('.nav').addEventListener('click', function (e) {  
  this.style.backgroundColor = randomColor();  
  console.log('NAV', e.target);  
});
```

The target is where the event originated, i.e. where the click happened and is not the event handler.

Clicking on a 'nav-link' will show the event originating on the link but it also bubbles up in all three handlers. Notice how the target is always the same 'nav-link'.

LINK	script.js:81
<pre>Features</pre>	
CONTAINER	script.js:86
<pre>Features</pre>	
NAV	script.js:91
<pre>Features</pre>	

```
document.querySelector('.nav__link').addEventListener('click', function (e) {
  this.style.backgroundColor = randomColor();
  console.log('LINK', e.target, e.currentTarget);
});
```

```
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);
});
```

We can look at the target where the event occurred and the current target where the event is happening.

The current target is exactly the same as the this keyword (e.currentTarget === this) where the event was initiated.

```
LINK script.js:81
<a class="nav__link" href="#" style="background-color:
rgb(201, 216, 15);">Features</a>
<a class="nav__link" href="#" style="background-color:
rgb(201, 216, 15);">Features</a>

CONTAINER script.js:86
<a class="nav__link" href="#" style="background-color:
rgb(201, 216, 15);">Features</a>
▶ <ul class="nav__links" style="background-color: rgb(24,
140, 196);">...</ul> flex

NAV script.js:91
<a class="nav__link" href="#" style="background-color:
rgb(201, 216, 15);">Features</a>
▶ <nav class="nav" style="background-color: rgb(156, 230,
191);">...</nav> flex
```



```
document.querySelector('.nav__link').addEventListener('click', function (e) {  
  this.style.backgroundColor = randomColor();  
  console.log('LINK', e.target, e.currentTarget);  
  console.log(e.currentTarget === this);
```

```
// Stop Propagation  
e.stopPropagation();
```

We can stop propagation from bubbling up

```
});
```

```
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);  
});
```



BANKIST

Features

Operations

Testimonials

Open account

In general it is not good to stop propagation of events. However it may solve issues with very complex applications with many events and handlers.

LINK

script.js:81

```
<a class="nav__link" href="#" style="background-color:  
rgb(197, 169, 75);">Features</a>
```

```
<a class="nav__link" href="#" style="background-color:  
rgb(197, 169, 75);">Features</a>
```

true

script.js:82

```
document.querySelector('.nav__link').addEventListener('click', function (e) {
  this.style.backgroundColor = randomColor();
  console.log('LINK', e.target, e.currentTarget);
  // console.log(e.currentTarget === this);

  // Stop Propagation
  // e.stopPropagation();
});
```

```
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);
  },
  true
);
```

AddEventListener can take a third parameter, set to true, which overrides the default behaviour of only handling events in the target and bubbling phase so now the event is handled in the capture phase. Note how the NAV is handled first now.

This is because the nav is listening for the event as it travels down the DOM where as the other two are listening as it travels up the DOM.

```
NAV script.js:97
<a class="nav__link" href="#" style="background-color:
rgb(6, 206, 125);">Features</a>
▶ <nav class="nav" style="background-color: rgb(237, 182,
6);">...</nav> flex

LINK script.js:81
<a class="nav__link" href="#" style="background-color:
rgb(6, 206, 125);">Features</a>
<a class="nav__link" href="#" style="background-color:
rgb(6, 206, 125);">Features</a>

CONTAINER script.js:90
<a class="nav__link" href="#" style="background-color:
rgb(6, 206, 125);">Features</a>
▶ <ul class="nav__links" style="background-color: rgb(20,
57, 152);">...</ul> flex
```

Event Delegation: Implementing Page Navigation

```
document.querySelectorAll('.nav__link').forEach(function (el) {  
  el.addEventListener('click', function (e) {  
    e.preventDefault();  
    const id = this.getAttribute('href');  
    // console.log(id);  
    document.querySelector(id).scrollIntoView({ behaviour: 'smooth' });  
  });  
});
```

We are using `querySelectorAll` to select all the 'nav__links' and for each of them adding an `addEventListener` on the click.

We are preventing the default behaviour of html jumping to that link so we can add smooth scrolling.

As the href link for each section in the html is the same as the section id we can `getAttribute href` and store it as an id variable.

The id can have a `scrollIntoView` method applied with smooth scrolling.

This code will work fine. However it adds an event listener to each of the navigation elements. This is not very efficient nor scalable code (if we had tens of elements or more!) so we need to use Event Delegation.

Event delegation is where we put a common event handler on the parent of the elements and we can catch the event as it bubbles up from the child element.

```
document.querySelector('.nav__links').addEventListener('click', function (e) {  
  e.preventDefault();  
  console.log(e.target);  
});
```

[Features](#section--1)
[Operations](#section--2)
[Testimonials](#section--3)

There are two steps to event delegation:

- 1) Add Event Listener to common parent element
- 2) Determine what element originated the event – e.target

```
document.querySelector('.nav__links').addEventListener('click', function (e) {  
  e.preventDefault();  
  // 3. matching Strategy  
  if (e.target.classList.contains('nav__link')) {  
    const id = e.target.getAttribute('href');  
    document.querySelector(id).scrollIntoView({ behaviour: 'smooth' });  
  }  
});
```

We need a matching strategy to only select the 'nav__link' to have the event handler and not all parent elements above due to bubbling up.

We use the same code as before but notice that we cannot use this because it applies to the element that was clicked. Now we need to identify it as e.target.

DOM Traversing

DOM traversing is basically walking through the DOM. We can select an element based on another element. i.e. a child based on a parent.

```
const h1 = document.querySelector('h1');
```

```
// going downwards: child
```

```
console.log(h1.querySelectorAll('highlight'));
```

```
▼ NodeList(2) [span.highlight, span.highlight] ⓘ  
  ▶ 0: span.highlight  
  ▶ 1: span.highlight  
  length: 2  
  ▶ [[Prototype]]: NodeList
```

We can select any children of a parent element with `querySelectorAll`. This will go down as many levels of the DOM tree as needed to find all child elements. If there are other elements on the page with class 'highlight' they would not get selected unless they were children of the h1 parent element.

```
▼ NodeList(9) [text, comment, text, span.highlight, text, br, text, span.highlight, text] ⓘ  
  ▶ 0: text  
  ▶ 1: comment  
  ▶ 2: text  
  ▶ 3: span.highlight  
  ▶ 4: text  
  ▶ 5: br  
  ▶ 6: text  
  ▶ 7: span.highlight  
  ▶ 8: text  
  length: 9  
  ▶ [[Prototype]]: NodeList
```

```
console.log(h1.childNodes);
```

We get a lot of stuff with `childNodes` because nodes can be anything such as text, comments etc..

```
console.log(h1.children);
```

Children will give us a html collection, which is a live collection, so it is updated. It contains all the direct html child elements.

```
▼ HTMLCollection(3) [span.highlight, br, span.highlight] ⓘ  
  ▶ 0: span.highlight  
  ▶ 1: br  
  ▶ 2: span.highlight  
  length: 3  
  ▶ [[Prototype]]: HTMLCollection
```

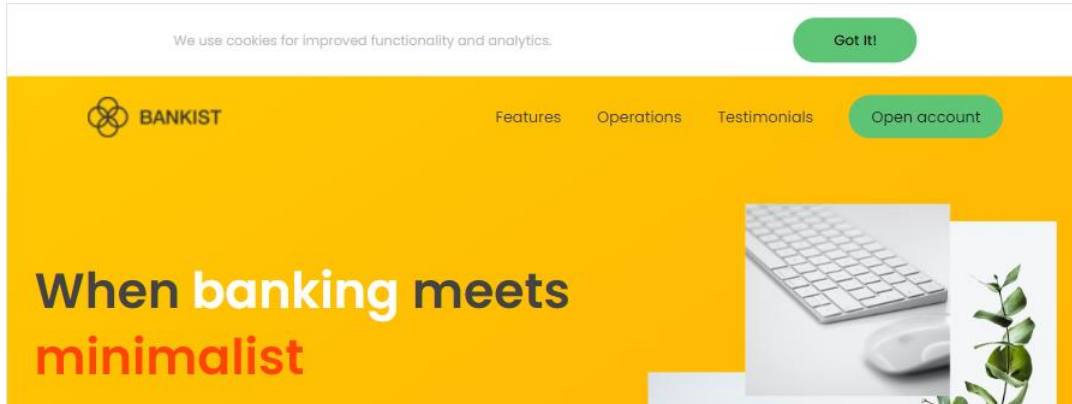
```
h1.firstChild.style.color = 'white';  
h1.lastElementChild.style.color = 'orangered';
```



```
// Going upwards: parents  
console.log(h1.parentNode);  
console.log(h1.parentElement);
```

```
▶ div.header__title  
▶ div.header__title
```

```
h1.closest('.header').style.background = 'var(--gradient-secondary)';
```



firstElementChild is the first child element under the parent and **lastElementChild** is the last.

We can use these to target a specific child in a list of children of the parent element.

We can go up one level to the direct parent of the **h1** element. **parentNode** will give us a node list and **parentElement** will give the element.

We can find the closes parent element, traversing up the DOM tree where we set matching criteria of, for example class name. This is especially useful in event delegation.

```
h1.closest('h1').style.background = 'var(--gradient-primary)';
```

When banking meets minimalist

The closest element to the actual element is itself!

```
// Going Sideways: siblings
console.log(h1.previousElementSibling);
console.log(h1.nextElementSibling);
```

```
null
```

```
<h4>A simpler banking experience for a simpler
life.</h4>
```

```
<div class="header__title">
  <h1>
    When
    <!-- Green highlight effect -->
    <span class="highlight">banking</span>
    meets<br />
    <span class="highlight">minimalist</span>
  </h1>
  <h4>A simpler banking experience for a simpler life.</h4>
  <button class="btn--text btn--scroll-to">Learn more &DownArrow;</button>
  
</div>
```

The h1 element is the first element of its parent so has no previousElementSibling, hence null. The nextElementSibling is the h4 element

```
console.log(h1.previousSibling); ▶ #text  
console.log(h1.nextSibling); ▶ #text
```

previousSibling and NextSibling will give a node details.

```
console.log(h1.parentElement.children);
```

If we want all the siblings of an element we can move up one layer to the parent and call the children method

```
HTMLCollection(4) [h1, h4,  
▶ button.btn--text.btn--scroll-to, img.header__img  
] 1  
▶ 0: h1  
▶ 1: h4  
▶ 2: button.btn--text.btn--scroll-to  
▶ 3: img.header__img  
length: 4  
▶ [[Prototype]]: HTMLCollection
```

```
[...h1.parentElement.children].forEach(function (el) {  
if (el !== h1) el.style.transform = 'scale(0.5)';  
});
```

The html list is not an array or an object but it is iterable so we can use the spread method to put it in an array to loop over in a forEach loop to apply style to all but one element.

When banking meets minimalist

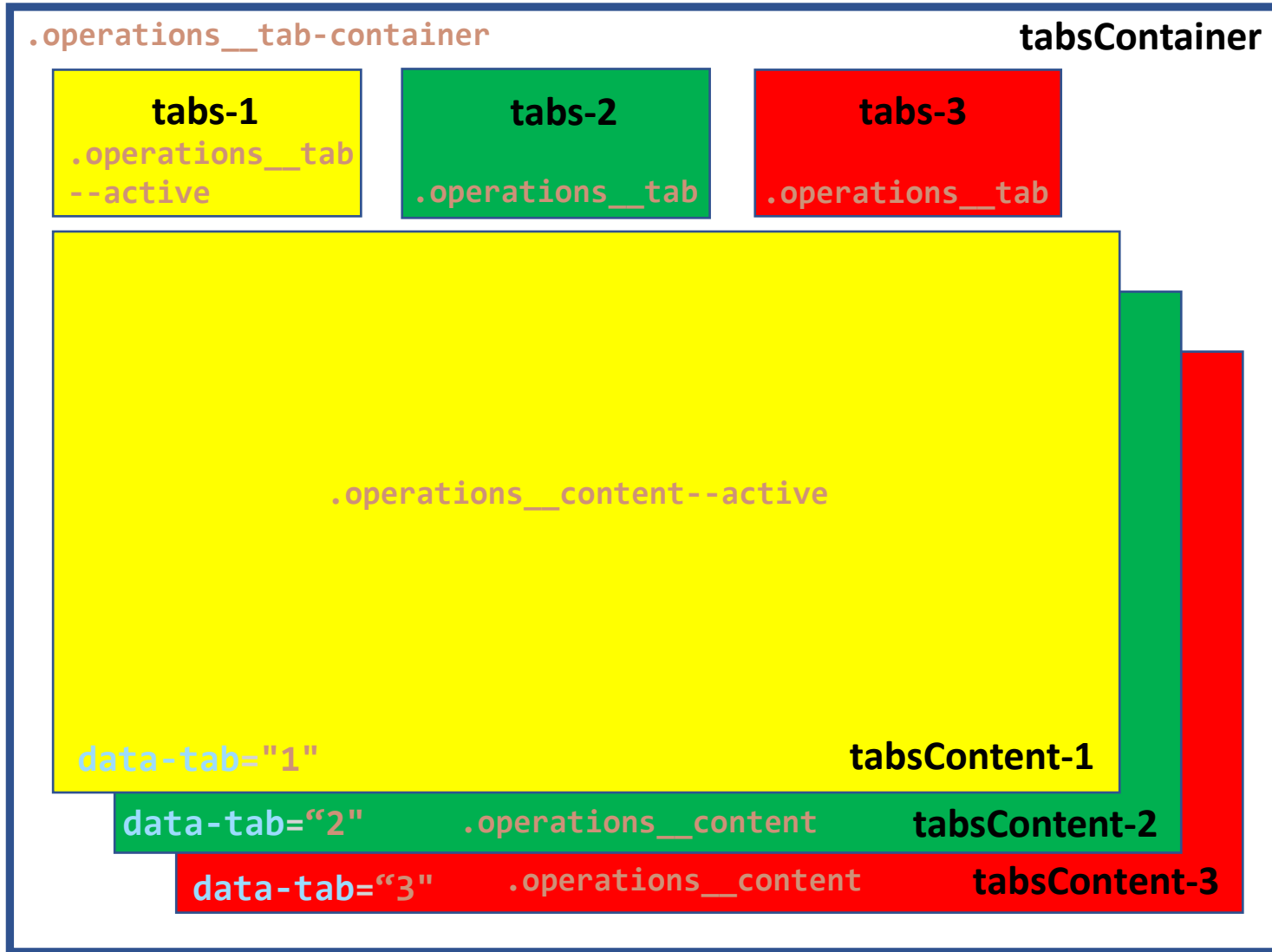
A simpler banking experience for a simpler life.

50% smaller



Building a Tabbed Component

```
const tabs = document.querySelectorAll('.operations__tab');  
const tabsContainer = document.querySelector('.operations__tab-container');  
const tabsContent = document.querySelectorAll('.operations__content');
```



The tabbed component has 3 types of html element. The tabs, the tab container and the tab content.

When one of the Tabs, the corresponding tabsContent becomes active (visible) while the rest are not active(invisible).

```

tabsContainer.addEventListener('click', function (e) {
  const clicked = e.target.closest('.operations__tab');
  // console.log(clicked);

```

We use event delegation to add an event listener on the parent of the tabs and tabsContent elements.

we define a variable of clicked and set it to the e.target closest element that has class operations tab.

```

// guard clause
if (!clicked) return;

```

The guard clause is used to return the function without doing anything, i.e. not clicked.

```

// remove active classes for tab and tab content
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
// console.log(clicked.dataset.tab);

```

```

document
  .querySelector(`.operations__content--${clicked.dataset.tab}`)
  .classList.add('operations__content--active');
});

```

To activate the content area we use select the html element where the class has the value that is equal to the dataset named tab on the corresponding tab button.

Parsing Arguments to Event Handlers

///// Menu Fade Navigation /////

```
nav.addEventListener('mouseover', function (e) {
  if (e.target.classList.contains('nav__link')) {
    const link = e.target;
    const siblings = link.closest('.nav').querySelectorAll('.nav__link');
    const logo = link.closest('.nav').querySelector('img');

    siblings.forEach(el => {
      if (el !== link) el.style.opacity = 0.5;
    });
    logo.style.opacity = 0.5;
  }
});

nav.addEventListener('mouseout', function (e) {
  if (e.target.classList.contains('nav__link')) {
    const link = e.target;
    const siblings = link.closest('.nav').querySelectorAll('.nav__link');
    const logo = link.closest('.nav').querySelector('img');

    siblings.forEach(el => {
      if (el !== link) el.style.opacity = 1;
    });
    logo.style.opacity = 1;
  }
});
```

We can use the code here to add and remove opacity to menu items when the mouse goes over them. However the code is not very efficient because the mouseover and mouse out are almost identical and it does not work for the logo.

The previous code can be refactored to have one `handleHover` function then pass in the `e` and `opacity` with two event listener functions but this is still a bit untidy.

```
const handleHover = function (e, opacity) {
  if (e.target.classList.contains('nav__link')) {
    const link = e.target;
    const siblings =
link.closest('.nav').querySelectorAll('.nav__link');
    const logo = link.closest('.nav').querySelector('img');

    siblings.forEach(el => {
      if (el !== link) el.style.opacity = opacity;
    });
    logo.style.opacity = opacity;
  }
};

nav.addEventListener('mouseover', function (e) {
  handleHover(e, 0.5);
});
nav.addEventListener('mouseout', function (e) {
  handleHover(e, 1);
});
```

We can use the bind method to pass in the opacity value to the handleHover function. The bind means that the 'this' keyword for the handle hover is that of the bound value.

```
const handleHover = function (e) {
  if (e.target.classList.contains('nav__link')) {
    const link = e.target;
    const siblings =
link.closest('.nav').querySelectorAll('.nav__link');
    const logo = link.closest('.nav').querySelector('img');

    siblings.forEach(el => {
      if (el !== link) el.style.opacity = this;
    });
    logo.style.opacity = this;
  }
};

nav.addEventListener('mouseover', handleHover.bind(0.5));
nav.addEventListener('mouseout', handleHover.bind(1));
```

Implementing a sticky Navigation: The scroll Event

```
.nav.sticky {  
  position: fixed;  
  background-color: rgba(255, 255, 255, 0.95);  
}
```

To implement this we are going to give the navigation the class of nav.sticky

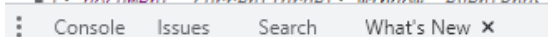
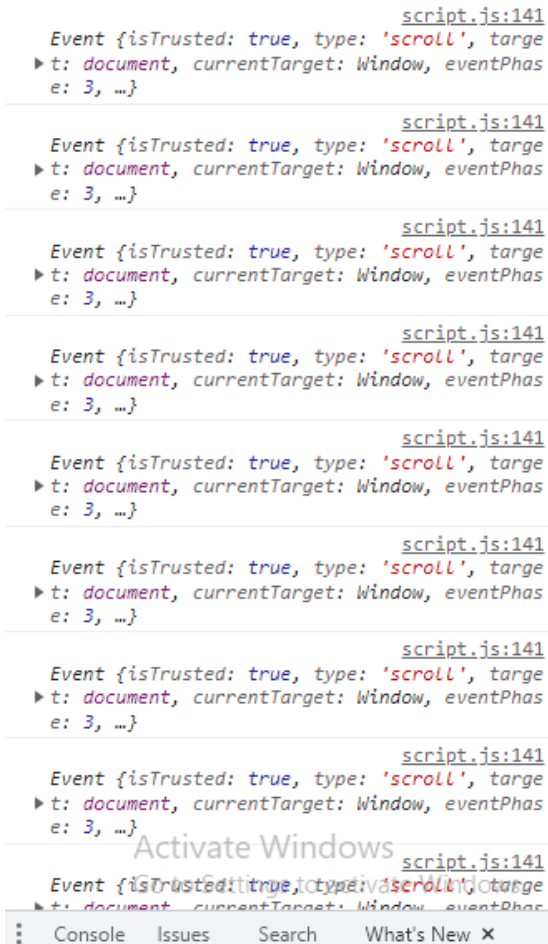
```
window.addEventListener('scroll', function (e) {  
  console.log(e);  
});
```

The scroll event is fired on the window scroll, not the document. As we can see here a small amount of scroll produces many events so it is not very efficient but lets continue anyway.

```
window.addEventListener('scroll', function (e) {  
  console.log(window.scrollY);  
});
```

The scroll.event is useless but we can get the window.scrollY position. Again a small amount of scrolling produces many values. This is the position in the viewport to the top of the page.

When should the navigation become sticky? When we reach the first section. We need to calculate this dynamically.



```
const initialCoords = section1.getBoundingClientRect();
console.log(initialCoords);
```

```
window.addEventListener('scroll', function (e) {
  console.log(window.scrollY);
});
```

script.js:140
DOMRect {x: 0, y: 760, width: 1262, height: 1366.375, top: 760, ...} 
 bottom: 2126.375
 height: 1366.375
 left: 0
 right: 1262
 top: 760
 width: 1262
 x: 0
 y: 760
 ► [[Prototype]]: DOMRect

When should the navigation become sticky? When we reach the first section.

We need to calculate this dynamically and define it as a variable (initialCoords) outside the function.

```
const initialCoords = section1.getBoundingClientRect();
console.log(initialCoords);
```

```
window.addEventListener('scroll', function (e) {
  console.log(window.scrollY);
```

```
    if (window.scrollY > initialCoords.top) nav.classList.add('sticky');
    else nav.classList.remove('sticky');
  });
```

Note the initialCoords value of 760. that is to say section 1 starts at 760px from the top. So we want the nav to become sticky if the window.scrollY value is greater than the initialCoords value.

To make the nav sticky we give it class of sticky.

Note the screen dimensions of 1212 x 760! We can see the scrollY values also. At 765 the nav would become sticky. Scrolling up and reducing scrollY to less than 760 will unstick the nav.

The screenshot shows a web browser with the 'BANKIST' website. The browser's developer tools are open, displaying the 'Dimensions: Responsive' tab. The width is set to 1212 and the height is set to 760. A red arrow points to the height input field. The 'Console' tab is also open, showing a list of scrollY values from 524 to 800, with a red arrow pointing to the value 765. The website content includes a navigation bar with 'Features', 'Operations', and 'Testimonials' links, and a main section titled 'FEATURES' with the text 'Everything you need in a modern bank and more.' and a '100% digital bank' section.

This is very inefficient because of using the scrollY method. A much better way is the new Intersection Observer API.

Implementing sticky Navigation: Intersection Observer API

The Intersection Observer API allows our code to observe the way our code intersects another element or the way it intersects the viewport.

```
const obsCallback = function (entries, observer) {  
  entries.forEach(entry => {  
    console.log(entry);  
  });  
};
```

4) create the observer callback function which is called with two arguments: the entries from the options and the observer. The callback function will get called each time the target element intersects the root element by the threshold amount.

The threshold options can contain multiple values so lets create a function that loops over the entries.

```
const obsOptions = {  
  root: null,  
  threshold: 0.1,  
};
```

3) create an options object with a root property. the root element is our target element is intersecting. We can put in another element or null to observe our target element intersecting the entire viewport. The threshold property is the amount of our target element that intersects the root.

```
const observer = new IntersectionObserver(obsCallback, obsOptions);
```

1) we first create a new IntersectionObserver and pass in a callback function and an options object.

```
observer.observe(section1);
```

2) We then call that observer and give it a method of observe and tell it what element we would like to observe.

Note that we get an intersection observer entry before any scrolling occurs but when the scroll reaches section1, a second real intersection observer entry is created. In the intersection observer entry we get the intersection ratio, isIntersecting and target.

The screenshot shows a web browser window with a page titled "OPERATIONS" and the heading "Everything as simple as possible, but no simpler." Below the heading are three buttons: "01 Instant Transfers" (yellow), "02 Instant Loans" (green), and "03 Instant Closing" (red). The "01 Instant Transfers" button is highlighted. Below the buttons is a section titled "Tranfser money to anyone, instantly! No fe" (partially visible) with a paragraph of Lorem ipsum text. A red arrow points from the text "Ut enim ad" to the "intersectionRatio" property in the second console log entry. Another red arrow points from the text "exercitation ullamco laboris nisi ut aliquip ex ea commo" to the "isIntersecting" property in the same entry. A third red arrow points from the text "Tranfser" to the "target" property in the same entry. The console log shows two entries of type "IntersectionObserverEntry". The first entry is from "script.js:155" and has a time of "6775.09999999940395". The second entry is also from "script.js:155" and has a time of "69378.10000002384". The second entry is expanded, showing properties: "boundingClientRect", "intersectionRatio", "intersectionRect", "isIntersecting", "isVisible", "rootBounds", "target", and "[[Prototype]]".

Dimensions: Responsive 1212 x 760 75% No throttling

OPERATIONS

Everything as simple as possible, but no simpler.

01 Instant Transfers 02 Instant Loans 03 Instant Closing

Tranfser money to anyone, instantly! No fe

Lorem ipsum dolor sit amet, consectetur adipisicing elit, incididunt ut labore et dolore magna aliqua. Ut enim ad exercitation ullamco laboris nisi ut aliquip ex ea commo

```
script.js:155
IntersectionObserverEntry {time: 6775.0999
999940395, rootBounds: DOMRectReadOnly, bo
undingClientRect: DOMRectReadOnly, interse
ctionRect: DOMRectReadOnly, isIntersectin
g: true, ...}
  ▶ boundingClientRect: DOMRectReadOnly {x: 0
intersectionRatio: 0.5823124647140503
  ▶ intersectionRect: DOMRectReadOnly {x: 0,
isVisible: true
isVisible: false
  ▶ rootBounds: DOMRectReadOnly {x: 0, y: 0,
target: section#section--1.section
time: 6775.0999999940395
  ▶ [[Prototype]]: IntersectionObserverEntry

script.js:155
IntersectionObserverEntry {time: 69378.10000002384, rootBounds: DOMRectReadOnly,
▶ boundingClientRect: DOMRectReadOnly, intersectionRect: DOMRectReadOnly, isInters
ecting: true, ...}
  ▶ boundingClientRect: DOMRectReadOnly {x: 0, y: -1548, width: 962, height: 1567.
intersectionRatio: 0.012243269942700863
  ▶ intersectionRect: DOMRectReadOnly {x: 0, y: 0, width: 962, height: 19.1875, to
isIntersecting: true
isVisible: false
  ▶ rootBounds: DOMRectReadOnly {x: 0, y: 0, width: 962, height: 760, top: 0, ...}
target: section#section--1.section
time: 69378.10000002384
  ▶ [[Prototype]]: IntersectionObserverEntry
```

```
const obsOptions = {  
  root: null,  
  threshold: [0, 0.2],  
};
```

5) Now the options object is modified so that the observer function is called when 0% is intersection or 20% is intersecting (for scroll up actions).

1) Now that we have finished experimenting with the intersection observer lets implement the sticky nav. When do we want the nav bar to become sticky? When the header element is completely out of view.

```
const header = document.querySelector('.header');
```

2) First we define the header

```
const stickyNav = function (entries) {  
  const [entry] = entries;  
  console.log(entry);  
};
```

5) In the stickyNav function we de-structure the options threshold array to get the entry

```
const headerObserver = new IntersectionObserver(stickyNav, {  
  root: null,  
  threshold: 0,  
});
```

3) Then a new IntersectionObserver that calls the stickynav function when the header is completely out of the viewport, i.e. threshold of zero.

```
headerObserver.observe(header);
```

4) We activate the observer on the header.

6) We can see that once the header is out of view the Intersection Observer fires.

The screenshot shows a web browser window with a responsive design view. The page content includes a header, a section titled "100% digital bank" with a green monitor icon, and a section titled "Watch your money grow" with a green line graph icon. The Chrome DevTools console is open, showing two log entries from script.js:155. The first entry is an IntersectionObserverEntry object with time 44.30000001192093, rootBounds: DOMRectReadOnly, boundingClientRect: DOMRectReadOnly, intersectionRect: DOMRectReadOnly, and isIntersecting: true. The second entry is an IntersectionObserverEntry object with time 1989.4000000059605, rootBounds: DOMRectReadOnly, boundingClientRect: DOMRectReadOnly, intersectionRect: DOMRectReadOnly, isIntersecting: false, isVisible: false, and target: header.header. The console also shows the prototype chain for the second entry.

```
IntersectionObserverEntry {time: 44.30000001192093, rootBounds: DOMRectReadOnly, boundingClientRect: DOMRectReadOnly, intersectionRect: DOMRectReadOnly, isIntersecting: true, ...}

IntersectionObserverEntry {time: 1989.4000000059605, rootBounds: DOMRectReadOnly, boundingClientRect: DOMRectReadOnly, intersectionRect: DOMRectReadOnly, isIntersecting: false, isVisible: false, target: header.header, time: 1989.4000000059605, [[Prototype]]: IntersectionObserverEntry}
```

7) We now want to create the condition that when the header is intersecting the sticky class is applied.

7) We now want to create the condition that when the header is not intersecting the sticky class is applied and vice versa.

```
const stickyNav = function (entries) {  
  const [entry] = entries;  
  console.log(entry);  
  if (!entry.isIntersecting) nav.classList.add('sticky');  
  else nav.classList.remove('sticky');  
};  
  
const headerObserver = new IntersectionObserver(stickyNav, {  
  root: null,  
  threshold: 0,  
});  
headerObserver.observe(header);
```

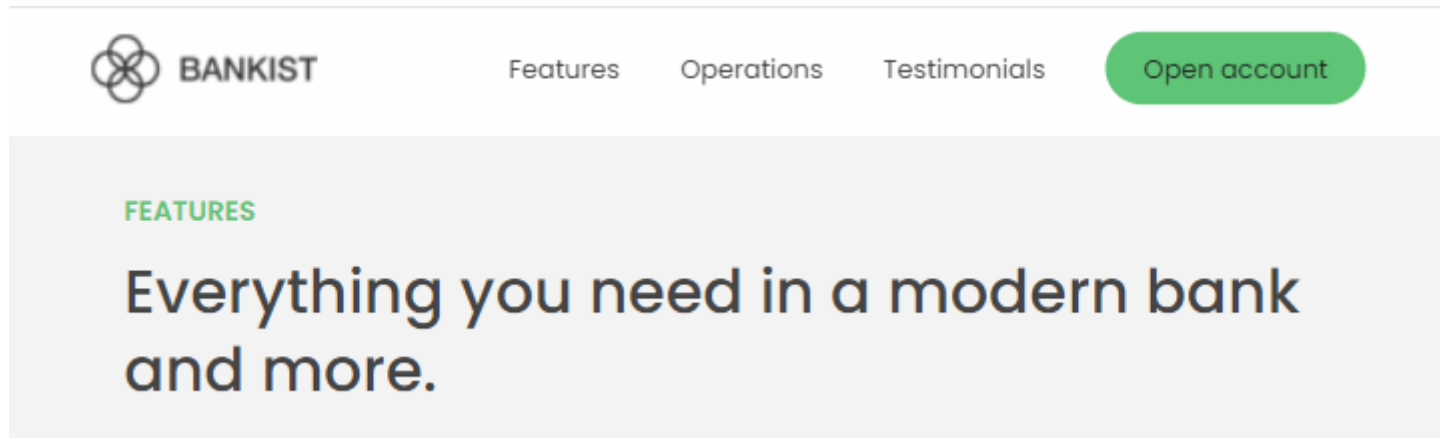
[Features](#)[Operations](#)[Testimonials](#)[Open account](#)

Everything you need in a modern bank
and more.

Notice how the nav is added when the header is completely out of view but that it immediately overlays the first section.

8) We can solve the overlap by adding another parameter to the IntersectionObserver options called `rootMargin`. Here we set this to the negative height of the nav element in pixels (percentage or rem does not work).

```
const stickyNav = function (entries) {  
  const [entry] = entries;  
  console.log(entry);  
  if (!entry.isIntersecting) nav.classList.add('sticky');  
  else nav.classList.remove('sticky');  
};  
  
const headerObserver = new IntersectionObserver(stickyNav, {  
  root: null,  
  threshold: 0,  
  rootMargin: '-90px',  
});  
headerObserver.observe(header);
```



Notice how the nav is now getting the sticky class 90px before it is intersecting.

8) But with responsive websites the nav height will change so it is best to calculate this dynamically by using `getBoundingClientRect.height`

```
const navHeight = nav.getBoundingClientRect().height;

const stickyNav = function (entries) {
  const [entry] = entries;
  console.log(entry);
  if (!entry.isIntersecting) nav.classList.add('sticky');
  else nav.classList.remove('sticky');
};

const headerObserver = new IntersectionObserver(stickyNav, {
  root: null,
  threshold: 0,
  rootMargin: `-${navHeight}px`,
});
headerObserver.observe(header);
```

Above is the finished code to implement sticky navigation.

The Intersection Observer API is the much better solution because it is more efficient by not using the `window.scrollY` property which is continually changing. The Intersection Observer API reacts when an observed element reaches a threshold of intersection only.

Revealing Elements on scroll - Intersection Observer API

```
.section--hidden {  
  opacity: 0;  
  transform: translateY(8rem);  
}
```

We will achieve this by removing this class to elements as they scroll into view using the Intersection Observer API

```
const allSections = document.querySelectorAll('section');
```

1) Select all elements

that are sections

```
const revealSection = function (entries, observer) {  
  const [entry] = entries;  
  if (!entry.isIntersecting) return;  
  
  entry.target.classList.remove('section--hidden');  
  observer.unobserve(entry.target);  
};
```

4) RevealSection function takes an entries and observer parameter.

If the entry is not intersection then the function returns.

Otherwise the target has the hidden classList removed and the observer is set to unobserve.

```
const sectionObserver = new IntersectionObserver(revealSection, {  
  root: null,  
  threshold: 0.15,  
});
```

3) A new IntersectionObserver is created and the revealSection function and options parameters are defined.

```
allSections.forEach(function (section) {  
  sectionObserver.observe(section);  
  section.classList.add('section--hidden');  
});
```

2) Within a forEach loop the sectionObserver is activated and the hidden class applied.

Lazy Loading Images - Intersection Observer API

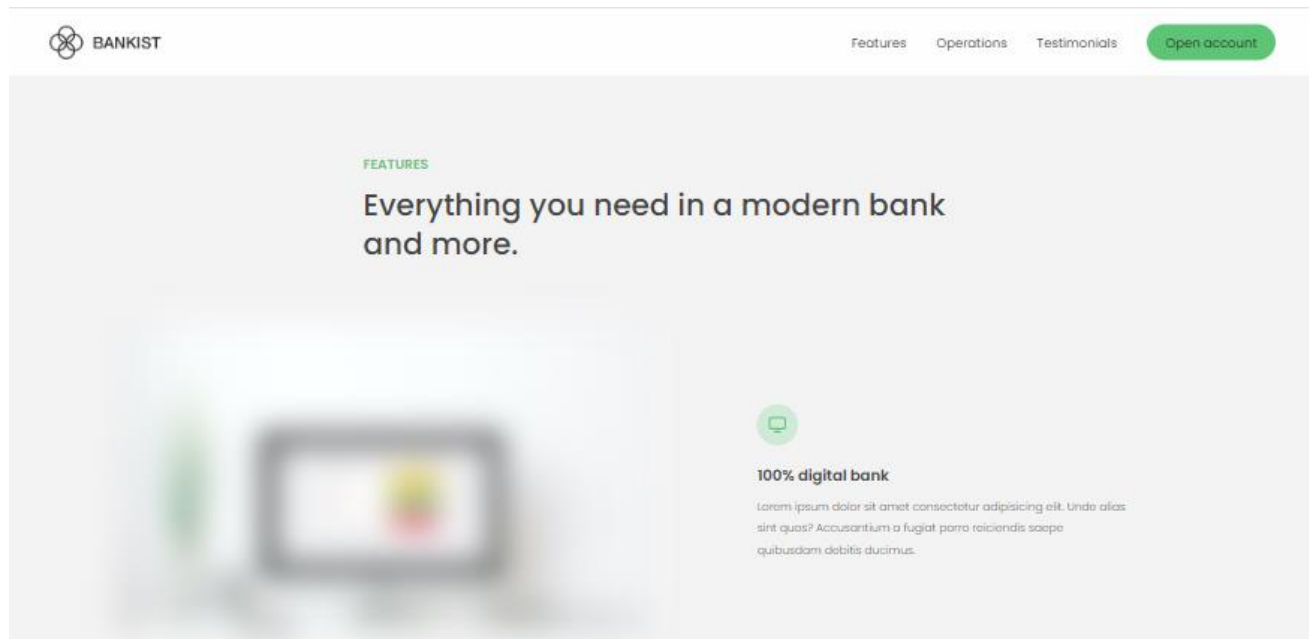
```

```

```
.lazy-img {  
  filter: blur(20px);  
}
```

The image html element has a source of a low resolution 200 x 120px 17kb image. Then a class of lazy-img applied.

The lazy-img class has a filter called blur which blurs the image to disguise its low quality and resolution.



The image html has a data-src property which points to the high quality image which is about 400mb in size.

The objective here is that when the image comes into view the low quality image is replaced with the full size image and the css blur is removed.

```
const imgTargets = document.querySelectorAll('img[data-src]');
```

1) Select all elements that are images and have the data-src property set.

```
const loadImg = function (entries, observer) {  
  const [entry] = entries;
```

```
  if (!entry.isIntersecting) return;  
  // Replace src with data-src  
  entry.target.src = entry.target.dataset.src;
```

```
  entry.target.addEventListener('load', function () {  
    entry.target.classList.remove('lazy-img');  
  });
```

```
  observer.unobserve(entry.target);  
};
```

```
const imgObserver = new IntersectionObserver(loadImg, {  
  root: null,  
  threshold: 0,  
  rootMargin: '200px',  
});
```

```
imgTargets.forEach(img => imgObserver.observe(img));
```

4) loadImg function takes an entries and observer parameter.

If the entry is not intersecting then the function returns.

Otherwise:

- a) The target src is replaced with the dataset src.
- b) An event listener is created to detect a load event and performs an anonymous function that removes the lazy-img class with the blur.
- c) Once the image has loaded the observer is set to unobserve.

3) A new IntersectionObserver is created and the loading function and options object is defined.

2) Within a forEach loop apply the observer to the images.

Building a slider component

```
<div class="slide"></div> style="transform: translateX(0%);"
<div class="slide"></div> style="transform: translateX(100%);"
<div class="slide"></div> style="transform: translateX(200%);"
<div class="slide"></div> style="transform: translateX(300%);"
```

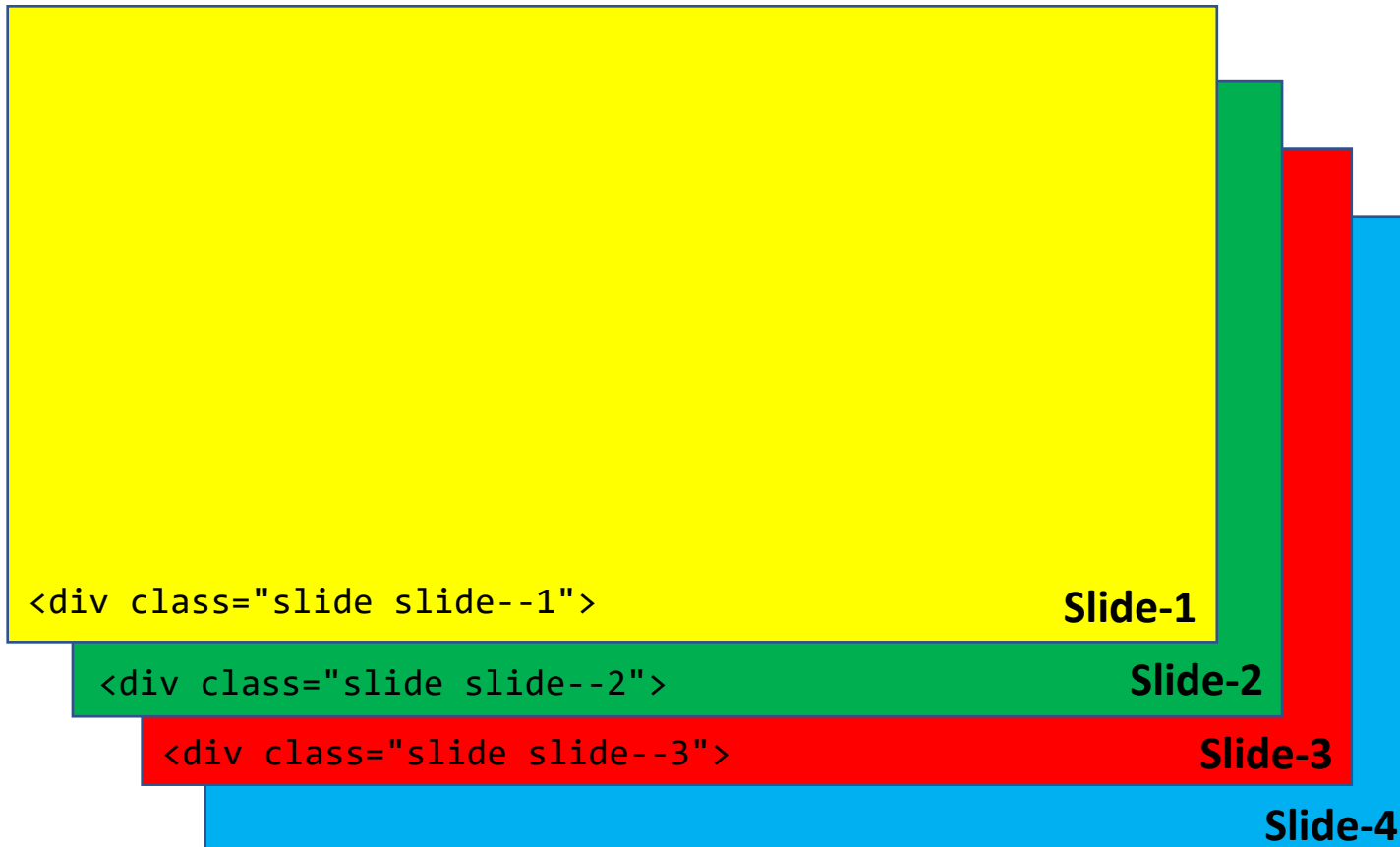
Currently the slides are all on top of each other but what we want to do is line them up horizontally, so that they are side by side.

translateX(0%)

translateX(100%)

translateX(200%)

translateX(300%)



For testing purposes we can reduce the size of the slider container to 40% of its normal size and set it to visible.

```
const slider = document.querySelector('.slider');  
slider.style.transform = 'scale(0.4) translateX(-800px)';  
slider.style.overflow = 'visible';
```

`const slides = document.querySelectorAll('.slide');` Each slide is put into variable slides

```
slides.forEach((s, i) => (s.style.transform = `translateX(${100 * i}%)`));
```

Now we loop over slides with a `forEach` loop with parameters of `s` for slide and `i` for iteration. In each loop we define the `s.style.transform` calculating the value from the index multiplied by 100 to get 0%, 100% 200% etc..



Now we need a current slide variable which is going to store the `i` value as we click the next or previous button. We also need a variable to mark the last slide or `maxSlide` which is the number of times the loop is iterated or `length`.

For the next slide button we add an event listener which if the `curSlide` value is equal to the last slide (`length - 1`: `length` is four and slide iterations are zero based) then it will set the `curSlide` to zero. Else it will increment `curSlide` by 1.

```
const slides = document.querySelectorAll('.slide');
const btnLeft = document.querySelector('.slider__btn--left');
const btnRight = document.querySelector('.slider__btn--right');
let curSlide = 0;
const maxSlide = slides.length;

slides.forEach((s, i) => (s.style.transform = `translateX(${100 * i}%)`));
// next slide
btnRight.addEventListener('click', function () {
  if (curSlide === maxSlide - 1) {
    curSlide = 0;
  } else {
    curSlide++;
  }

  Now we transform the slide positions decreasing them all by -100% within
  another forEach loop.

  slides.forEach(
    (s, i) => (s.style.transform = `translateX(${100 * (i - curSlide)}%)` )
  );
});
```

```

const slides = document.querySelectorAll('.slide');
const btnLeft = document.querySelector('.slider__btn--left');
const btnRight = document.querySelector('.slider__btn--right');
let curSlide = 0;
const maxSlide = slides.length;

const goToSlide = function (slide) {
  slides.forEach(
    (s, i) => (s.style.transform = `translateX(${100 * (i - slide)}%)`)
  );
};
goToSlide(0);

```

Code re-factoring: goToSlide is its own function with default of goToSlide(0).

```

// next slide
const nextSlide = function () {
  if (curSlide === maxSlide - 1) {
    curSlide = 0;
  } else {
    curSlide++;
  }

  goToSlide(curSlide);
};

```

Code re-factoring: nextSlide is its own function with the same logic as before.

```

btnRight.addEventListener('click', nextSlide);

```

Code re-factoring: The button right event listener now calls the next slide function on a click.

```
// next slide
const nextSlide = function () {
  if (curSlide === maxSlide - 1) {
    curSlide = 0;
  } else {
    curSlide++;
  }
}
```

```
goToSlide(curSlide);
```

```
};
```

```
// prev slide
```

```
const prevSlide = function () {
  if (curSlide === 0) {
    curSlide = maxSlide - 1;
  } else {
    curSlide--;
  }
}
```

```
goToSlide(curSlide);
```

```
};
```

```
btnRight.addEventListener('click', nextSlide);
btnLeft.addEventListener('click', prevSlide);
```

The previous slide function is almost identical to the next slide except the condition is modified. If the current slide is zero then we want to set the curSlide to the last slide. Else we want to decrement the current slide by 1.

Now we can add an event listener for the previous button and add a new function called previous slide.

```
const slider = document.querySelector('.slider');
slider.style.transform = 'scale(0.4) translateX(-800px)';
slider.style.overflow = 'visible';
```

This testing code is also annulled

```

btnRight.addEventListener('click', nextSlide);
btnLeft.addEventListener('click', prevSlide);

document.addEventListener('keydown', function (e) {
  if (e.key === 'ArrowLeft') prevSlide();
  e.key === 'ArrowRight' && nextSlide();
});

```

We also want the arrow left and arrow right keys on the keyboard to slide the images so we can add an event listener for a 'keydown' event.

The function for this event can use if statement or short circuiting.

```
<div class="dots"></div>
```

For the dots container we have an empty dots div.

```
const dotContainer = document.querySelector('.dots');
```

Declare dotContainer as a variable

```
let curSlide = 0;
const maxSlide = slides.length;
```

```

const createDots = function () {
  slides.forEach(function (_, i) {
    dotContainer.insertAdjacentHTML(
      'beforeend',
      `<button class="dots__dot" data-slide="${i}"></button>`
    );
  });
};

createDots();

```

We call the CreateDots function.

Declare createDots as a function and forEach slide insertAdjacentHTML to the dotsContainer as the last item before the end of the container. Note the throwaway variable defined as _. This is because we do not need the slide s.

We insert a button html element with class of 'dots__dot' and data attribute of data-slide set to the I value.



Now we can attach an event listener to the dotContainer to listen to elements that have a class of 'dots__dot'.

The slide is de-structured and set equal to the event target dataset which we previously defined as the i value.

goToSlide is now called on the value of i which is equal to slide.

```
dotContainer.addEventListener('click', function (e) {  
  if (e.target.classList.contains('dots__dot')) {  
    const { slide } = e.target.dataset;  
    goToSlide(slide);  
  }  
});
```

```
const activateDot = function (slide) {
  document
    .querySelectorAll('.dots__dot')
    .forEach(dot => dot.classList.remove('dots__dot--active'));

  document
    .querySelector(`.dots__dot[data-slide="${slide}"]`)
    .classList.add('dots__dot--active');
};
```

We want the dot that corresponds to the image on screen to have an active class.

We can do this with an activateDot function that takes slide as a variable and remove the active class from all the dots then add it to only the dot that has the same data attribute as the current slide.

```
goToSlide(0);
activateDot(curSlide);

dotContainer.addEventListener('click', function (e) {
  if (e.target.classList.contains('dots__dot')) {
    const { slide } = e.target.dataset;
    goToSlide(slide);
    activateDot(slide);
  }
});
```

We just need to call the activateDot function in the goToSlide, prevSlide and NextSlide functions, And in the event listener for the dot buttons clicks.

Now the functionality is complete. It just needs a little refactoring to tidy it up.

```

const slider = function () {
  const slides = document.querySelectorAll('.slide');
  const btnLeft = document.querySelector('.slider__btn--left');
  const btnRight = document.querySelector('.slider__btn--right');
  const dotContainer = document.querySelector('.dots');

  let curSlide = 0;
  const maxSlide = slides.length;

  // Functions
  const createDots = function () {
    slides.forEach(function (_, i) {
      dotContainer.insertAdjacentHTML(
        'beforeend',
        `<button class="dots__dot" data-slide="${i}"></button>`
      );
    });
  };

  const activateDot = function (slide) {
    document
      .querySelectorAll('.dots__dot')
      .forEach(dot => dot.classList.remove('dots__dot--active'));

    document
      .querySelector(`.dots__dot[data-slide="${slide}"]`)
      .classList.add('dots__dot--active');
  };

  const goToSlide = function (slide) {
    slides.forEach(
      (s, i) => (s.style.transform = `translateX(${100 * (i - slide)}%)`)
    );
  };

  const nextSlide = function () {
    if (curSlide === maxSlide - 1) {
      curSlide = 0;
    } else {
      curSlide++;
    }
    goToSlide(curSlide);
    activateDot(curSlide);
  };
}

```

All the code to make the slider work is placed inside a slider function so as not to pollute the global namespace with unneeded variables and functions.

At the end of the function it is called with slider();

The layout is divided into 3 sections:

At the top the variable declarations needed.

In the middle the functions

At the end the event listeners.

```
const prevSlide = function () {
  if (curSlide === 0) {
    curSlide = maxSlide - 1;
  } else {
    curSlide--;
  }
  goToSlide(curSlide);
  activateDot(curSlide);
};
```

```
const init = function () {
  goToSlide(0);
  createDots();

  activateDot(0);
};
init();
```

```
// Event handlers
btnRight.addEventListener('click', nextSlide);
btnLeft.addEventListener('click', prevSlide);

document.addEventListener('keydown', function (e) {
  if (e.key === 'ArrowLeft') prevSlide();
  e.key === 'ArrowRight' && nextSlide();
});
```

```
dotContainer.addEventListener('click', function (e) {
  if (e.target.classList.contains('dots__dot')) {
    const { slide } = e.target.dataset;
    goToSlide(slide);
    activateDot(slide);
  }
});
};

slider();
```

An init function is added to initialise the slider and call the three functions that set the current slide to zero, createDots and activate dot.

The slider is being built dynamically depending on the number of images or slides in our html markup.

Lifecycle DOM events

Lifecycle DOM events are events that occur from the first moment the page is accessed.

The first event is DOM content loaded event which is the moment that all elements have been parsed and added to the DOM tree.

```
document.addEventListener('DOMContentLoaded', function (e) {  
  console.log('HTML parsed and DOM tree built', e);  
});
```

```
HTML parsed and script.js:493  
DOM tree built  
Event {isTrusted: true, type:  
  'DOMContentLoaded', target:  
    document, currentTarget:  
      document, eventPhase: 2, ...}
```

```
isTrusted: true  
bubbles: true  
cancelBubble: false  
cancelable: false  
composed: false  
currentTarget: null  
defaultPrevented: false  
eventPhase: 0  
▶ path: (2) [document, Window]  
returnValue: true  
▶ srcElement: document  
▶ target: document  
timeStamp: 42.400000059604  
type: "DOMContentLoaded"  
▶ [[Prototype]]: Event
```

The screenshot shows a web browser with the Bankist landing page. The Network tab is open, displaying a list of resources. A red box highlights the 'DOMContentLoaded' event at 41 ms.

Name	Status	Type	Initiator	Size	T	Waterfall
hero.png	304	png	(index)	298 B	5...	
digital-lazy.jpg	304	jpeg	(index)	297 B	6...	
grow-lazy.jpg	304	jpeg	(index)	296 B	6...	
card-lazy.jpg	304	jpeg	(index)	297 B	7...	
img-1.jpg	304	jpeg	(index)	299 B	7...	
img-2.jpg	304	jpeg	(index)	299 B	4...	
img-3.jpg	304	jpeg	(index)	298 B	4...	
img-4.jpg	304	jpeg	(index)	298 B	4...	
icon.png	304	png	(index)	296 B	4...	
icons.svg	304	svg+xml	(index)	298 B	3...	
pxi8yp8kv8JHgFvRlDz8Z1xlF...	200	font	css?family=P...	(mem...	0...	
pxi8yp8kv8JHgFvRlJfecnFHG...	200	font	css?family=P...	(mem...	0...	
pxi8yp8kv8JHgFvRlGT9Z1xlF...	200	font	css?family=P...	(mem...	0...	
pxi8yp8kv8JHgFvRlEj6Z1xlFd...	200	font	css?family=P...	(mem...	0...	
ws	101	webso...	(index):378	0 B	P...	
icon.png	304	png	Other Window	296 B	3...	

21 requests | 4.5 kB transferred | 5.6 MB resources | 50% finished | 48 ms | DOMContentLoaded: 41 ms

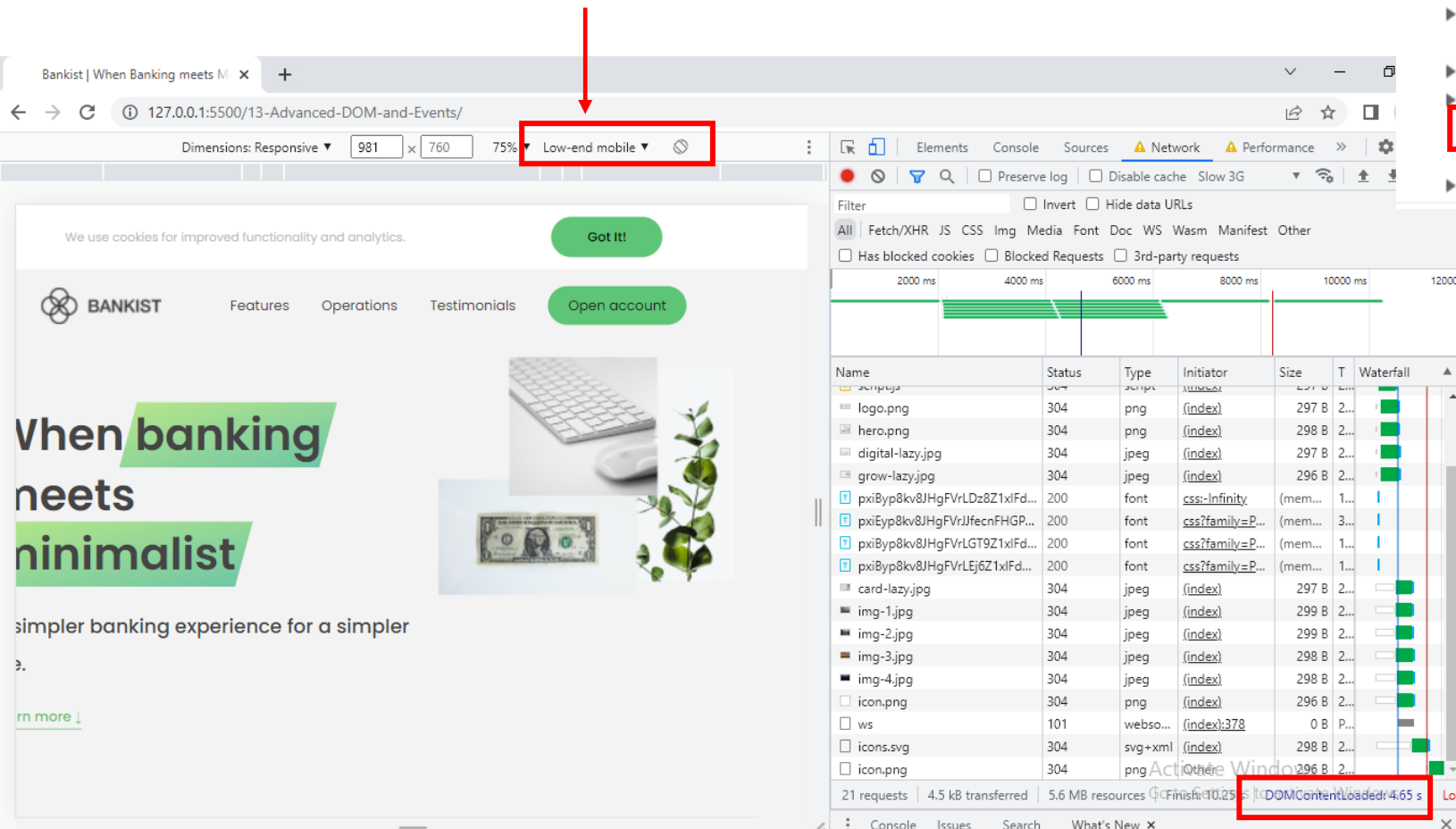
It takes 48ms to load all the elements into the DOM tree.

Ideally we want our JavaScript code to only be executed after the DOM has loaded.

We can achieve this by putting our script tag at the end of the html document.

If throttling is set to the speed of a low end mobile we see that the page takes about 4.65 seconds to load.

```
HTML parsed and DOM tree built
script.js:493
Event {isTrusted: true, type: 'DOMContentLoaded', target: document, currentTarget: document, eventPhase: 2, ...}
isTrusted: true
bubbles: true
cancelBubble: false
cancelable: false
composed: false
currentTarget: null
defaultPrevented: false
eventPhase: 0
path: (2) [document, Window]
returnValue: true
srcElement: document
target: document
timestamp: 4627.599999994039
type: "DOMContentLoaded"
[[Prototype]]: Event
```



The load event is fired by the window as soon as the html is parsed including all the images and external resources and we can listen to this event.

```
window.addEventListener('load', function (e) {  
  console.log('page fully loaded', e);  
});
```

The load event on a slow mobile connection for this page takes 8.27 seconds aprox.s

```
page fully loaded script.js:497  
Event {isTrusted: true, type:  
'load', target: document, curr  
entTarget: Window, eventPhase:  
2, ...} 1  
  isTrusted: true  
  bubbles: false  
  cancelBubble: false  
  cancelable: false  
  composed: false  
  ▶ currentTarget: Window {window  
  defaultPrevented: false  
  eventPhase: 2  
  ▶ path: [Window]  
  returnValue: true  
  ▶ srcElement: document  
  ▶ target: document  
  timestamp: 8196.30000011921  
  type: "load"  
  ▶ [[Prototype]]: Event
```

The screenshot shows a web browser with the URL `127.0.0.1:5500/13-Advanced-DOM-and-Events/`. The browser's developer tools are open, showing the 'Network' tab with a list of resources and a 'Performance' tab with a timeline. A red box highlights the 'Low-end mobile' connection type in the browser's address bar. Another red box highlights the 'Load: 8.27 s' in the performance timeline. A third red box highlights the 'timestamp: 8196.30000011921' in the console log.

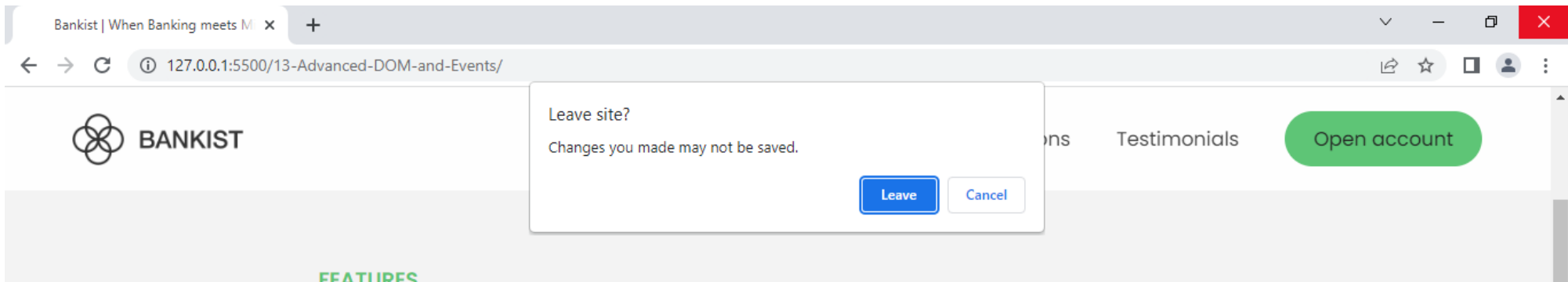
Name	Status	Type	Initiator	Size	T.	Waterfall
script.js	200	script	(index)	297 B	2...	
logo.png	304	png	(index)	298 B	2...	
hero.png	304	png	(index)	298 B	2...	
digital-lazy.jpg	304	jpeg	(index)	297 B	2...	
grow-lazy.jpg	304	jpeg	(index)	296 B	2...	
pxiByp8kv8JHgFVrLDz8Z1xiFd2JQEk...	200	font	css?family=Pop...	(memor...	1...	
pxiEyp8kv8JHgFVrJfecnFHGPc.woff2	200	font	css?family=Pop...	(memor...	1...	
pxiByp8kv8JHgFVrLT9Z1xiFd2JQEk...	200	font	css?family=Pop...	(memor...	1...	
pxiByp8kv8JHgFVrLEj6Z1xiFd2JQEk...	200	font	css?family=Pop...	(memor...	1...	
card-lazy.jpg	304	jpeg	(index)	297 B	2...	
img-1.jpg	304	jpeg	(index)	299 B	2...	
img-2.jpg	304	jpeg	(index)	299 B	2...	
img-3.jpg	304	jpeg	(index)	298 B	2...	
img-4.jpg	304	jpeg	(index)	298 B	2...	
icon.png	304	png	(index)	296 B	2...	
ws	101	websocket	(index):378	0 B	P...	
icons.svg	304	svg+xml	(index)	298 B	2...	
icon.png	304	png	(index)	296 B	2...	

21 requests | 4.5 kB transferred | 5.6 MB resources | Finish: 10.33 s | DOMContentLoaded: 4.55 s | Load: 8.27 s

The before unload event also gets fired on window load. This event is fired immediately before they close the page.

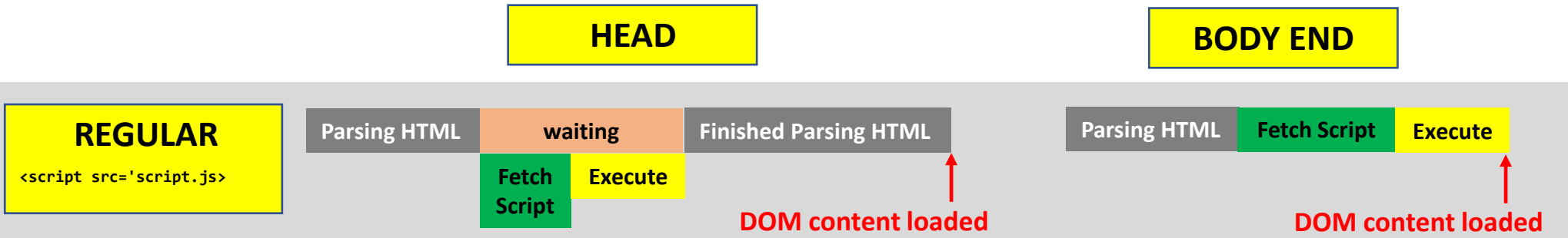
```
window.addEventListener('beforeunload', function (e) {  
    e.preventDefault();  
    console.log(e);  
    e.returnValue = '';  
});
```

The return value set to an empty string will force the browser to display a confirmation of closing page message but it only works if something has been clicked like a nav link first.



The return value message used to be customisable but due to developer abuse, it is not the default message only. This should only be used when a user is in the middle of a task such as filling out a form when data would actually be lost.

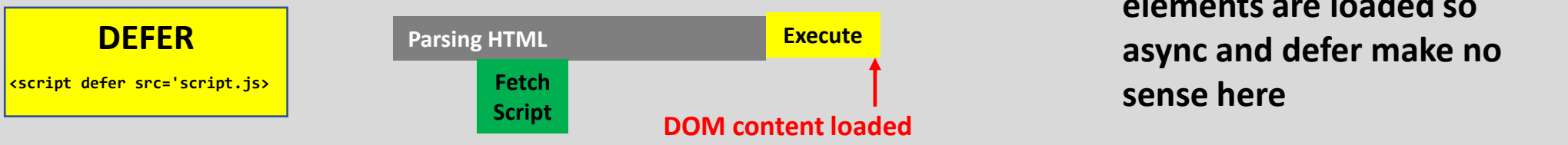
Efficient Script loading: Async and Defer



As the user loads the page the browser will parse the html and create the DOM tree. When it gets to the JavaScript file the DOM tree loading is paused while the script is parsed then it terminates loading the DOM.



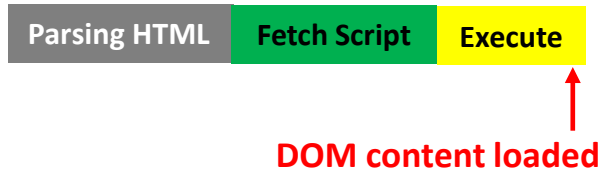
The script is loaded at the same time as the html is parsed however the html parsing is still stopped while the script is executed.



In the body fetching and executing the script always happens after the html elements are loaded so async and defer make no sense here

The script is still loaded asynchronously but the script is executed at the end so the html parsing is never interrupted. In many occasions this is wat we want.

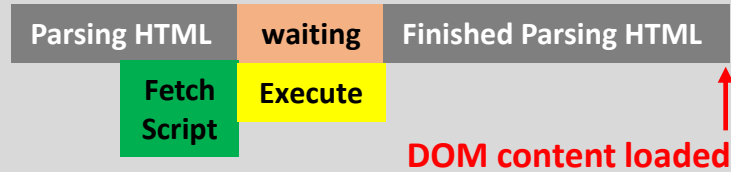
END OF BODY



- Scripts are fetched and executed **after the HTML is completely parsed**

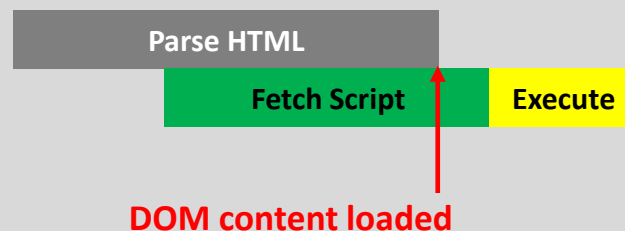
Use if you need to support old browsers

ASYNC IN HEAD

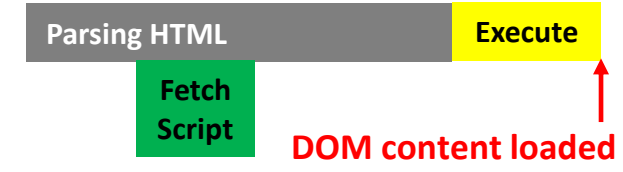


- Scripts are fetched **asynchronously** and executed **immediately**
- Usually the DOMContentLoaded event waits for **all** scripts to execute, except for async scripts. So, DOMContentLoaded does **not** wait for an async script

Use for third party scripts where order does not matter (eg Google Analytics)

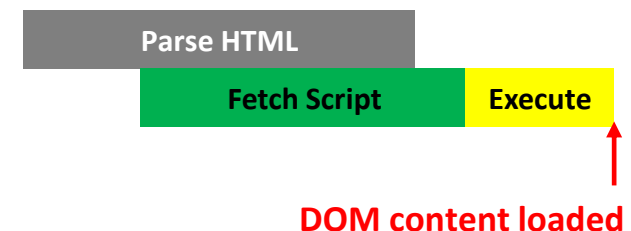


DEFER IN HEAD



- Scripts are fetched **asynchronously** and executed **after the html is completely parsed.**
- DOMContentLoaded event fires **after defer script is executed.**
- Scripts are executed **in order.**

This is the overall best solution!
Use it for own scripts and when order matters (i.e. adding an external library)



If our code needs an third party external library then the external library will be loaded first before our own script so that we can use code from the external library so we need async because the order is important.

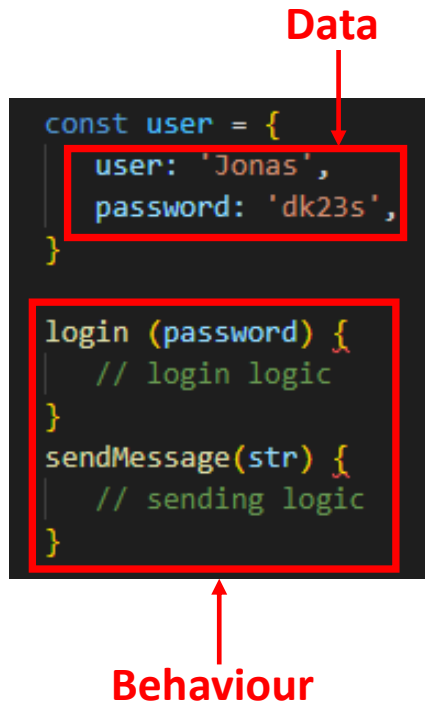
```
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <meta http-equiv="X-UA-Compatible" content="ie=edge" />
  <link rel="shortcut icon" type="image/png" href="img/icon.png" />

  <link
    href="https://fonts.googleapis.com/css?family=Poppins:300,400,500,600&display=
swap"
    rel="stylesheet"
  />
  <link rel="stylesheet" href="style.css" />
  <title>Bankist | When Banking meets Minimalist</title>
  <script defer src="script.js"></script>
</head>
```

Object Orientated Programming (OOP) with JavaScript

What is object orientated programming (OOP) ?

- Object-orientated programming (OOP) is a programming paradigm based on the concept of objects.
- We use Objects to **model** (describe) real world or abstract features.
 - ← E.g. A to do list
 - ↑ E.g. A html component or data structure
- Objects may contain data (properties) and code (methods). By using objects, we pack **data and the corresponding behaviour** into one block.
- 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 **organising** code, to make it **more flexible and easier to maintain** (avoid "spaghetti code").



The diagram shows a JavaScript object definition. The object is named 'user' and contains two properties: 'user' (value 'Jonas') and 'password' (value 'dk23s'). It also contains two methods: 'login' (which takes a 'password' argument and has a comment '// login logic') and 'sendMessage' (which takes a 'str' argument and has a comment '// sending logic'). A red box highlights the data properties, and a red arrow points to it from the word 'Data'. Another red box highlights the methods, and a red arrow points to it from the word 'Behaviour'.

```
const user = {  
  user: 'Jonas',  
  password: 'dk23s',  
  
  login (password) {  
    // login logic  
  },  
  sendMessage(str) {  
    // sending logic  
  }  
}
```

Data

Behaviour

Classes and Instances (Traditional OOP)



class is like a blueprint which can then be used to create a new object and is really an abstract plan like a blueprint.

CLASS

```
user {  
  user  
  password  
  email  
  
  login (password) {  
    // login logic  
  }  
  sendMessage(str) {  
    // sending logic  
  }  
}
```

Just a representation, NOT actual JavaScript syntax!

JavaScript does NOT support real classes like represented here.

The virtual model of the class is just a description of user data and behaviours about the user.

Now object created from the class. Like a real house created from an abstract blueprint.

INSTANCE

```
const user {  
  user: 'jonas',  
  password: 'dk23s',  
  email: 'hello@jonas.io',  
  
  login (password) {  
    // login logic  
  }  
  sendMessage(str) {  
    // sending logic  
  }  
}
```



INSTANCE

```
const user {  
  user: 'Gary',  
  password: 'dk245s',  
  email: 'hello@gary.io',  
  
  login (password) {  
    // login logic  
  }  
  sendMessage(str) {  
    // sending logic  
  }  
}
```

The virtual model of the class is used to make many instances of objects with the same type of data.

`new user 'jonas';`

`new user 'Gary';`

How do we design a class - Abstraction

ABSTRACTION

ENCAPSULATION

INHERITANCE

POLYMORPHISM

A phone



Do we really need all these low-level details?

```
Phone {  
  charge  
  volume  
  voltage  
  temperature  
  
  homeBtn() {}  
  volumeBtn() {}  
  screen() {}  
  verifyVolt() {}  
  verifyTemp() {}  
  vibrate() {}  
  soundSpeaker() {}  
  soundEar() {}  
  frontCamOn() {}  
  frontCamOff() {}  
  rearCamOn() {}  
  rearCamOff() {}  
}
```

Abstracted phone



```
Phone {  
  charge  
  volume  
  
  homeBtn() {}  
  volumeBtn() {}  
  screen() {}  
}
```

Details have been abstracted away.

Abstraction: Ignoring or hide details that **don't matter** allowing us to get an **overview** perspective of the *thing* we are implementing, instead of messing with details that don't really matter to our implementation.

How do we design a class - Encapsulation

ABSTRACTION

ENCAPSULATION

INHERITANCE

POLYMORPHISM

Again, not actually
JavaScript syntax
(the private keyword
does not exist)

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  
  }  
}
```

STILL accessible
from within the
class!

Prevents external code from accidentally manipulating internal properties/state (state = an objects data).

Allows us 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).

How do we design a class - Inheritance

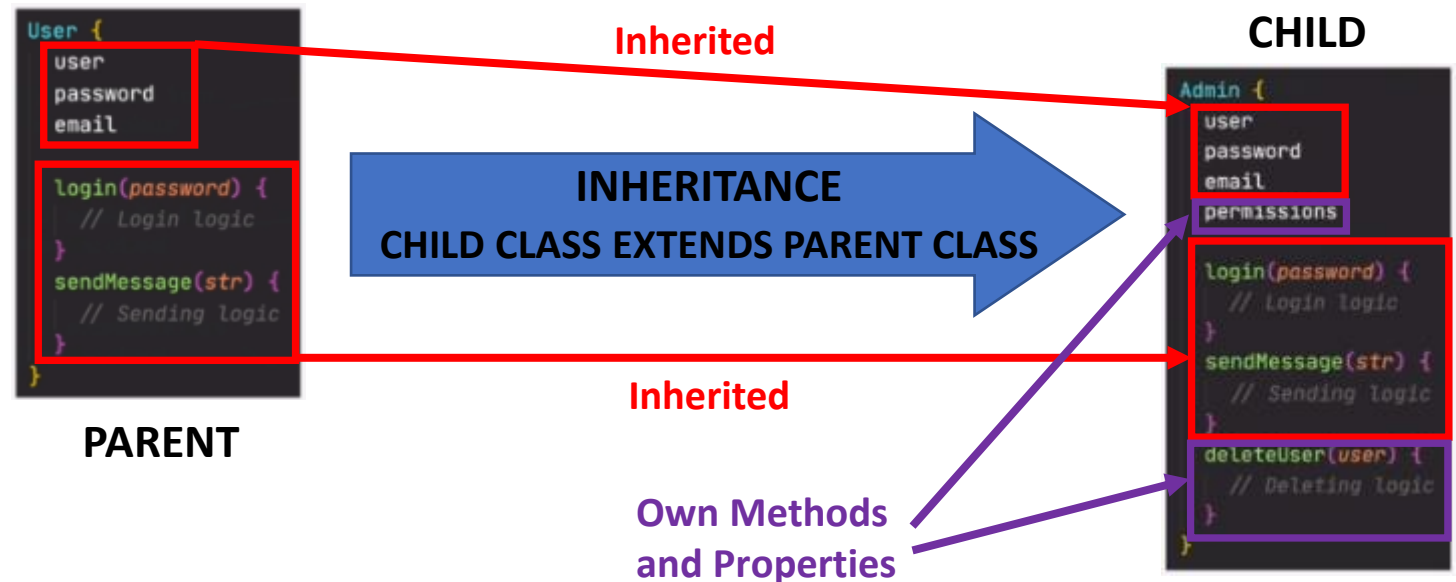
User class and admin class are closely related. the Admin class has some additional properties and methods.

ABSTRACTION

ENCAPSULATION

INHERITANCE

POLYMORPHISM



Inheritance: Making all properties and methods of a certain class **available to a child class**, forming a hierarchical structure between classes. This allows us to **reuse common logic** and to model real world relationships.

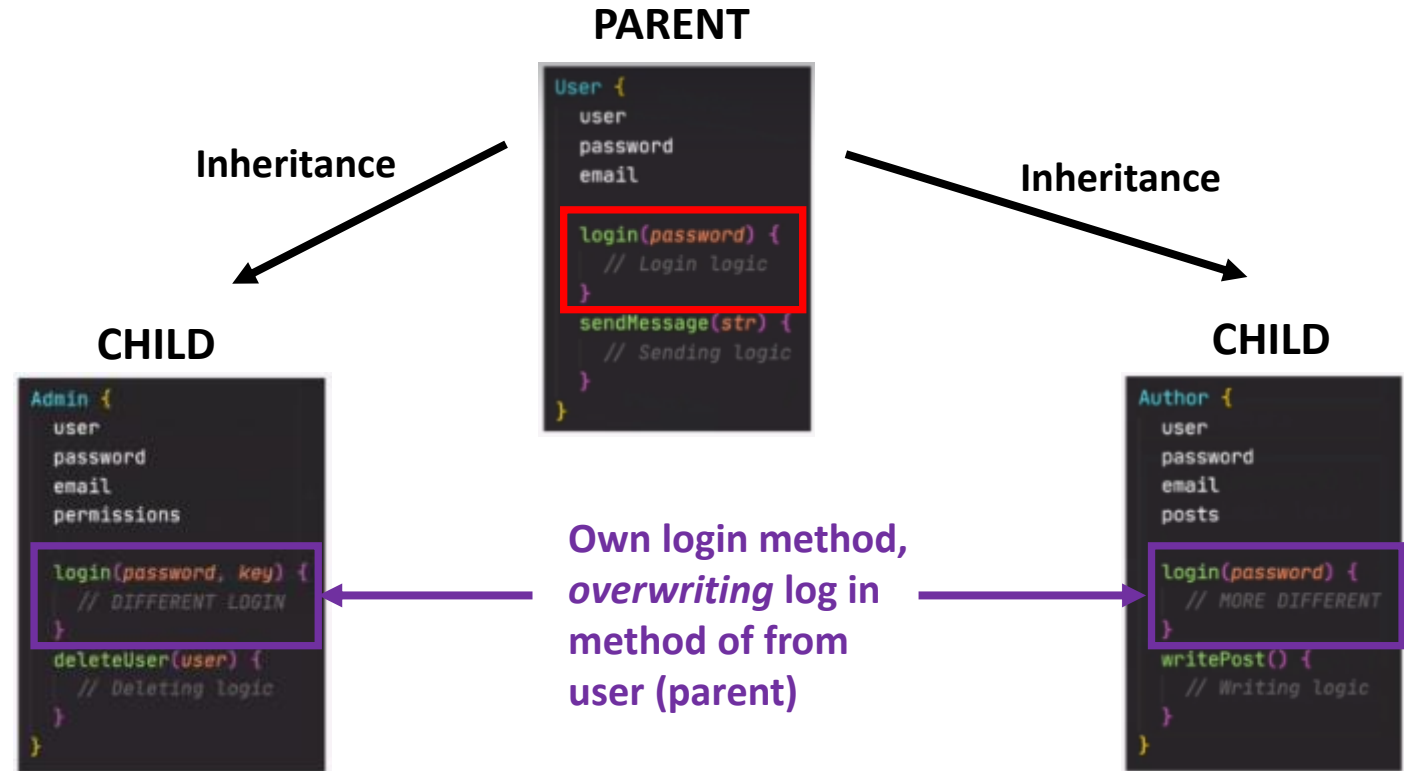
How do we design a class - Polymorphism

ABSTRACTION

ENCAPSULATION

INHERITANCE

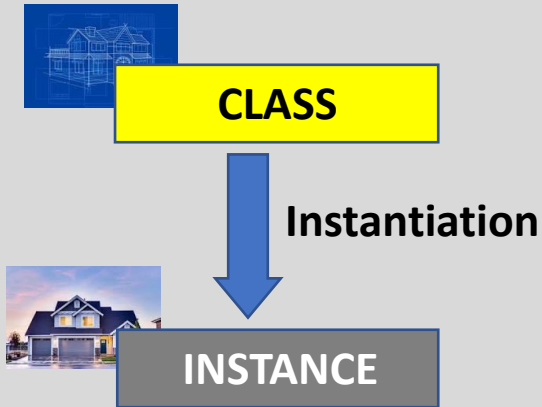
POLYMORPHISM



Polymorphism: A child's class can **overwrite** a method it inherited from a parent class (it's more complex than that!)

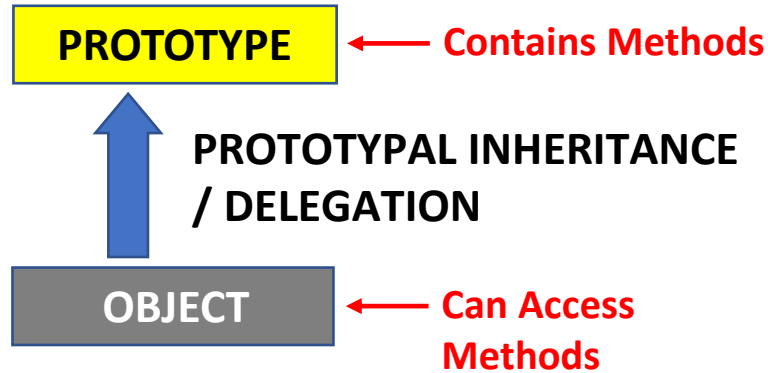
OOP in JavaScript - prototypes

"Classical OOP" : CLASSES



- Objects (instances) are **instantiated** from a class, which functions like a blueprint
- Behaviour (methods) is **copied** from class to all instances

OOP in JavaScript : PROTOTYPES



- Objects are **linked** to a prototype object.
- **Prototypal Inheritance:** The prototype contains methods (behaviour) that are **accessible to all objects linked of that prototype.**
- Behaviour 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: f ()  
  length: 0  
  constructor: f Array()  
  concat: f concat()  
  map: f map()  
}
```

3 ways of implementing Prototypal Inheritance in JavaScript

How do we actually create prototypes? And how do we link objects to prototypes? How can we create new objects, without having classes?

1

Constructor Functions

- Technique to create objects from functions
- 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 the constructor functions
- ES6 classes do NOT behave like classes in “classical OOP”.

3

`object.create()`

- The easiest and most straight forward way of linking an object to a prototype object.

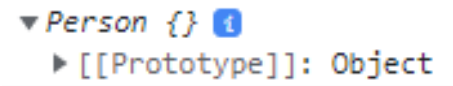
Constructor Functions and the New Operator

A constructor function can be used to build an object programmatically.

A constructor function is like any other function but the convention is that the name of the function starts with a capital letter. This is the same as existing built in constructor functions like Map or Array follow this convention.

An arrow function will not work as a function constructor because it does not have its own this keyword and we need that.

```
const Person = function (firstName, BirthYear) {  
  console.log(this);  
};
```



This constructor function is going to produce an object, in this case person so we need to specify the parameters for the values in the object.

```
new Person('Jonus', 1991);
```

The only difference between a regular function and a constructor function is that we call it using the new keyword.

Behind the scenes there have been four steps:

- 1) New {} empty object has been created, called Person.
 - 2) Function is called and in this function call the this keyword will be set to the newly created object. this = {}.
 - 3) This newly created object is linked to a prototype.
 - 4) The object that was created in the beginning is automatically returned from the constructor function. At this point the object no longer needs to be empty.
- All of this happens because of the new keyword when we call the constructor function.

```
const Person = function (firstName, BirthYear) {  
  this.firstName = firstName;  
  this.BirthYear = BirthYear;  
};
```

```
const jonas = new Person('Jonus', 1991);  
console.log(jonas);
```

```
▼ Person {firstName: 'Jonus', BirthYear: 1991} ⓘ  
  BirthYear: 1991  
  firstName: "Jonus"  
  ► [[Prototype]]: Object
```

Now within the constructor function we fill the object by setting the 'this' keyword to each of the parameters of the function. The convention is that the parameter in the brackets of the constructor function should be the same as that set on the 'this' keyword to define the property.

When we now call the constructor function the new object is created with all the values in it.

```
const matilda = new Person('Matilda', 2017);  
const jack = new Person('Jack', 1975);
```

```
console.log(matilda, jack);
```

We are creating an object, in fact three, from a constructor function so they are like instances of Person.

```
▼ Person {firstName: 'Matilda', BirthYear: 2017} ⓘ  
  BirthYear: 2017  
  firstName: "Matilda"  
  ► [[Prototype]]: Object  
▼ Person {firstName: 'Jack', BirthYear: 1975} ⓘ  
  BirthYear: 1975  
  firstName: "Jack"  
  ► [[Prototype]]: Object
```

```
console.log(jonas instanceof Person); true
```

In JavaScript we can test if Jonas is an instance of person.

```
this.firstName = firstName;  
this.BirthYear = BirthYear;
```

These are called instance properties because they will be available on any object created in the constructor function.

```
const Person = function (firstName, BirthYear) {  
  this.firstName = firstName;  
  this.BirthYear = BirthYear;  
  
  this.calcAge = function () {  
    console.log(2037 - this.BirthYear)  
  };  
};
```

We can now use the calcAge method on the jonas object even though it is not part of the jonas object but we have access to it because of prototypal inheritance.

This solves the issue of adding the calcAge method directly in the function and having a copy of it in each object. There is only one version of the prototypal method.

We can create a method within the constructor function which will work but is really bad practice because then each of the objects created by the constructor function would carry around this function and so if we were creating many objects from the constructor function then this would be very bad for performance.

```
▼ Person {firstName: 'Jonus', BirthYear: 1991} ⓘ  
  BirthYear: 1991  
  firstName: "Jonus"  
  ▼ [[Prototype]]: Object  
    ► calcAge: f ()  
    ► constructor: f (firstName, BirthYear)  
    ► [[Prototype]]: Object
```

Prototypes

- Each an every function in JavaScript automatically has a property called prototype, and this included constructor functions.
- Every object that is created by our constructor function will get access, or inherit, the properties and methods of the constructor function.

```
console.log(Person.prototype);
```

The prototype for Person is currently empty.

```
▼ {constructor: f} ⓘ  
  ► constructor: f (firstName, BirthYear)  
  ► [[Prototype]]: Object
```

```
Person.prototype.calcAge = function () {  
  console.log(2037 - this.BirthYear);  
};
```

```
▼ {constructor: f} ⓘ  
  ► calcAge: f ()  
  ► constructor: f (firstName, BirthYear)  
  ► [[Prototype]]: Object
```

Now the prototype has a method called calcAge.

```
jonas.calcAge(); 46
```

So all the person Instances have access to all the methods of the constructor function so we can call a method of the constructor and indeed we do get the result.

```
matilda.calcAge();  
jack.calcAge();
```

```
▼ Person {firstName: 'Matilda', BirthYear: 2017} ⓘ  
  BirthYear: 2017  
  firstName: "Matilda"  
  ▼ [[Prototype]]: Object  
    ► calcAge: f ()  
    ► constructor: f (firstName, BirthYear)  
    ► [[Prototype]]: Object  
▼ Person {firstName: 'Jack', BirthYear: 1975} ⓘ  
  BirthYear: 1975  
  firstName: "Jack"  
  ▼ [[Prototype]]: Object  
    ► calcAge: f ()  
    ► constructor: f (firstName, BirthYear)  
    ► [[Prototype]]: Object
```

We can also use the method on other instances of Person created by the constructor function.

Note that the method of calcAge does not get added as a property in each instance of the person Object but they do have access to the methods and properties of its prototype.


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

The prototype of matilda, jack and jonas is Person.prototype. We can look at this. The prototype of the jonas object is essentially the property of the prototype function.

```
▼ {calcAge: f, constructor: f} ⓘ  
  ► calcAge: f ()  
  ► constructor: f (firstName, BirthYear)  
  ► [[Prototype]]: Object
```

```
console.log(jonas.__proto__ === Person.prototype); true
```

So should Person.prototype be the prototype of person?

No. Person.prototype is not actually prototype of Person but it is what is going to be used as the prototype of all the objects created with the Person Constructor.

```
console.log(Person.prototype.isPrototypeOf(jonas)); true  
console.log(Person.prototype.isPrototypeOf(matilda)); true  
console.log(Person.prototype.isPrototypeOf(Person)); false
```

We can test if person.prototype is the prototype of another object which is true. i.e. Person.prototype is the prototype of jonas.

But Person.Prototype is not the prototype of Person.

Best way to think of this is that Prototype property is the prototype of linked objects.

Where does the prototype property on the jonas object actually come from?

the __proto__ property is created with the 'new' operator. It sets or links the prototype property of the empty object to the prototype property of the constructor prototype.

```
Person.prototype.species = 'Homo Sapiens';  
console.log(jonas, matilda);
```

We can also set properties on prototypes and not just methods.

species is something all of the objects will have in common. We can see the species property in the jonus object in the prototype.

```
▼ Person {firstName: 'Jonus', BirthYear: 1991} ⓘ  
  BirthYear: 1991  
  firstName: "Jonus"  
  ▼ [[Prototype]]: Object  
    ► calcAge: f ()  
    species: "Homo Sapiens"  
    ► constructor: f (firstName, BirthYear)  
    ► [[Prototype]]: Object
```

```
Person.prototype.species = 'Homo Sapiens';  
console.log(jonas.species, matilda.species);
```

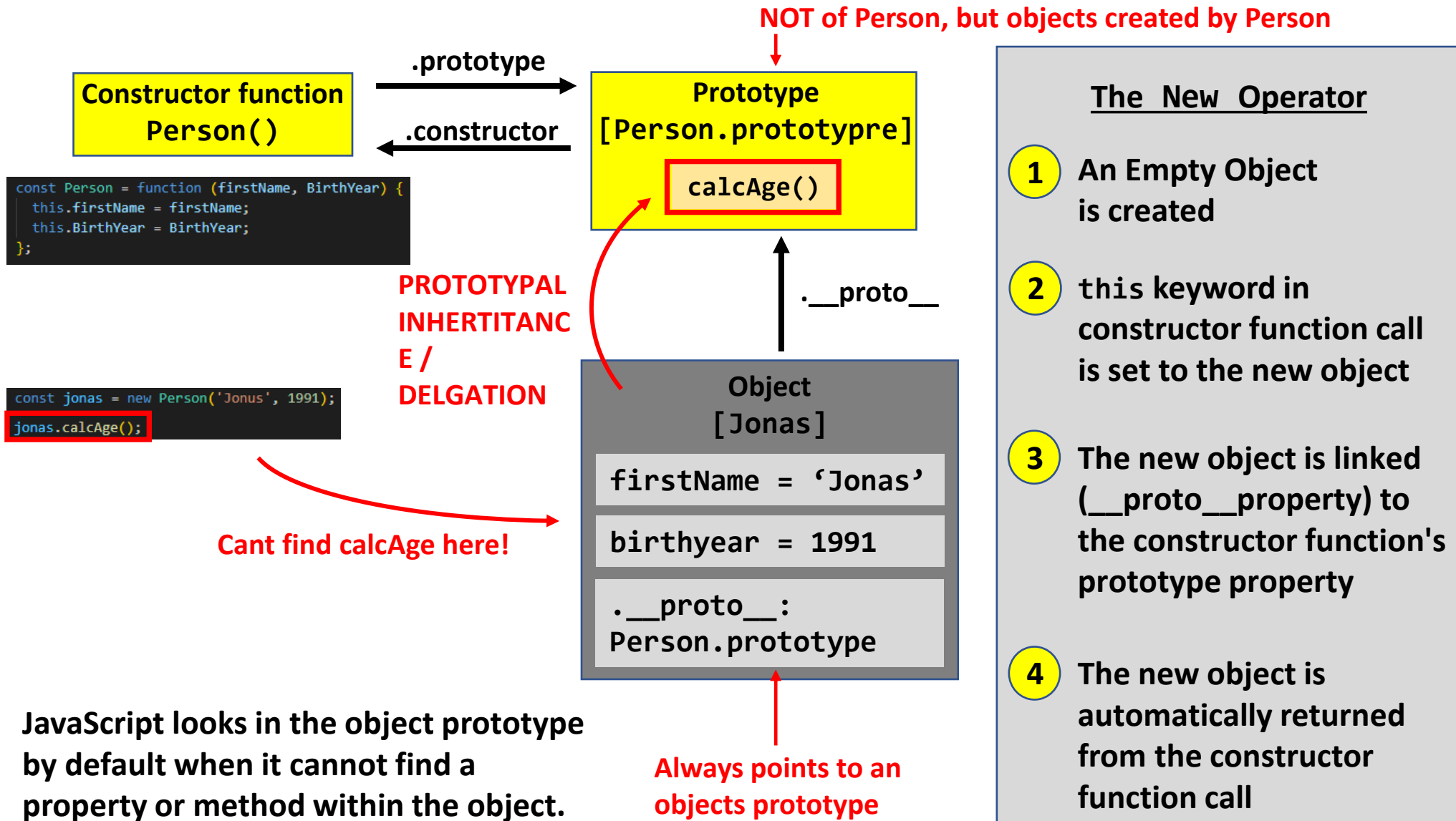
Homo Sapiens Homo Sapiens

Jonas and matilda objects have access the these properties that inherited via the prototype but they are not properties directly in the object of jonus or matilda.

```
console.log(jonas.hasOwnProperty('firstName')); true  
console.log(jonas.hasOwnProperty('species')); false
```

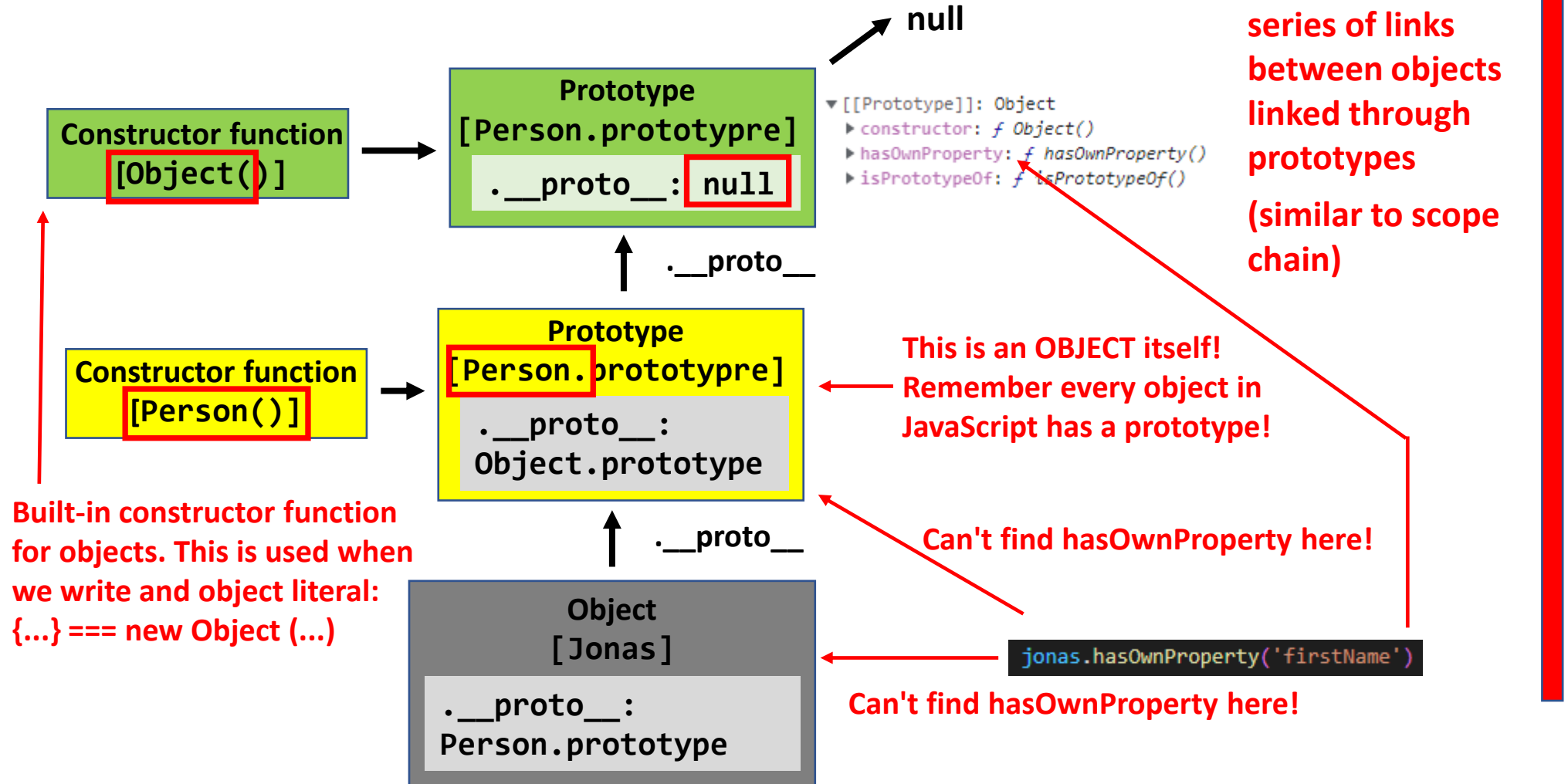
We can check if an object property is its own, i.e in the object. Note how firstName is the own property of the jonus object but species property is not.

Prototypal Inheritance and the Prototype Chain



This is how it works for Constructors and ES6 classes

- All objects in Javascript have a prototype so the prototype of Person.prototype itself is an Object therefore it will have a prototype called Object.prototype.
- Person.prototype needs to have a prototype.
- The Object.prototype does not have a prototype so its prototype is null.
- Object.prototype is at the top of the chain.



Prototypal Inheritance on Built-in Objects

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

The prototype of jonus object is the person.prototype

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

The prototype of the person.prototype is the Object.prototype

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

The prototype of the Object.prototype is null because it is at the top of the chain.

```
console.log(Person.prototype.constructor);
```

Person.prototype itself has a constructor property that points back to the person.

```
console.dir(Person.prototype.constructor);
```

We can inspect the prototype constructor and see that it points back at Person

```
script.js:44
▶ {species: 'Homo Sapiens', calcAge: f, constructor: f}

script.js:45
▼ {constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__: f, ...} ⓘ
  ▶ constructor: f Object()
  ▶ hasOwnProperty: f hasOwnProperty()
  ▶ isPrototypeOf: f isPrototypeOf()
  ▶ propertyIsEnumerable: f propertyIsEnumerable()
  ▶ toLocaleString: f toLocaleString()
  ▶ toString: f toString()
  ▶ valueOf: f valueOf()
  ▶ __defineGetter__: f __defineGetter__()
  ▶ __defineSetter__: f __defineSetter__()
  ▶ __lookupGetter__: f __lookupGetter__()
  ▶ __lookupSetter__: f __lookupSetter__()
  ▶ __proto__: (...)
  ▶ get __proto__: f __proto__()
  ▶ set __proto__: f __proto__()

null
script.js:46
```

```
f (firstName, BirthYear) {
  this.firstName = firstName;
  this.BirthYear = BirthYear;
}
```

```
▼ f Person(firstName, BirthYear) ⓘ
```

```
length: 2
name: "Person"
▶ prototype: {species: 'Homo Sapiens', calcAge: f, constru
arguments: (...)
caller: (...)
[[FunctionLocation]]: script.js:3
▶ [[Prototype]]: f ()
▶ [[Scopes]]: Scopes[2]
```

```
const arr = [3,6,4,5,7,5,9];
console.log(arr.__proto__);
```

- If we have a simple array and we look at the proto property of the array we see that it has all the in-built methods.
- Each array does not contain all of the array method but inherits them.

```
console.log(arr.__proto__ === Array.prototype); true
```

```
console.log(arr.__proto__.__proto__);
```

```
{constructor: f, __defineGetter__: f, __defineSetter__: f,
  hasOwnProperty: f, __lookupGetter__: f, ...}
```

```
script.js:51
[constructor: f, concat: f, copyWithin: f, fill: f, find:
f, ...] ④
  ▶ at: f at()
  ▶ concat: f concat()
  ▶ constructor: f Array()
  ▶ copyWithin: f copyWithin()
  ▶ entries: f entries()
  ▶ every: f every()
  ▶ fill: f fill()
  ▶ filter: f filter()
  ▶ find: f find()
  ▶ findIndex: f findIndex()
  ▶ findLast: f findLast()
  ▶ findLastIndex: f findLastIndex()
  ▶ flat: f flat()
  ▶ flatMap: f flatMap()
  ▶ forEach: f forEach()
```

The array.prototype has Object,Prototype as its prototype

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

If we check the MDN documentation for the in-built method of array.filter then it is listed as array.prototype.filter because it lives in the prototype property of the array constructor.

The prototype is a mechanism for repeating code so that all the in-built functions live in the JavaScript engine and can be inherited by any object which is of the correct type.

```
const arr = [3, 6, 5, 7, 4, 5, 7, 5, 9];
```

```
Array.prototype.unique = function () {  
  return [...new Set(this)];  
};
```

```
console.log(arr.unique());
```

```
► (6) [3, 6, 5, 7, 4, 9]
```

We can add a method to the Array.Prototype, for example to return all the unique elements in an array.

We can create a function to de-construct an array and return a new set. The this keyword will be the array on which this method is called.

Now when we call that method on the array it does indeed return an array with unique values only because it is using the prototype method we just created.

- However, extending the prototype of a built in object is generally not a good idea. It works and would not be an issue but if multiple coders are working on a project and multiple coders create the same method with a different name then it will create confusion and bugs.
- Additionally, in a future release of JavaScript there may be a built in method that does the same as what we just manually added to the prototype so this could cause bugs.

```
const h1 = document.querySelector('h1');  
console.dir(h1);
```

A html element will also have a prototype which contains built-in methods.

```
virtualKeyboardPolicy: ""  
▼ [[Prototype]]: HTMLHeadingElement  
  align: (...)  
  ► constructor: f HTMLHeadingElement()  
    Symbol(Symbol.toStringTag): "HTMLHeadingElement"  
  accessKey: (...)  
  ariaAtomic: (...)  
  ariaAutoComplete: (...)  
  ariaBusy: (...)  
  ariaChecked: (...)  
  ariaColCount: (...)
```

Coding Challenge #1

```
// 1.
const Car = function (make, speed) {
  this.make = make;
  this.speed = speed;
};

// 2.
Car.prototype.accelerate = function () {
  console.log((this.speed += 10));
};

// 3.
Car.prototype.brake = function () {
  console.log((this.speed -= 5));
};

// 4.
const bmw = new Car('BMW', 120);
const mercedes = new Car('Mercedes', 95);

console.log(bmw, mercedes);

bmw.accelerate();
bmw.accelerate();
bmw.brake();
mercedes.accelerate();
mercedes.accelerate();
mercedes.accelerate();
mercedes.brake();
```

1. Use a constructor function to implement a Car. A car has a make and a speed property. The speed property is the current speed of the car in km/h;
2. Implement an 'accelerate' method that will increase the car's speed by 10, and log the new speed to the console;
3. Implement a 'brake' method that will decrease the car's speed by 5, and log the new speed to the console;
4. Create 2 car objects and experiment with calling 'accelerate' and 'brake' multiple times on each of them.

DATA CAR 1: 'BMW' going at 120 km/h

DATA CAR 2: 'Mercedes' going at 95 km/h

The advantages of object orientated programming are that the functionality and the data are packed into objects.

Each object, BMW and Mercedes contain the data and the methods to manipulate the data.

ES6 classes

ES6 Classes allows us to do the same thing but with a nicer and more modern syntax.

```
const PersonCl = class {  
  constructor(firstName, birthYear) {  
    this.firstName = firstName;  
    this.BirthYear = birthYear;  
  }  
};
```

Method 1: Using a class expression.

```
const jessica = new PersonCl('Jessica', 1996);  
console.log(jessica);
```

```
class PersonCl {  
  constructor(firstName, birthYear) {  
    this.firstName = firstName;  
    this.BirthYear = birthYear;  
  }  
}
```

Method 2: Using a class declaration.

```
const jessica = new PersonCl('Jessica', 1996);  
console.log(jessica);
```

```

class PersonCl {
  constructor(firstName, birthYear) {
    this.firstName = firstName;
    this.BirthYear = birthYear;
  }
  calcAge() {
    console.log(2037 - this.BirthYear);
  }
}

```

We can create methods inside of the class.
 Note that the function is outside of the constructor so it does get added to the object prototype.

Methods will be added to the .prototype property

```

const jessica = new PersonCl('Jessica', 1996);
console.log(jessica);    ► PersonCl {firstName: 'Jessica', BirthYear: 1996}
jessica.calcAge();      41

console.log(jessica.__proto__ === PersonCl.prototype); true

```

- Classes are NOT hoisted
- Classes are first class citizens (we can pass them into functions and return them)
- Classes are always executed in strict mode.

Setters and Getters – regular object

Getters and Setters are functions that get and set values.

```
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);  
  
account.latest = 50;  
console.log(account.movements);
```

The get latest function within the account object is a getter because it gets the last movement from the array.

The set latest function within the account object is setter because it adds a new movement to the array. Note that setters need to have at least one parameter for what is being set.

Setters and Getters – Object Class

```
class PersonCl {  
  constructor(firstName, birthYear) {  
    this.firstName = firstName;  
    this.BirthYear = birthYear;  
  }  
  calcAge() {  
    console.log(2037 - this.BirthYear);  
  }  
  
  get age() {  
    return 2037 - this.BirthYear;  
  }  
}
```

In the object class the getter is written the same way within the class but outside of the constructor.

```
const jessica = new PersonCl('Jessica', 1996);  
console.log(jessica);  
jessica.calcAge();  
console.log(jessica.age);
```

▶ PersonCl {firstName: 'Jessica', BirthYear: 1996}
41
41

Note how we call the getter like a method of the Jessica object.

```

class PersonCl {
  constructor(fullName, birthYear) {
    this.fullName = fullName;
    this.BirthYear = birthYear;
  }

  set fullName(name) {
    console.log(name);
    if (name.includes(' ')) this.fullName = name;
    else alert(`${name} is not a full name!`);
  }
}

```

```

const jessica = new PersonCl('Jessica Davies', 1996);
console.log(jessica);

```

However this produces an error because we are trying to set a property that is already defined in the constructor.

Here we are creating a class to build an object with fullName and birthyear.

Within this class we have a setter that will check if the full name has a space in it and set the fullName or produce an alert.

Jessica Davies	script.js:135
Jessica Davies	script.js:135
Jessica Davies	script.js:135
194 Jessica Davies	script.js:135
✖ ▶ Uncaught RangeError: Maximum call stack size exceeded script.js:136 at PersonCl.set fullName [as fullName] (script.js:136:43) at PersonCl.set fullName [as fullName] (script.js:136:43) at PersonCl.set fullName [as fullName] (script.js:136:43)	

```
class PersonCl {  
  constructor(fullName, birthYear) {  
    this.fullName = fullName;  
    this.BirthYear = birthYear;  
  }  
}
```

1) Because we cannot set a property that is already defined in the constructor the convention is to call it something else by adding an underscore before.

```
  set fullName(name) {  
    console.log(name);  
    if (name.includes(' ')) this._fullName = name;  
    else alert(`${name} is not a full name!`);  
  }  
}
```

3) A common way of solving this is to then create a getter for fullName that returns _fullName. Jessica Davies

```
  get fullname() {  
    return this._fullName;  
  }  
}
```

2) However now the onject we created has property of _fullName and not fullname.

```
const jessica = new PersonCl('Jessica Davies', 1996);  
console.log(jessica); ▶ PersonCl {_fullName: 'Jessica Davies', BirthYear: 1996}
```

Static Methods

```
console.log(Array.from(document.querySelectorAll('h1')));
```

```
▼ [h1] 1  
  ▶ 0: h1  
    length: 1  
  ▶ [[Prototype]]: Array(0)
```

The from method is really a method that is attached to the array constructor and we could not use the from method on an array. It is attached to the entire array constructor and not the prototype so arrays do not inherit the from method. The from method is said to be in the array namespace.

```
console.log([1, 2, 3].from());
```

```
✖ ▶ Uncaught TypeError: [1,2,3].from is not a function  
   at script.js:175:23
```

```
console.log(Number.parseFloat(12));
```

parseFloat is another static method but is available in the Number constructor. We usually use these as helpers and are related to certain constructors.

We can add a static method to a constructor function.

```
const Person = function (firstName, BirthYear) {  
  this.firstName = firstName;  
  this.BirthYear = BirthYear;  
};
```

```
const jonas = new Person('Jonus', 1991);  
console.log(jonas);
```

```
Person.hey = function () {  
  console.log(this);  
  console.log('Hey there');  
};  
Person.hey(),
```

All we have to do is write *Person.name of static method* and call it. However the static method is not inherited because it is not inside the constructor.

This static method is not inherited so we cannot call `jonas.hey()` because it is not in the jonas Object. (`console.log(this)`).

```
f (firstName, BirthYear) {  
  this.firstName = firstName;  
  this.BirthYear = BirthYear;  
}
```

```
jonus.hey();
```

```
✖ ▶ Uncaught ReferenceError:  
   jonus is not defined  
   at script.js:192:1
```

We can add a static method to a class.

```
class PersonCl {  
  constructor(firstName, birthYear) {  
    this.firstName = firstName;  
    this.BirthYear = birthYear;  
  }  
  
  static hey() {  
    console.log('Hey there');  
    console.log(this);  
  }  
}  
  
const jessica = new PersonCl('Jessica', 1996);  
console.log(jessica);  
  
PersonCl.hey();
```

This time the **this** keyword of the static method points to the entire class.

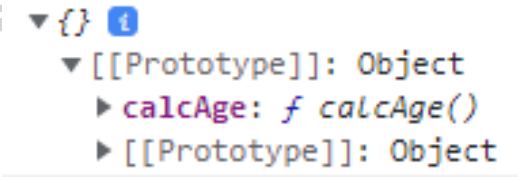
```
class PersonCl {  
  constructor(firstName, birthYear) {  
    this.firstName = firstName;  
    this.BirthYear = birthYear;  
  }  
  
  static hey() {  
    console.log('Hey there');  
    console.log(this);  
  }  
}
```

Note that static methods are not available on the instance but are sometimes helpful as helper functions.

Object Create

- With `Object.create` there is still the idea of prototypal inheritance or delegation. However there are no prototype properties involved and no constructor functions and no `new` operator.
- We can use `Object.create` to manually set the prototype to any object.

```
const PersonProto = {  
  calcAge() {  
    console.log(2037 - this.birthYear);  
  },  
};  
  
const steven = Object.create(PersonProto);  
console.log(steven);
```



We can create a person prototype because this will be an object prototype. Like before we add a method for the methods that we want the person to inherit.

We use `Object.Create` to create a new object and pass in the object prototype that we want the person to inherit.

right now steven is an empty object but it is linked to the prototype with the properties we want it to inherit like `calcAge`.

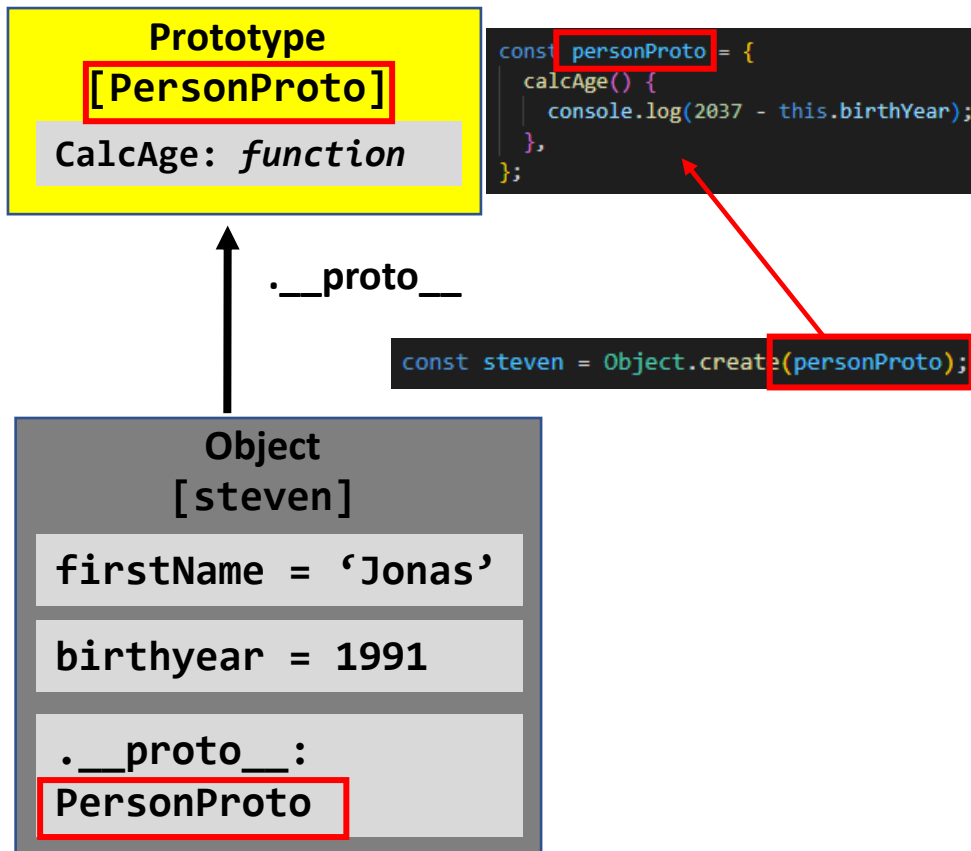
```
steven.name = 'Steven';  
steven.birthYear = 2002;  
steven.calcAge();
```

Now we can pass in values and properties to the steven object and also call methods from the object prototype.

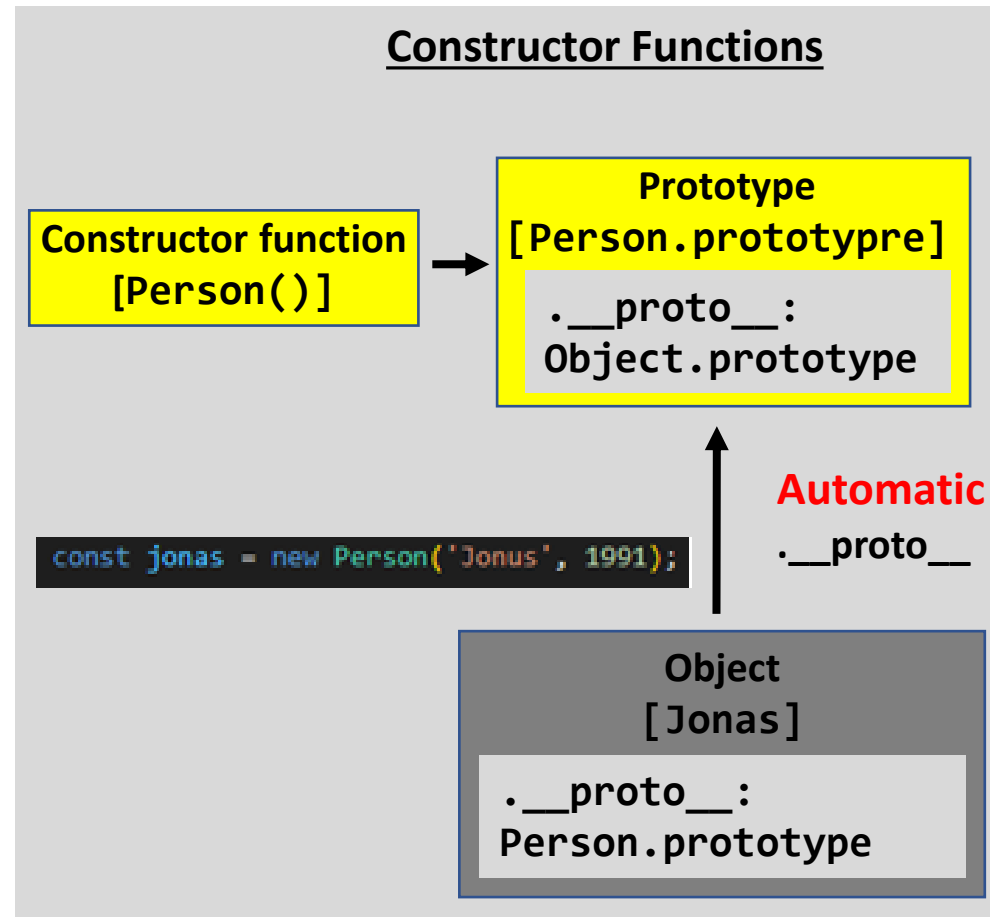
```
console.log(steven.__proto__ === PersonProto);
```

- With the constructor function when we use the new operator it automatically sets the prototype of the instances to the constructors prototype property.
- With `object.create` we can set the prototype of objects to any objects that we want. In this case we have set the `steven` object prototype to the `personProto` object.
- Now looking up properties or methods in the prototype chain is the same as a function constructor or class. The big difference is that we did not need a constructor function and also no prototype property at all to achieve the same thing.

Object.Create



Constructor Functions



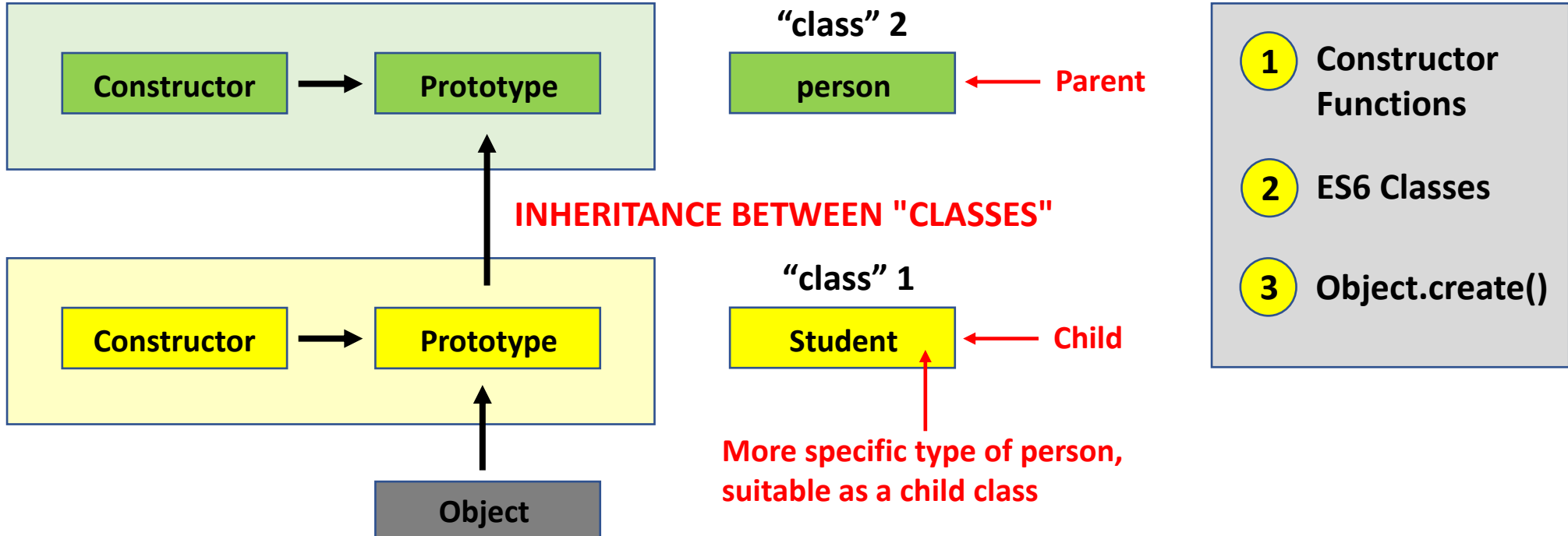
```
const PersonProto = {  
  calcAge() {  
    console.log(2037 - this.birthYear);  
  },  
  
  init(firstName, birthYear) {  
    this.firstName = firstName;  
    this.birthYear = birthYear;  
  },  
};
```

Within the PersonProto object we can add another method that adds the properties of firstName and birthYear.

We can call methods for the sarah object from the prototype.

```
const sarah = Object.create(PersonProto);  
sarah.init('Sarah', 1979);  
sarah.calcAge();
```

Inheritance between Classes: Constructor Functions



- We will have a parent class of people with a child class of student which will inherit from the parent class. Student is a sub type of a person but more specific.
- The student can use generic person methods and also have its own specific methods and we will learn how to set up the prototype chain.
- We will do this for Constructor function, ES6 classes and Object.create.

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

We start off with a Person “class” built using the constructor. It has the calcAge method in the prototype.

```
Person.prototype.calcAge = function () {  
  console.log(2037 - this.birthYear);  
};
```

```
const Student = function (firstName, birthYear, course) {  
  this.firstName = firstName;  
  this.birthYear = birthYear;  
  this.course = course;  
};
```

Here is Student “class” built using the constructor. It has the introduce method in the 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();
```

We create ‘new’ student mike and call the introduce method from the Student class.

But we have duplicate code. firstName and birthYear are the same in both constructor functions. This could be an issue in the future if the constructor changes for Person because we have to change it also in Student.

```
const Student = function (firstName, birthYear, course) {
  Person(firstName, birthYear);
  // this.firstName = firstName;
  // this.birthYear = birthYear;
  this.course = course;
};
```

```
✖ ▶ Uncaught TypeError: Cannot set properties of
undefined (setting 'firstName')
    at Person (script.js:308:18)
    at new Student (script.js:317:3)
    at script.js:327:14
```

So we want to call the Person and pass in firstName and birthyear.

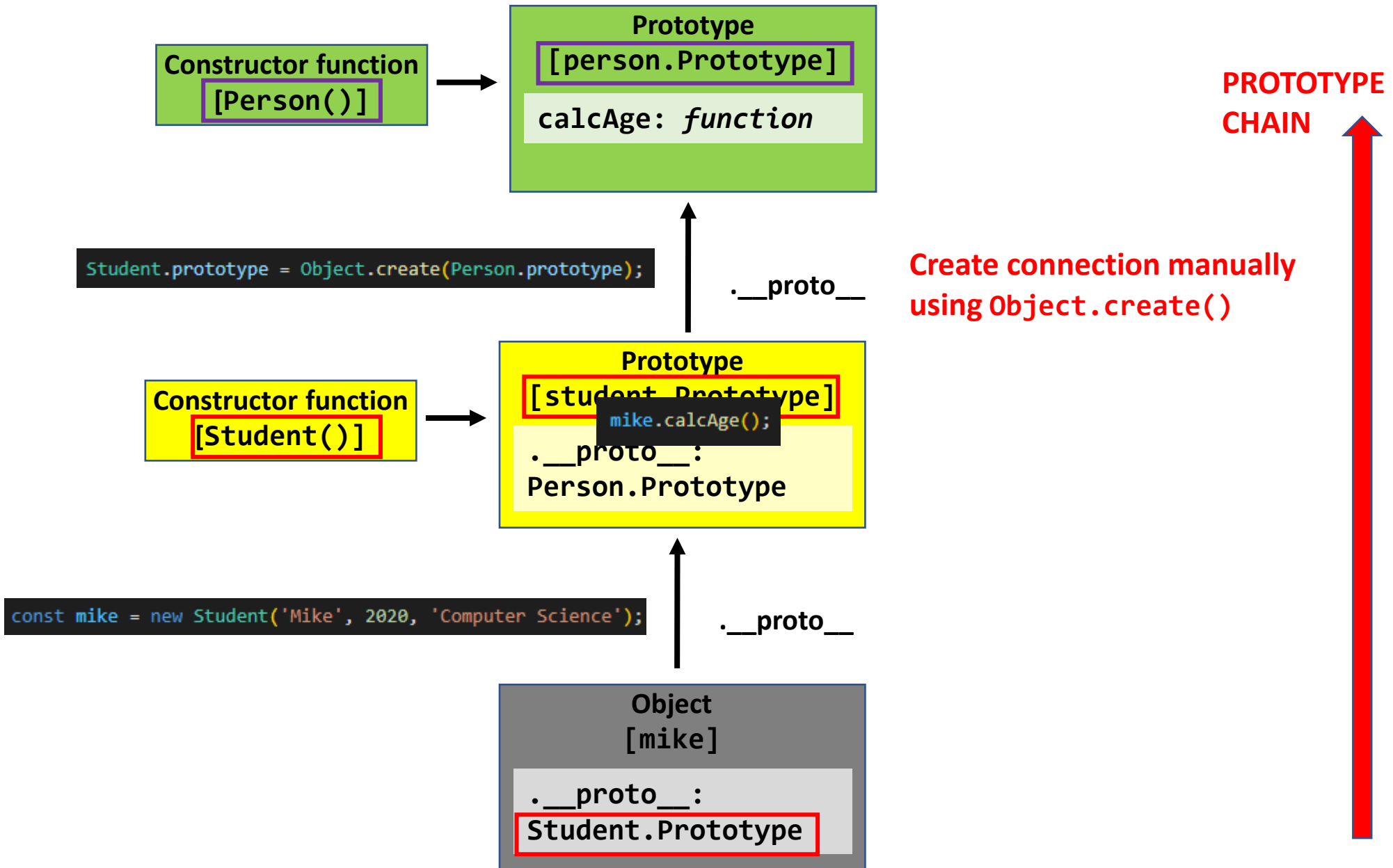
This has not worked because we are calling Person as a regular function call and not as a new. Remember that in a regular function call the this keyword is set to undefined.

```
const Student = function (firstName, birthYear, course) {
  Person.call(this, firstName, birthYear);
  this.course = course;
};
```

My name is Mike and I study Computer Science

We can use the method of call on the this keyword so now the this keyword is set to person.

We need to set the parent class of Person as the object.prototype for the child class of Student. This needs to be done manually. See following slide.



```

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;
};

```

```

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();  My name is Mike and I study Computer Science
mike.calcAge();    17

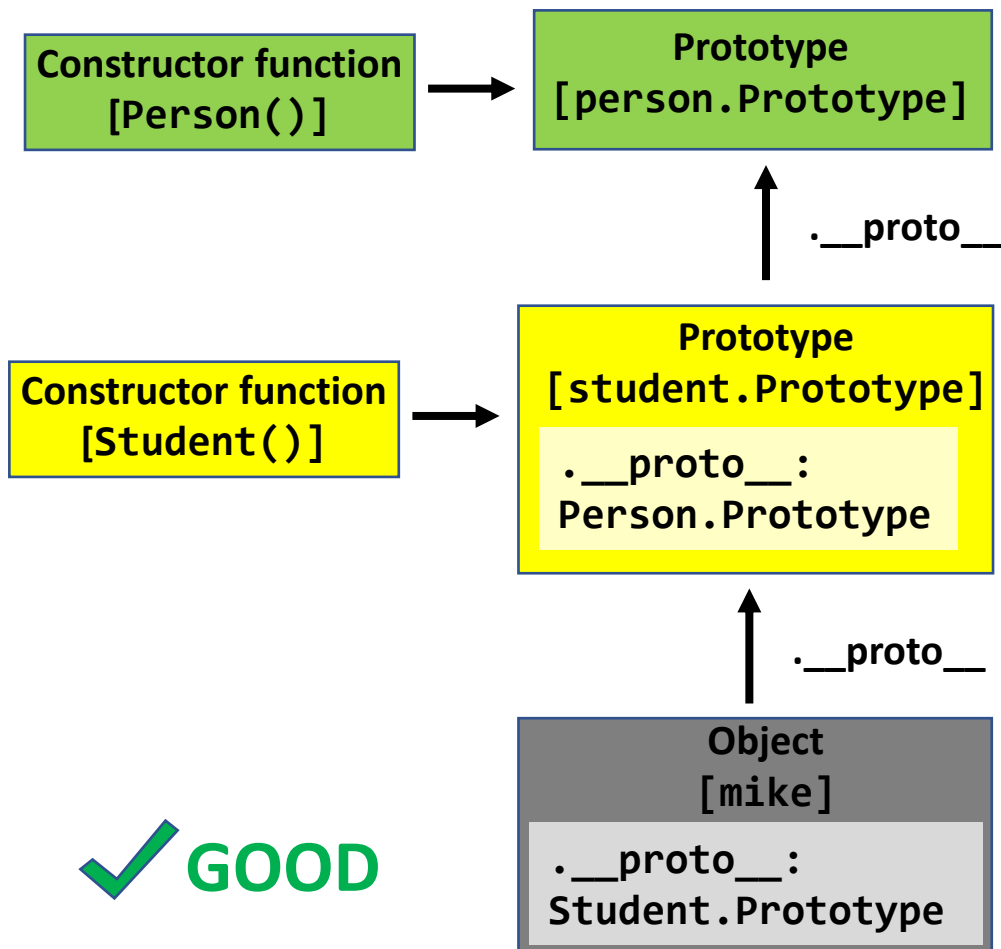
```

The `Object.create` needs to go here immediately after declaring `Student` because it creates an empty object. It must be placed before any prototype methods because if it is after, then the empty object will overwrite the prototype methods.

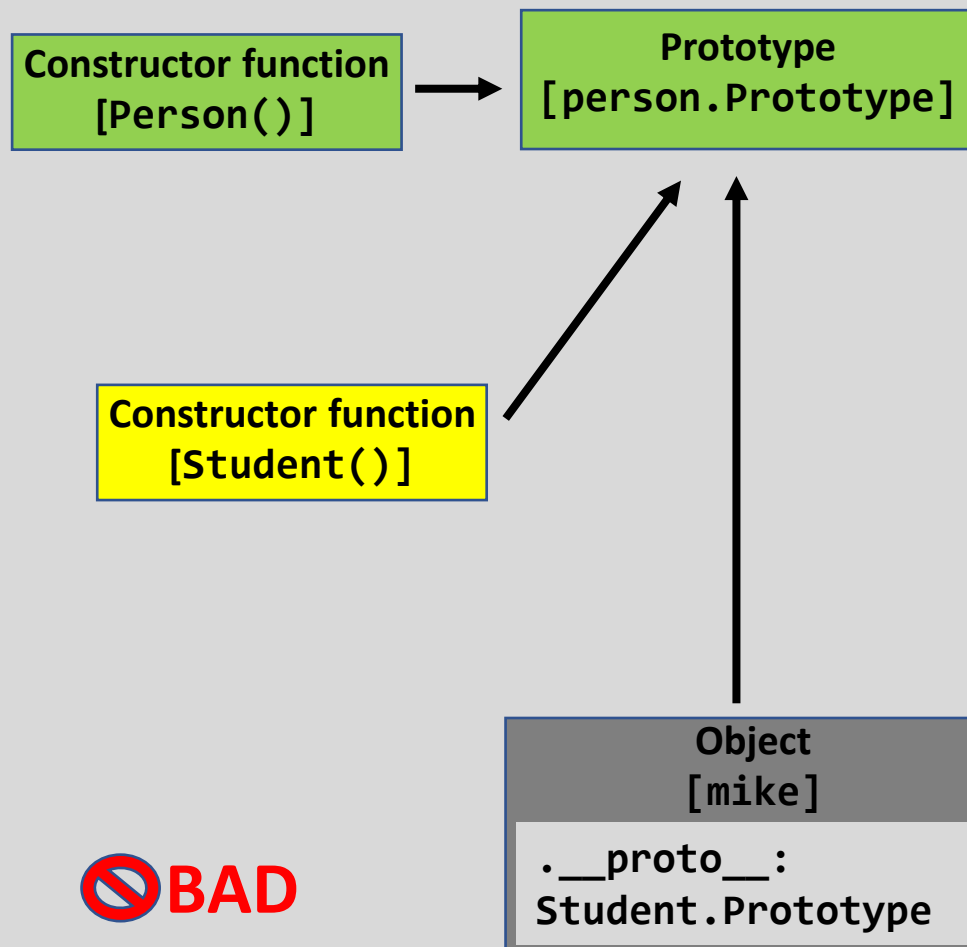
Now the `mike` instance can inherit from the parent of `Person`. In this case we can call the `calcAge` method from the parent.

- Note: we must use `Object.create` to generate the prototype chain that we need.
- if we use `Student.prototype = Person.prototype` then what we are saying is that `Student.prototype` is exactly the same thing as `Person.prototype` and we do not inherit.

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

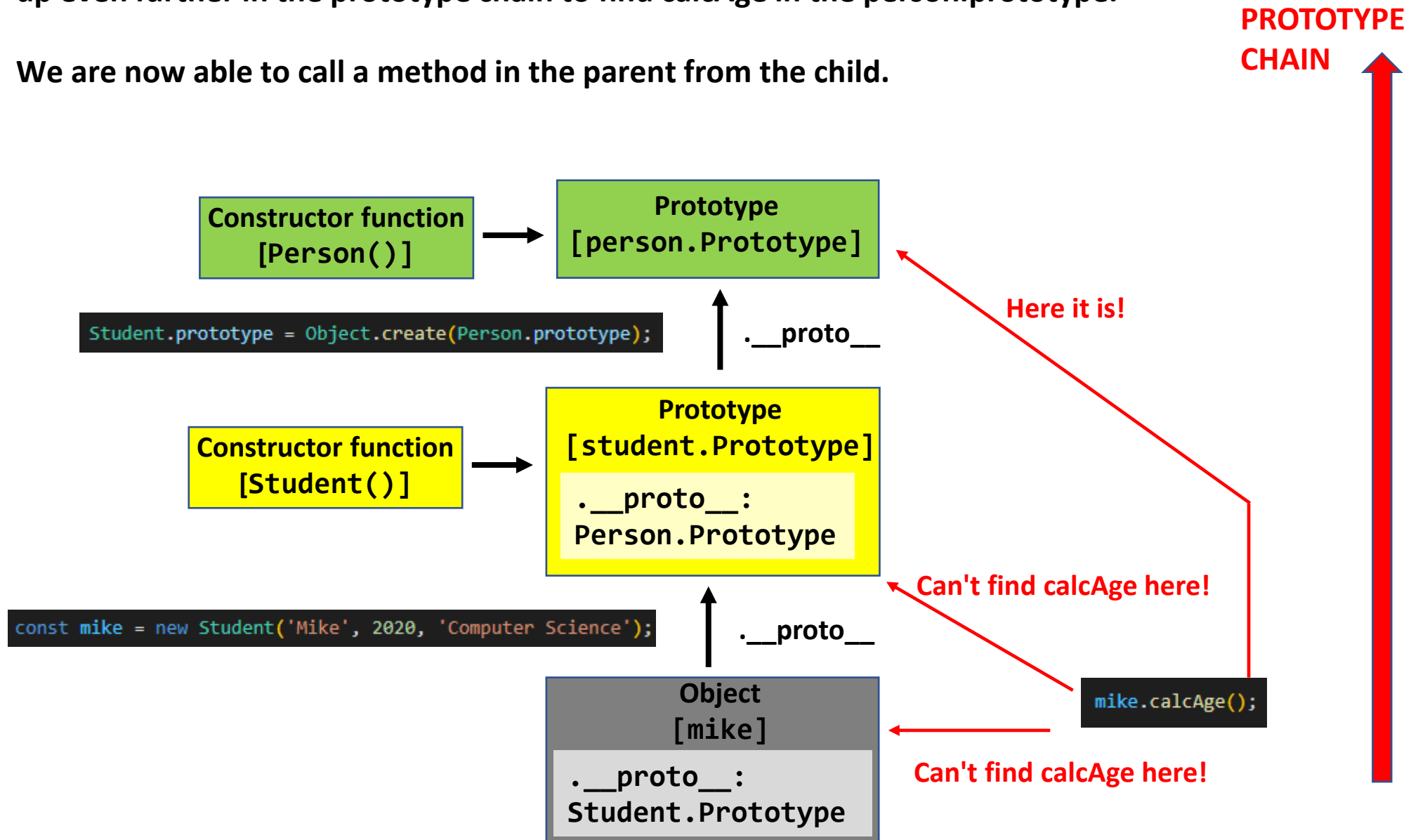


```
Student.prototype = Person.prototype;
```



What is happening here is that `mike.calcAge()` is effectively doing a property or method lookup. The `calcAge` method is not on the `mike` object or the `student.prototype` so javascript is looking up even further in the prototype chain to find `calcAge` in the `person.prototype`.

We are now able to call a method in the parent from the child.



```
console.log(mike.__proto__);
```

If we look at the prototype for the mike object we can see that is person and has the introduce method. Note that is actually student

```
console.log(mike.__proto__.__proto__);
```

If we look at the prototype for the student object we can see that is person and has the calcAge method.

```
console.dir(Student.prototype.constructor);
```

```
▼ f Student(firstName, birthYear, course) ⓘ  
  length: 3  
  name: "Student"  
  ▶ prototype: Person {introduce: f, constructor: f}  
    arguments: (...)  
    caller: (...)  
    [[FunctionLocation]]: script.js:316  
  ▶ [[Prototype]]: f ()  
  ▶ [[Scopes]]: Scopes[2]
```

```
console.log(mike instanceof Student); true  
console.log(mike instanceof Person); true  
console.log(mike instanceof Object); true
```

```
▼ Person {introduce: f} ⓘ  
  ▶ introduce: f ()  
  ▶ [[Prototype]]: Object  
  
▼ {calcAge: f, constructor: f} ⓘ  
  ▶ calcAge: f ()  
  ▶ constructor: f (firstName, birthYear)  
  ▶ [[Prototype]]: Object
```

```
▼ f Person(firstName, birthYear) ⓘ  
  length: 2  
  name: "Person"  
  ▶ prototype: {calcAge: f, constructor: f}  
    arguments: (...)  
    caller: (...)  
    [[FunctionLocation]]: script.js:307  
  ▶ [[Prototype]]: f ()  
  ▶ [[Scopes]]: Scopes[2]
```

We can fix the prototype constructor for Student by manually setting it to Student.

```
Student.prototype.constructor = Student;  
console.dir(Student.prototype.constructor);
```

Mike is an instance of Student and also an Instance of Person and also an instance of Object so the prototype chain is working.

Inheritance between Classes: ES6 Classes

```
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 🙌');
  }
}
```

The starting point is the PersonCl class created previously.

```
class StudentCl extends PersonCl {
  constructor(fullName, birthYear, course) {
    // Always needs to happen first
    super(fullName, birthYear);
    this.course = course;
  }
}
```

We need two keywords to make Student class function as a child class of Person.

extend. We specify that StudentCl extends the parent class of PersonCl.

super. We call the parent class with the keyword of super and specify the properties we want to inherit.

```
  introduce() {
    console.log(`My name is ${this.fullName} and I study
    ${this.course}`);
  }
}
```

We can create a new instance of student class using the new keyword and passing in the parameters.

```
const martha = new StudentCl('Martha Jones', 2012, 'Computer
Science');
```

Now the object martha (an instance of StudentCl) can access methods in the object and parent object

```
martha.introduce(); My name is Martha Jones and I study Computer Science
martha.calcAge(); 25
```

Note that the super keyword needs to happen first directly after the constructor because it is what which brings in the 'this' keyword.

```
class StudentCl extends PersonCl {
  // constructor(fullName, birthYear, course) {
  // Always needs to happen first
  // super(fullName, birthYear);
  // this.course = course;
}
```

```
> martha
```

```
< StudentCl {_fullName: 'Martha Jones', birthYear: 2012}
```

```
const martha = new StudentCl('Martha Jones', 2012);
```

The constructor is only need if the child class has additional properties from the parent class. If it just extends only the properties of the parent class then we do not even need the super keyword.

```
console.log(martha);
```

```
StudentCl {_fullName: 'Martha Jones', birthYear: 2012, course: 'Computer Science'}
  birthYear: 2012
  course: "Computer Science"
  _fullName: "Martha Jones"
  age: (...)
  fullName: (...)
  [[Prototype]]: PersonCl
    constructor: class StudentCl
      length: 3
      name: "StudentCl"
      prototype: PersonCl {constructor: f, arguments: (...), caller: (...), [[FunctionLocation]]: script.js:454}
      [[Prototype]]: class PersonCl
      [[Scopes]]: Scopes[2]
      introduce: f introduce()
      age: (...)
      fullName: (...)
  [[Prototype]]: Object
    age: (...)
    calcAge: f calcAge()
    constructor: class PersonCl
      fullName: (...)
    greet: f greet()
    get age: f age()
    get fullName: f fullName()
    set fullName: f fullName(name)
```

We can see that the PersonCl prototype inherits all the methods from the parent personCl.

```

class StudentCl extends PersonCl {
  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.`);
  }
}

```

Now we have a calcAge method in the StudentCl class. What will happen?

Well if we remember the theory about the Inheritance chain then it will first look in the object of Martha for calcAge and not find it. It will next go up one level in the inheritance chain to the martha.__proto__ where it will find calcAge. It does not need to go up to the personCl.prototype to find calcAge.

CalcAge in PersonCl.prototype overwrites the PersonCl.prototype and is said to be shadowing it.

Inheritance between Classes: Object.create

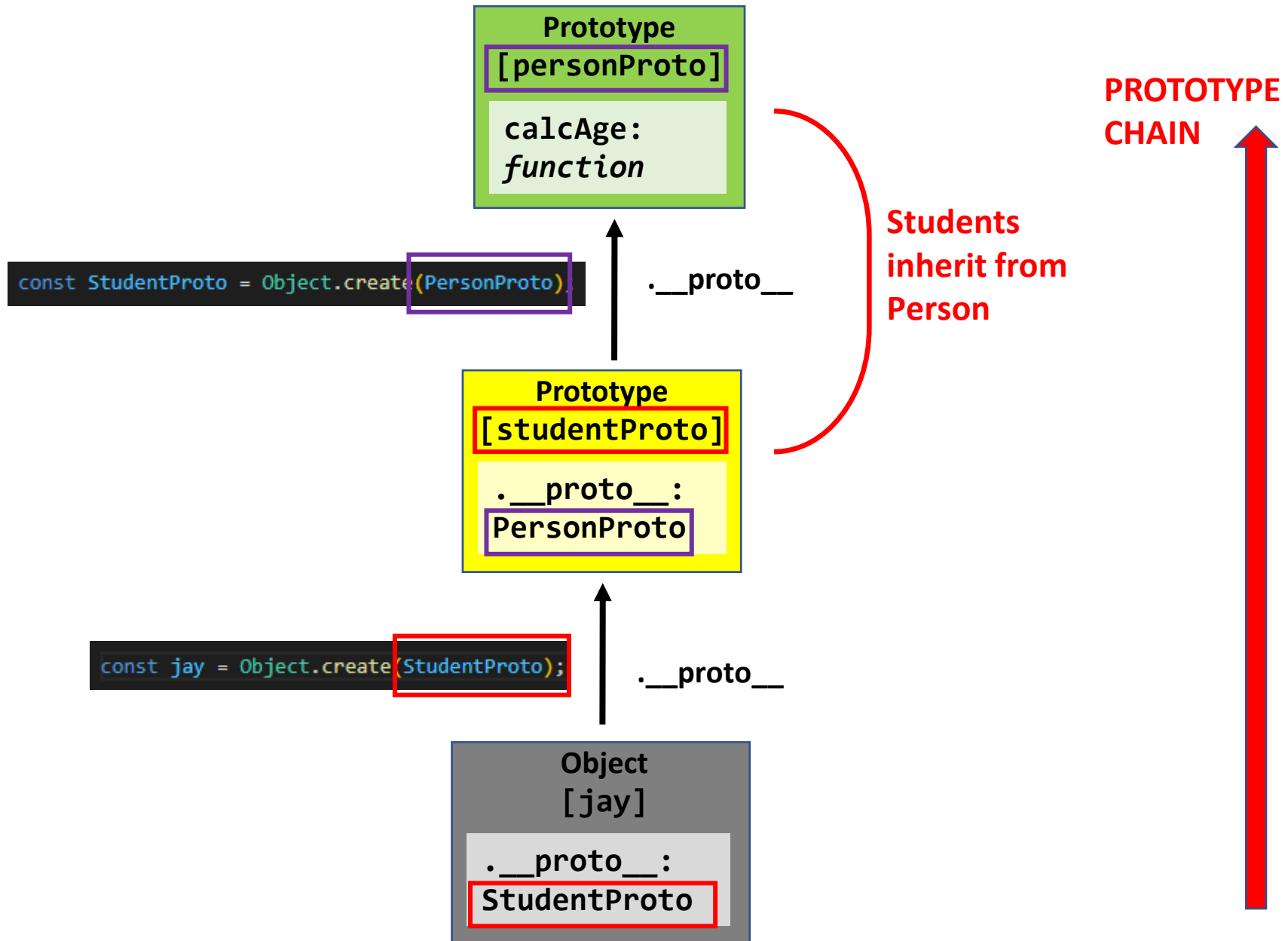
```
const PersonProto = {  
  calcAge() {  
    console.log(2037 - this.birthYear);  
  },  
  
  init(firstName, birthYear) {  
    this.firstName = firstName;  
    this.birthYear = birthYear;  
  },  
};
```

PersonProto is the parent class

```
const steven = Object.create(PersonProto);  
  
const StudentProto = Object.create(PersonProto);  
const jay = Object.create(StudentProto);
```

Now the child object of StudentProto has parent set in the Object.create of PersonProto.

Object Jay has the prototype set to StudentProto and so is an instance of student.



```
const PersonProto = {
  calcAge() {
    console.log(2037 - this.birthYear);
  },

  init(firstName, birthYear) {
    this.firstName = firstName;
    this.birthYear = birthYear;
  },
};
```

As before we can call the init method in PersonProto from the StudentProto so that we do not have to manually define properties from PersonProto.

```
const StudentProto = Object.create(PersonProto);
StudentProto.init = function (firstName, birthYear, course) {
  PersonProto.init.call(this, firstName, birthYear);
  this.course = course;
};
```

The call is used to set the this keyword from the parent to the this keyword of the child.

```
StudentProto.introduce = function () {
  console.log(`My name is ${this.firstName} and I study ${this.course}`);
};
```

We set the 'this' keyword to the course because it is unique to the child prototype.

```
const jay = Object.create(StudentProto);
jay.init('jay', 2010, 'computer Science');
jay.introduce();
```

My name is jay and I study
computer Science

Any instance of the child has access to all methods and properties in the prototype chain.

Another class Example

We can define the account from the Bankist app as a class.

```
class Account {  
  constructor(owner, currency, pin) {  
    this.owner = owner;  
    this.curency = currency;  
    this.pin = pin;  
    this.movements = [];  
    this.locale = navigator.language;  
  
    console.log(`Thanks for opening an account ${owner}`);  
  }  
  
  deposit(val) {  
    this.movements.push(val);  
  }  
  
  withdraw(val) {  
    this.deposit(-val);  
  }  
}
```

```
const acc1 = new Account('jonas', 'EUR', 1111, []);  
console.log(acc1);
```

```
acc1.deposit(250);  
acc1.withdraw(140);
```

In the constructor we pass values of owner, currency, pin.
we can build the values in the constructor with the this keyword.

Note how in the constructor we can build values without having to pass them in. i.e. movements is an empty array to start with but we do not need to pass in an empty array.

locale does not need to be passed in but can be calculated from the browser.

Using the new keyword we can create a new account object with the three parameters.

Deposits and withdrawals can be extracted away to be methods that interact with the properties of the class.

The deposit method just pushes a positive deposit value into the movements array.

Withdraw can interact with the deposit method and subtract a value from the array push of the deposit method.
The deposit and withdraw methods are the public interface of our account object.

```
console.log(acc1.pin);
```

Properties of the object can be accessed from outside which poses a risk from unintentional or malicious manipulation.

Now there are two additional methods to handle loans.

The `aproveLoan` method simply returns true but it could check if the account has a total value (from the movements array) that is greater than the loan and return true if so.

The request loan can call the approve loan method before executing the deposit value into the movements array.

The `aproveLoan` method should be private and not be possible to call it from outside of the method which is why this needs data encapsulation – see next lesson.

```
aproveLoan(val) {  
  return true;  
}
```

```
requestLoan(val) {  
  if (this.aproveLoan(val)) {  
    this.deposit(val);  
    console.log('Loan aproved');  
  }  
}
```

```
acc1.requestLoan(1000);  
acc1.aproveLoan(1000);
```

```
Thanks for opening an account jonas      script.js:517  
                                          script.js:541  
▼ Account {owner: 'jonas', currency: 'EUR', pin: 1111, movemen  
  ts: Array(0), locale: 'en-US'} ⓘ  
  currency: "EUR"  
  locale: "en-US"  
  ► movements: (3) [250, -140, 1000]  
    owner: "jonas"  
    pin: 1111  
    ► [[Prototype]]: Object  
1111                                     script.js:546  
Loan aproved                           script.js:535
```

Encapsulation: Protected Properties and Methods

We implemented a new class which showed the need for data protection and privacy. Encapsulation means to keep some properties and methods as private within the class whereas some are public APIs that can be accessed from outside of the class. JavaScript does not truly support private class fields yet but there is a proposal to implement it in future versions. In this lecture we will fake data privacy by a coding convention.

```
class Account {  
  constructor(owner, currency, pin) {  
    this.owner = owner;  
    this.curency = currency;  
    this.pin = pin;  
    this._movements = [];  
    this.locale = navigator.language;  
  
    console.log(`THanks for opening an account ${owner}`);  
  }  
  
  deposit(val) {  
    this._movements.push(val);  
  }  
  
  acc1._movements.push(250);  
  acc1._movements.push(-140);
```

The property of movements has a underscore before it. This is a coding convention which does not make the movements property truly private but rather it is said to be protected.

We can still access the `_movements` property to manipulate the data in the array when we should be using the public API of `requestLoan`, `deposit` or `withdraw`.

This is to show other coders on the team that the movements property should not be accesed from outside of the object, it does not stop anyone from coding that though. Just a convention to mark it as protected.

```

class Account {
  constructor(owner, currency, pin) {
    this.owner = owner;
    this.curency = currency;
    this._pin = pin;
    this._movements = [];
    this.locale = navigator.language;

    console.log(`Thanks for opening an
account ${owner}`);
  }

  getMovements() {
    return this._movements;
  }

  deposit(val) {
    this._movements.push(val);
  }
}

```

Of course this does not make any of the properties actually private but assumes that other coders working on the project will not code directly to access properties marked with an underscore and assumes that every coder knows the convention.

Setting the underscore is just a visual marker so that if any one tries to access the movements from outside the object then they will have to write the underscore or use a getter.

The pin could also be another property that protected by marking it with an underscore.

Aproveloan could also be marked as protected.

```

    _aproveLoan(val) {
      return true;
    }

    requestLoan(val) {
      if (this._aproveLoan(val)) {
        this.deposit(val);
        console.log('Loan approved');
      }
    }
  }
}

```

Encapsulation: Private class fields and Methods

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Private_class_fields

private class fields are a new feature in JavaScript. The naming convention comes from other programming languages where properties of classes are called fields.

There can be said to be public fields, private fields, public methods, private methods. AN object made from these can said to be an instance public field etc..

```
class Account {  
  // public fields (instances)  
  locale = navigator.language;  
  
  // private fields  
  #movements = [];
```

```
  constructor(owner, currency, pin) {  
    this.owner = owner;  
    this.curency = currency;  
    // protected property  
    this._pin = pin;
```

```
console.log(acc1.#movements);  
console.log(acc1.locale);
```

public fields are declared outside of the constructor and can be accessed from outside of the method.

private fields are now also declared from outside of the constructor but marked with a hash in front of the name.

If I try and access a private field from outside of the class I get an error message. However if I try a public field then it works.

✖ Uncaught SyntaxError: Private field '#movements' must be declared in an enclosing class (at [script.js:563:16](#))

[en-US](#)

```
acc1.#movements.push(250);
```

✖ Uncaught SyntaxError: Private field '#movements' must be declared in an enclosing class (at [script.js:563:5](#))

I cannot directly manipulate the private fields of the object. But I can interact with the public methods.

```
acc1.deposit(250);  
acc1.withdraw(140);  
acc1.requestLoan(1000);
```

I can still get the movements from the object but I have to interact with the public `getMovements` method to do it.

```
getMovements() {  
  return this.#movements;  
}
```

[script.js:550](#)
▼ Account {locale: 'en-US', owner: 'jonas', currency: 'EUR', _pin: 1111, #movements: Array(0)} ⓘ
 currency: "EUR"
 locale: "en-US"
 owner: "jonas"
 _pin: 1111
 ▼ #movements: Array(3)
 0: 250
 1: -140
 2: 1000
 length: 3
 ► [[Prototype]]: Array(0)
 ► [[Prototype]]: Object

```
console.log(acc1.getMovements());  
► (3) [250, -140, 1000]
```



```

class Account {
  // public fields (instances)
  locale = navigator.language;

  // private fields
  #movements = [];
  #pin;

  constructor(owner, currency, pin) {
    this.owner = owner;
    this.currency = currency;
    this.#pin = pin;
  }

```

```

console.log(acc1.#pin);

```

```

✖ Uncaught SyntaxError: Private field '#pin' must be
  declared in an enclosing class (at script.js:561:17)

```

Now we want to set the pin to private which has to be outside of the constructor yet we are passing the pin value into the constructor. The means of achieving this is to first define #pin as an empty private field then inside the constructor set the passed in pin to be equal to this.#pin.

```

script.js:550
Account {locale: 'en-US', owner: 'jonas', currency: 'EUR', #movements: A
  rray(0), #pin: 1111}
  currency: "EUR"
  locale: "en-US"
  owner: "jonas"
  ▶ #movements: Array(3)
  #pin: 1111
  ▶ [[Prototype]]: Object

```

#pin is truly private.

```
// public methods
getMovements() {
  return this.#movements;
}

deposit(val) {
  this.#movements.push(val);
}

withdraw(val) {
  this.deposit(-val);
}

requestLoan(val) {
  if (this.#approveLoan(val)) {
    this.deposit(val);
    console.log('Loan approved');
  }
}

// private methods
#approveLoan(val) {
  return true;
}
```

Public and private methods follow the same syntax as public and private fields.

`getMovements`, `deposit`, `withdraw` and `requestLoan` are public methods that can be accessed from outside.

`#approveLoan` is only accessible from inside the object.

```
acc1.#approveLoan(1000);
```

✖ Uncaught SyntaxError: Private field '#approveLoan' must be declared in an enclosing class (at [script.js:561:5](#))

```
Account {locale: 'en-US', owner: 'jonas', #approveLoan: f, #movements:
  Array(0), #pin: 1111, ...}
  currency: "EUR"
  locale: "en-US"
  owner: "jonas"
  #approveLoan: f #approveLoan(val)
  #movements: Array(3)
  #pin: 1111
  [[Prototype]]: Object
    constructor: class Account
    deposit: f deposit(val)
    getMovements: f getMovements()
    requestLoan: f requestLoan(val)
    withdraw: f withdraw(val)
    [[Prototype]]: Object
```

Note that private methods are seen as private properties by the browser and not fully implemented as yet.

```

class Account {
  locale = navigator.language;

  #movements = [];
  #pin;

  constructor(owner, currency, pin) {
    this.owner = owner;
    this.curency = currency;
    this.#pin = pin;

    console.log(`Thanks for opening an account ${owner}`);
  }
  getMovements() {
    return this.#movements;
  }
  deposit(val) {
    this.#movements.push(val);
  }
  withdraw(val) {
    this.deposit(-val);
  }
  requestLoan(val) {
    if (this.#approveLoan(val)) {
      this.deposit(val);
      console.log('Loan approved');
    }
  }
  #approveLoan(val) {
    return true;
  }
}

```

Chaining Methods

Chaining methods from a class object is mor or less the same a before so we will continue with the previous example class object.

```

acc1.deposit(300).deposit(500).withdraw(35).
requestLoan(25000).withdraw(4000);

```

✖ ▶ Uncaught TypeError: Cannot read properties of undefined (reading 'deposit')
at [script.js:622:18](#)

This will not work because the first deposit works but the second returns nothing because in the deposit method we are not returning anything explicitly. So it says undefined.

We need to return 'this' to make these methods chainable.

```
deposit(val) {  
  this.#movements.push(val);  
  return this;  
}  
  
withdraw(val) {  
  this.deposit(-val);  
  return this;  
}  
  
requestLoan(val) {  
  if (this.#approveLoan(val)) {  
    this.deposit(val);  
    console.log('Loan approved');  
    return this;  
  }  
}
```

Now public methods inside the class are chainable.

```
acc1.deposit(300).deposit(500).withdraw(35).requestLoan(25000).withdraw(4000);  
console.log(acc1.getMovements());
```

Loan approved

► (8) [250, -140, 1000, 300, 500, -35, 25000, -4000]

ES6: Classes summary

public field (similar to property, available on created object)

Private field (not accesible outside of class) denoted by #

Call to parent(super) class (necessary with extend). Needs to happen before accessing 'this'.

Instance property (available on created object)

Redefining private field

Public method

Referring private field and method

Private method

Getter method

Setter Method (use _ to set property with same name as method, and also getter)

Static Method (available only on the class. can not access instance properties nor methods, only static ones)

Create new object with new operator

```
class Student extends Person {  
  university = 'University of Lisbon';  
  #studyHours = 0;  
  #course;  
  static numSubjects = 10;  
  
  constructor(fullName, birthYear, startYear, course) {  
    super(fullName, birthYear);  
  
    this.startYear = startYear;  
  
    this.#course = course;  
  }  
  
  introduce() {  
    console.log(`I study ${this.#course} at ${this.university}`);  
  }  
  
  study(h) {  
    this.#makeCoffe();  
    this.#studyHours += h;  
  }  
  
  #makeCoffe() {  
    return 'Here is a coffe for you ☺';  
  }  
  
  get testScore() {  
    return this._testScore;  
  }  
  
  set testScore(score) {  
    this._testScore = score <= 20 ? score : 0;  
  }  
  
  static printCurriculum() {  
    console.log(`There are ${this.numSubjects} subjects`);  
  }  
}  
  
const student = new Student('Jonas', 2020, 2037, 'Medicine');
```

Person is the parent class of Student

Extends automatically sets inheritance between child class and parent class.

Student is a child class of Person

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

- Classes are “**syntax sugar**” over constructor functions.
- Classes are **not** hoisted
- Classes are **first class** citizens
- Class body is always executable in **strict mode**

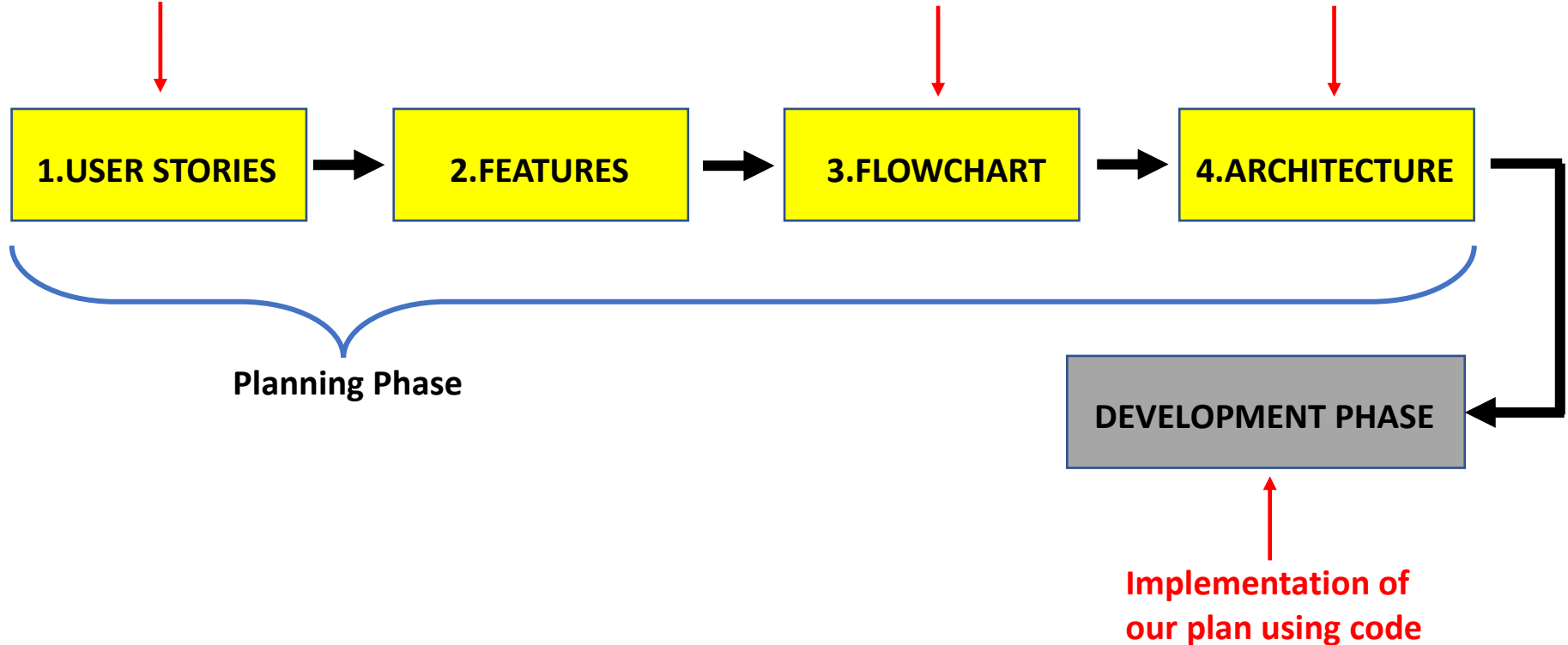
Mapty App: OOP,
Geolocation, External
Libraries and more

How to plan a web project

Description of the application's functionality from the user's perspective. all user Stories put together describe the entire application.

What we will build

How we will build it



Planning: User Story

- **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 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, 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 at a glance, 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 the app over time.

Planning: Features

STORIES

1

Log running workouts with location, distance, time, pace and steps/minute.

2

Log cycling workouts with location, distance, speed, and elevation gain.

3

See all my workouts at a glance

4

See workouts on a map.

5

See all workouts when I leave the app and come back later.

FEATURES

- Map where user clicks to add a new workout(best way to get Geo-location co-ordinates)
- 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.

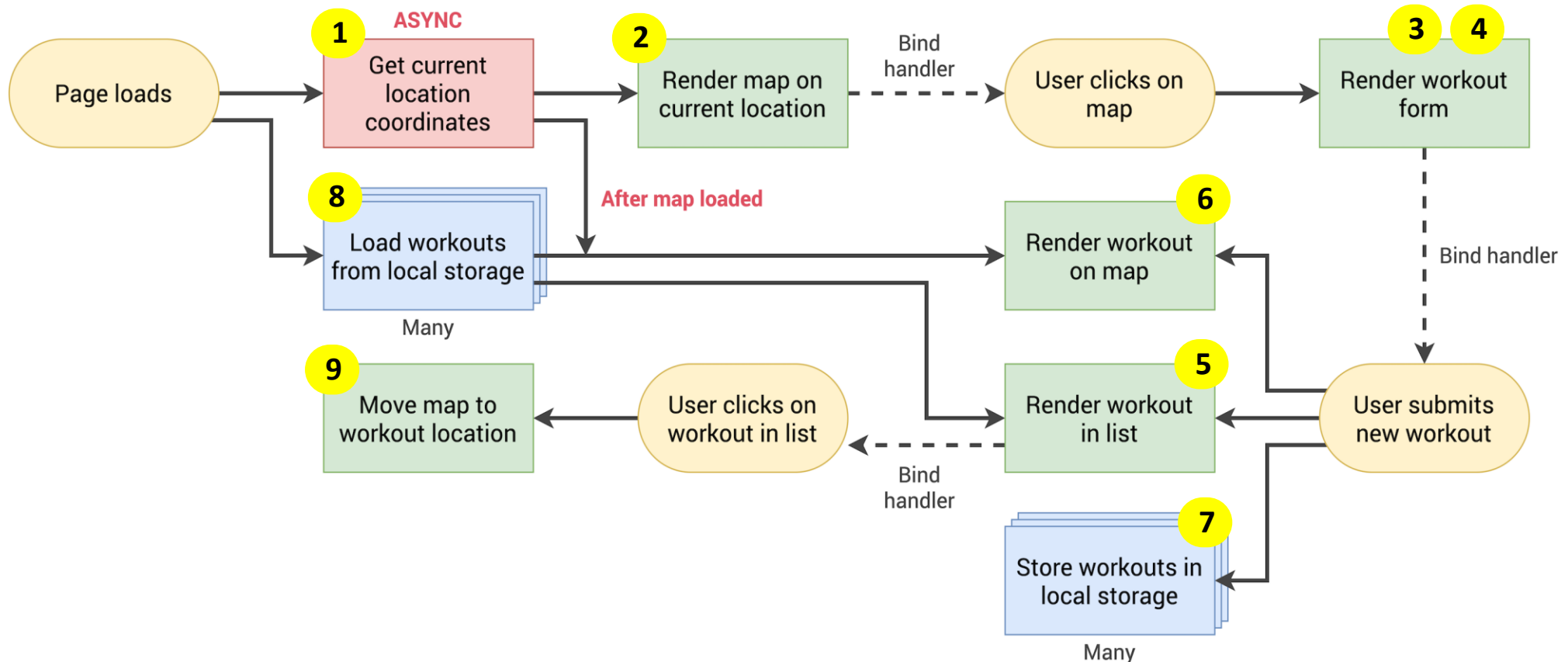
- Display all workouts on a map.

- Store workout data in the browser using storage API
- On page load, read the saved data from local storage and display

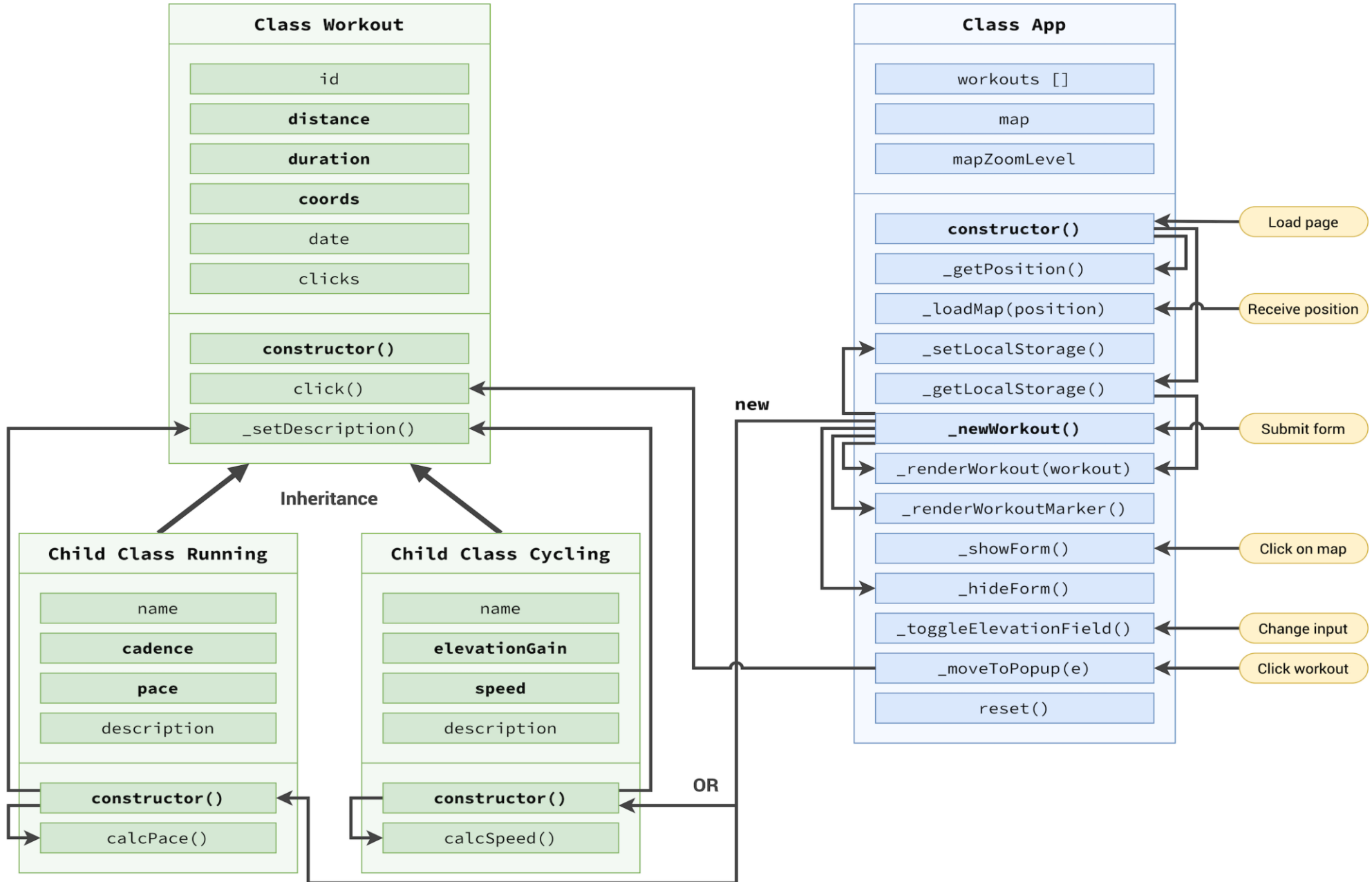
Planning: Flowchart

- 1 Geolocation to display the map at current location
- 2 Map where user clicks to add location
- 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 date in the browser
- 8 On page load, read the saved data and display
- 9 Move map to workout location on click



Planning: Architecture



Using the Geolocation API

The Geolocalisation API is a browser API like internationalisation or to access a users camera or to make a phone vibrate.

```
if (navigator.geolocation)
  navigator.geolocation.getCurrentPosition(
    function (position) {
      const { latitude } = position.coords;
      const { longitude } = position.coords;
      console.log(
        `https://www.openstreetmap.org/#map=18/${latitude}/${longitude}` );
    },
    function () {
      alert('Could not get your position');
    }
  );
```

If (navigator.geolocation) is used in case browser is really old version that does not support geolocalisation.

Function that takes position and de-structures it to latitude and longitude.

Testing latitude and longitude on open street maps.

Anonymous function to handle if geolocation cannot be obtained from the browser.

NOTE: for this to work, geolocation needs to be enabled in windows 10.

Windows->settings->privacy->app permissions->location 'Allow access to location on this device' = ON

Displaying a map using Leaflet library

<https://leafletjs.com/> is the leading open-source JS library for mobile-friendly interactive maps

From the downloads section of leaflet website we can use a Hosted Version of Leaflet from a CDN. The latest stable Leaflet release is available on several CDN's — to start using it straight away, place this in the head of your HTML code:

```
<link rel="stylesheet" href="style.css" />
```

```
<!-- LEFLET CDN & CSS -->
```

```
<link
  rel="stylesheet"
  href="https://unpkg.com/leaflet@1.8.0/dist/leaflet.css"
  integrity="sha512-
hoalWLoI8r4UszCkZ5kL8vay0GVae1oxXe/2A4A06J9+580uKHD03JdHb7NzwwzK5xr/Fs0W40kiNHxM9vyTtQ=="
  crossorigin=""
/>
```

We will use the leaflet library in our script code so the leaflet library needs to load first. additionally as we are putting the script in the header it needs to load with defer.

```
<script
  defer
  src="https://unpkg.com/leaflet@1.8.0/dist/leaflet.js"
  integrity="sha512-
BB3hKbKW0c9Ez/TAWyWxNXeoV9c1v6FieYiBieIWkplJauysF18NzgR1MBNBXf8/KABd1kX68nAh1wcDFLGPCQ=="
  crossorigin=""
></script>
```

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

```
<title>mapty // Map your workouts</title>
```



an open-source JavaScript library
for mobile-friendly interactive maps

[Overview](#) [Tutorials](#) [Docs](#) [Download](#) [Plugins](#) [Blog](#)



Apr 18, 2022 — [Leaflet 1.8.0](#) has been released!

Leaflet is the leading open-source JavaScript library for mobile-friendly interactive maps. Weighing just about 42 KB of JS, it has all the mapping [features](#) most developers ever need.

Leaflet is designed with *simplicity*, *performance* and *usability* in mind. It works efficiently across all major desktop and mobile platforms, can be extended with lots of [plugins](#), has a beautiful, easy to use and [well-documented API](#) and a simple, readable [source code](#) that is a joy to [contribute](#) to.



Here we create a map in the 'map' div, add [tiles](#) of our choice, and then add a marker with some text in a popup:

```
var map = L.map('map').setView([51.505, -0.09], 13);

L.tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
  attribution: '©< a href="https://www.openstreetmap.org/copyright">OpenStreetMap</a> contributors'
}).addTo(map);

L.marker([51.5, -0.09]).addTo(map)
  .bindPopup('A pretty CSS3 popup.<br>Easily customizable.')
  .openPopup();
```

The full use of leaflet can be found in the tutorials section but for now we only need to code on the overview tab in our JS code.

```

if (navigator.geolocation)
  navigator.geolocation.getCurrentPosition(
    function (position) {
      const { latitude } = position.coords;
      const { longitude } = position.coords;
      console.log(
        `https://www.openstreetmap.org/#map=18/${latitude}/${longitude}` );

      const coords = [latitude, longitude]; Create an array of latitude and longitude

      const map = L.map('map').setView(coords, 15);
      Insert the coords array into the default map view open

      L.tileLayer('https://{s}.tile.openstreetmap.fr/hot/{z}/{x}/{y}.png', {
        attribution:
          '&copy; <a
href="https://www.openstreetmap.org/copyright">OpenStreetMap</a> contributors',
      }).addTo(map);

      L.marker(coords) Insert the coords array into the default map marker
        .addTo(map)
        .bindPopup('A pretty CSS3 popup.<br> Easily customizable.')
        .openPopup();
    },
    function () {
      alert('Could not get your position');
    }
  );

```

L is a global variable for leaflet. Global variables are available to other JS files as long as they are loaded after where the global variable was loaded. I.e. L from the leaflet script is available in our script.js file because script.js was loaded after leaflet.js.

.org is changed to fr/hot

Displaying a map Marker Leaflet library

```
map.on('click', function (mapEvent) {  
    console.log(mapEvent);  
});
```

Map.on is a method defined in the leaflet.js file. It is like an event listener but specifically for leaflet.

In this case we are passing into the map.on Event listener a click action and a function to perform, in this case console log the map event. We can see that this map.on event listener generates an object with lots of properties, one being latlng which is the latitude and longitude of the location that was clicked.

```
map.on('click', function (mapEvent) {  
    console.log(mapEvent);  
    const { lat, lng } = mapEvent.latlng;  
    L.marker([lat, lng]).addTo(map).bindPopup('Workout').openPopup();  
});
```

```
▼ {originalEvent: PointerEvent, containerPoint: p, layerPoint: p, latlng:  
  v, type: 'click', ...} ⓘ  
  ► containerPoint: p {x: 529, y: 658}  
  ▼ latlng: v  
    lat: 41.379545742670516  
    lng: 2.128257751464844  
    ► [[Prototype]]: Object  
  ► layerPoint: p {x: 529, y: 658}  
  ► originalEvent: PointerEvent {isTrusted: true, pointerId: 2, width: 1,  
  ► sourceTarget: i {options: {...}, _handlers: Array(7), _layers: {...}, _zoo  
  ► target: i {options: {...}, _handlers: Array(7), _layers: {...}, _zoomBounc  
    type: "click"  
  ► [[Prototype]]: Object
```

We can create a lat and lng variable by de-structuring the mapEvent. Then cut the marker code from above and paste it in adding in the [lat, lng] array. WE can also change the marker text to 'Workout'.

<https://leafletjs.com/reference.html#popup>

From the leaflet docs section we can see what needs to be in the marker popup object.

TABLE OF CONTENTS

Map

Map Methods

Map Misc

UI Layers

Raster Layers

Vector Layers

Other Layers

Basic Types

Controls

Utility

DOM Utility

Base Classes

Misc

📄

Popup

Path overlays like polylines also have a `bindPopup` method. Here's a more complicated way to open a popup on a map:

```
var popup = L.popup()  
  .setLatLng(latlng)  
  .setContent('<p>Hello world!<br />This is a nice popup.</p>')  
  .openOn(map);
```

Creation

Factory	Description
<code>L.popup(<Popup options> options?, <Layer> source?)</code>	Instantiates a Popup object given an optional options object that describes its appearance and location and an optional source object that is used to tag the popup with a reference to the Layer to which it refers.

Options

Option	Type	Default	Description
pane	String	'popupPane'	Map pane where the popup will be added.
offset	Point	Point(0, 7)	The offset of the popup position.
maxWidth	Number	300	Max width of the popup, in pixels.



```
map.on('click', function (mapEvent) {  
  // console.log(mapEvent);  
  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();  
});
```

<https://leafletjs.com/reference.html#popup>

The L.popup documentation lists all the properties we can give this object and what they do.

Effectively here we have a method – L.popup() with an object inside L.popup({ object })

<https://leafletjs.com/reference.html#layer>

SetPopupContent is inherited from layer. Note how the technical documentation says that layer methods return 'this' as we previously studied in object orientated programming.

Rendering workout Input Form

```
let map, mapEvent;
```

```
if (navigator.geolocation)
```

... Code omitted for brevity

```
// Handling clicks on map
```

```
map.on('click', function (mapE) {
```

```
  mapEvent = mapE;
```

```
  form.classList.remove('hidden');
```

```
  inputDistance.focus();
```

```
});
```

```
},
```

```
function () {
```

```
  alert('Could not get your position');
```

```
}
```

```
);
```

```
form.addEventListener('submit', function (e) {
```

```
  e.preventDefault();
```

map and mapEvent variables at the top in global scope so multiple functions can access them.

MapEvent on the map click handler is not global so it is changed to mapE. The mapEvent is set to mapE in this handler.

When we click on the map we want the form to become visible and the input distance field of the form to have focus

We need an event listener for the form for a submit action. The default behaviour of form submit is to reload the page so we annul this with preventDefault(e).

```
// clear input fields
inputDistance.value =
inputDuration.value =
inputCadence.value =
inputElevation.value =
'';
```

When the form is submitted we want all the values in the fields to be blank.

```
// Display marker
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',
  })
)
```

We want the form to display a marker on the map. Note that here the form submit event listener needs access to the map and mapEvent variables so we need to put them at the top in global scope.

```
.setPopupContent('Workout')
.openPopup();
});
```

In the html the cadence is without class hidden and the elevation gain is with class hidden. This event listener is listening for a change to the drop down list for running or cycling and toggling the class of hidden on these two elements.

```
inputType.addEventListener('change', function () {
  inputElevation.closest('.form__row').classList.toggle('form__row--hidden');
  inputCadence.closest('.form__row').classList.toggle('form__row--hidden');
});
```

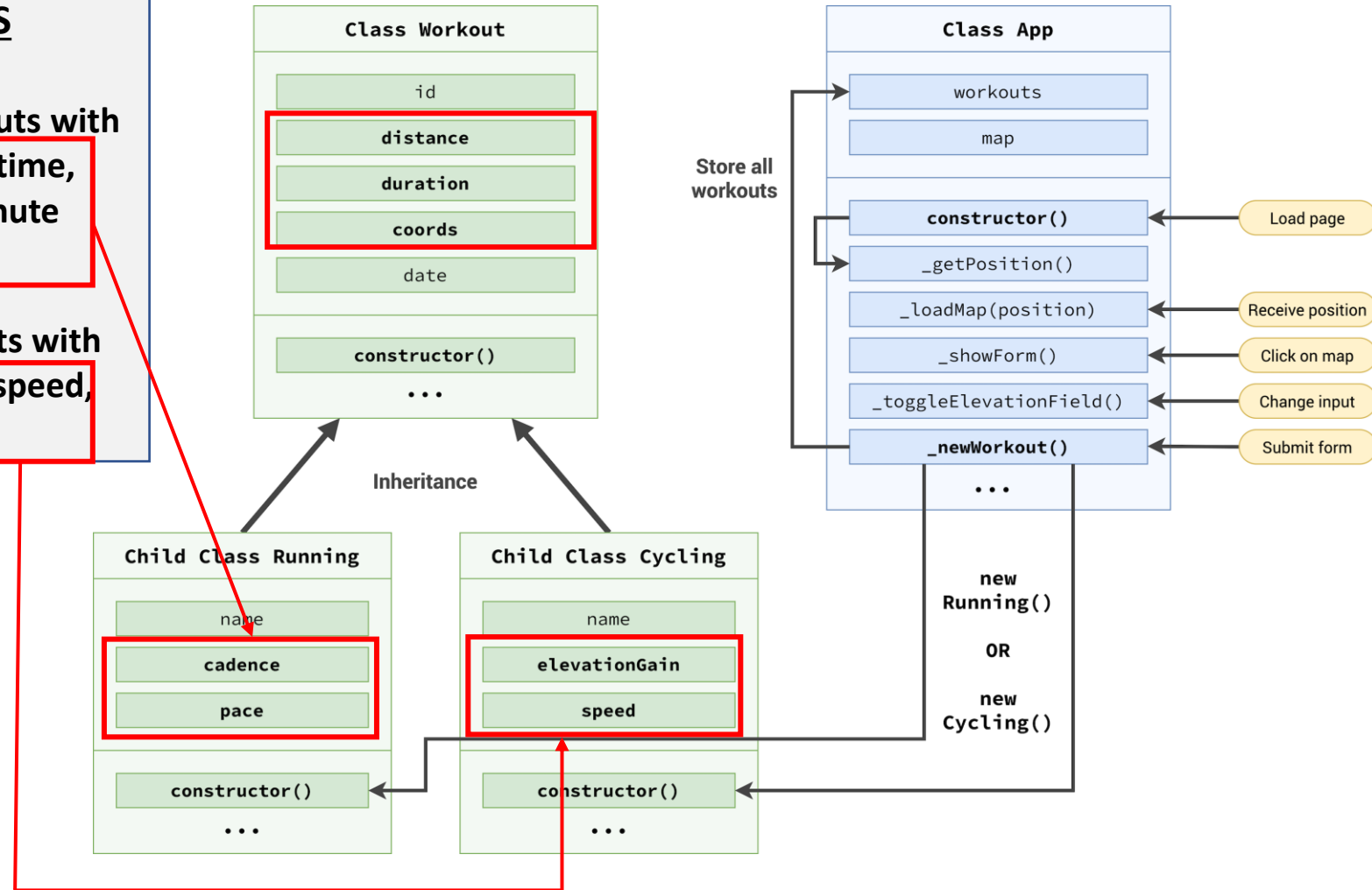
Project Architecture

Architecture is all about giving a project structure so that we can develop the functionality within the structure. In this lesson we will use classes and objects to give this project structure.

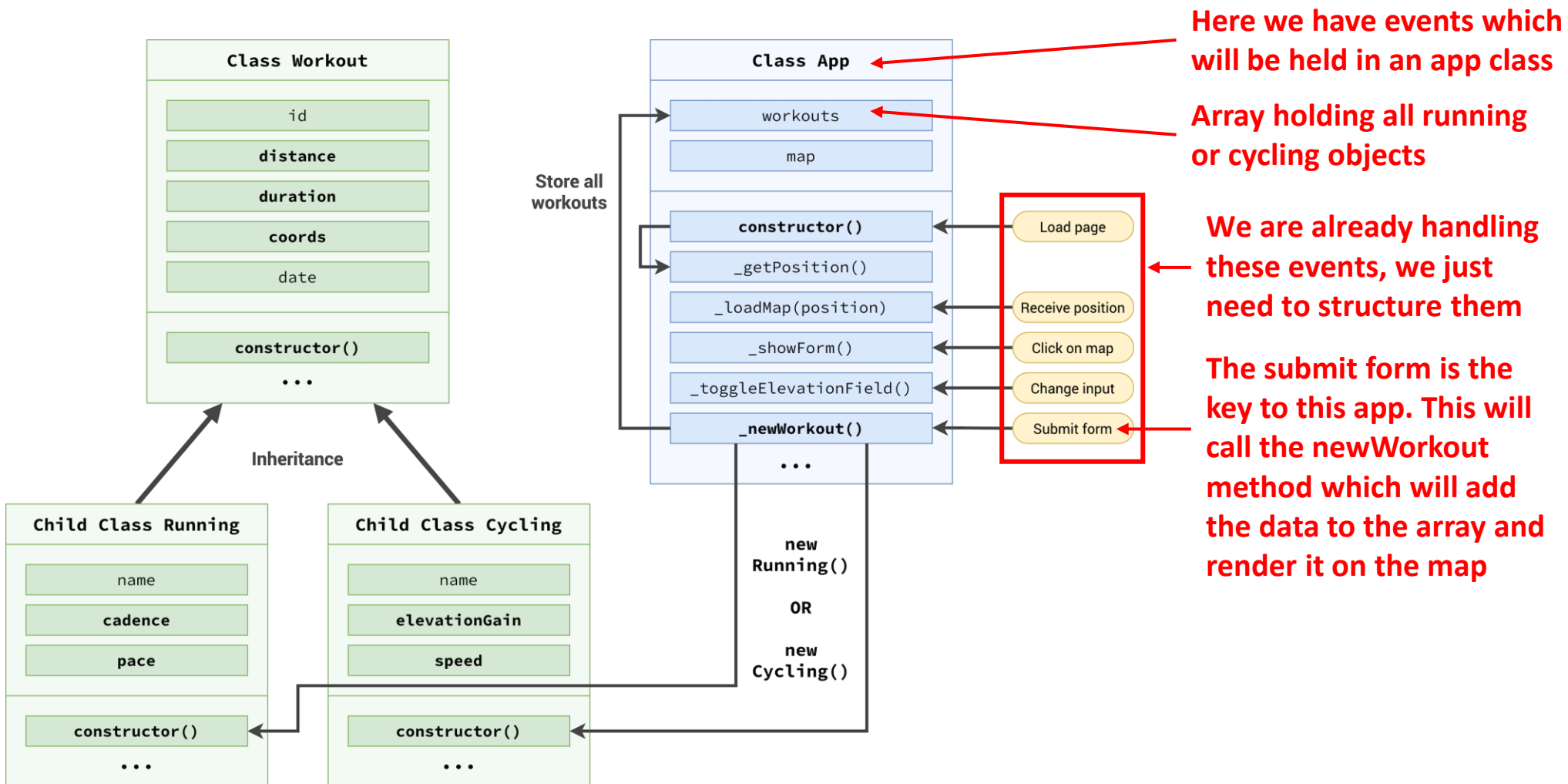
USER STORIES

1 Log running workouts with location, distance, time, pace and steps/minute (cadence).

2 Log cycling workouts with location, distance, speed, and elevation gain.



Workout class is the parent class because both running and cycling have Distance, duration and cords in common. Running and cycling (child classes) will inherit these from the parent class.



The reason for organising everything into classes is that properties and methods can be set to private or public or protected to implement encapsulation depending on if they need to interact outside of the class. This will make the code a lot easier to work with ensuring that the data is not being worked somewhere it should not be.

Refactoring for Project Architecture

```
class App {  
  #map;  
  #mapEvent;
```

1) Map and mapEvent become private properties within the class of App.

```
  constructor() {  
    this._getPosition();
```

2) Within the constructor we add the things we want to happen straight away. For 'this' App we want to get position. Note that it is a protected property.

```
    form.addEventListener('submit', this._newWorkout.bind(this));
```

```
    inputType.addEventListener('change', this._toggleElevationField);
```

```
  }
```

3) We need the event listeners for the form submit and toggleElevationField. By default the 'this' keyword is bound to the form but we want it to be bound to the App object for the newWorkout.

```
  _getPosition() {  
    if (navigator.geolocation)  
      navigator.geolocation.getCurrentPosition(  
        this._loadMap.bind(this),  
        function () {  
          alert('Could not get your position');  
        }  
      );  
  }  
}
```

4) The get position method also needs the load map function to to be bound to 'this' App class. This is because it is a regular call-back function so the this is undefined unless we bind it.

```
loadMap(position) {  
  const { latitude } = position.coords; ← 5) Load map now takes a position  
  const { longitude } = position.coords;  
  // console.log(`https://www.openstreetmap.org/#map=18/${latitude}/${longitude}`);
```

```
  const coords = [latitude, longitude];  
  this.#map = L.map('map').setView(coords, 15); ← 6) Map is now a private property  
  // console.log(map); ← this.#map
```

```
  L.tileLayer('https://{s}.tile.openstreetmap.fr/hot/{z}/{x}/{y}.png', {  
    attribution: &copy; <a  
href="https://www.openstreetmap.org/copyright">OpenStreetMap</a> contributors',  
  }).addTo(this.#map);
```

```
  // Handling clicks on map  
  this.#map.on('click', this._showForm.bind(this)); ← 7) When we click on the map we want  
  } ← it to show form, a protected method.  
  It also needs to be bound to 'this'.
```

```
  _showForm(mapE) {  
    this.#mapEvent = mapE; ← 8) Show form takes a MapE. MapE is also a private property  
    form.classList.remove('hidden'); ← of App class so mapEvent becomes this.#mapEvent  
    inputDistance.focus();  
  }
```

```
  toggleElevationField() { ← 9) Toggle elevation field stays the same but is put  
    inputElevation.closest('.form__row').classList.toggle('form__row--hidden'); ← into it's own method.  
    inputCadence.closest('.form__row').classList.toggle('form__row--hidden');  
  } _
```



```
_newWorkout(e) {  
  e.preventDefault();
```

10) New workout needs a prevent default because the default action of a form submit is a page reload which we do not want.

```
// clear input fields  
inputDistance.value = inputDuration.value = inputCadence.value =  
inputElevation.value = '';
```

```
// Display marker  
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();
```

11) These two private properties now need a this keyword

```
}  
}  
  
const app = new App();
```

12) The constructor method gets called immediately when the App class is called. So we want get position, the form event listener and the toggle event listener to be in the constructor. Everything else can be its own protected or unprotected method within the class.

Managing workout Data: Creating Classes

```
class Workout {  
    date = new Date();  
    id = (Date.now() + '').slice(-10);  
  
    constructor(coords, distance, duration) {  
        this.coords = coords; // [lat, lng]  
        this.distance = distance; // in KM  
        this.duration = duration; // in mins  
    }  
}
```

1) The parent class of Workout is created according to the architecture plan. Note the Date is automatically generated with new Date and the ID is just generating a 'random' number by slicing off the last 10 digits of the date and time now.

2) The constructor takes the three parameters as per the plan and we define them with the this keyword.

3) The child class of running extends the parent class of Workout.

4) The constructor takes the same three parameters as the workout class plus the additional cadence.

5) We can calculate the pace within the Running class and at the same time call it from the constructor.

```
class Running extends Workout {  
    constructor(coords, distance, duration, cadence) {  
        super(coords, distance, duration);  
        this.cadence = cadence;  
        this.calcPace();  
    }  
  
    calcPace() {  
        // min/Km  
        this.pace = this.duration / this.distance;  
        return this.pace;  
    }  
}
```

6) The child class of cycling extends the parent class of Workout.

7) The constructor takes the same three parameters as the workout class plus the additional elevationGain.

8) We can calculate the speed within the Cycling class and at the same time call it from the constructor.

```
class Cycling extends Workout {  
  constructor(coords, distance, duration, elevationGain) {  
    super(coords, distance, duration);  
    this.elevationGain = elevationGain;  
    this.calcSpeed();  
  }  
  calcSpeed() {  
    // km/h  
    this.speed = this.distance / this.duration / 60;  
    return this.speed;  
  }  
}
```

9) We can test the three classes by passing in parameters into each of the child class and logging what we get.

```
const run1 = new Running([39, -12], 5.2, 24, 178);  
const cycling1 = new Cycling([39, -12], 5.2, 27, 95, 523);  
console.log(run1, cycling1);
```

```
Running {date: Mon May 23 2022 16:30:13 GMT+0200 (Central  
▼ European Summer Time), id: '3316213694', coords: Array(2),  
  distance: 5.2, duration: 24, ...} ⓘ  
  cadence: 178  
  ► coords: (2) [39, -12]  
  ► date: Mon May 23 2022 16:30:13 GMT+0200 (Central European  
  distance: 5.2  
  duration: 24  
  id: "3316213694"  
  pace: 4.615384615384615  
  ► [[Prototype]]: Workout
```

```
Cycling {date: Mon May 23 2022 16:30:13 GMT+0200 (Central  
▼ European Summer Time), id: '3316213694', coords: Array(2),  
  distance: 5.2, duration: 27, ...} ⓘ  
  ► coords: (2) [39, -12]  
  ► date: Mon May 23 2022 16:30:13 GMT+0200 (Central European  
  distance: 5.2  
  duration: 27  
  elevationGain: 95  
  id: "3316213694"  
  speed: 0.0032098765432098763  
  ► [[Prototype]]: Workout
```

Creating a new Workout

The best way to code this is to write a list of tasks that we want the new workout method to perform then work through the list coding them one by one.

2a) the array of values is de-structured and for every input it checks if the value is a finite number returning true or false if one or more values is not.

```
_newWorkout(e) {  
  // validate inputs as numbers helper function  
  const validInputs = (...inputs) =>  
    inputs.every(inp => Number.isFinite(inp));  
  // validate inputs as positive numbers helper function  
  const allPositive = (...inputs) => inputs.every(inp => inp > 0);  
  
  e.preventDefault();  
  
  // 1. get data from form  
  const type = inputType.value;  
  const distance = +inputDistance.value;  
  const duration = +inputDuration.value;  
  
  const { lat, lng } = this.#mapEvent.latlng;  
  let workout;
```

2b) All positive works the same way but checks that value is greater than zero.

1) We get the values from the html elements that have already been querySelected at the top of the code.

4a 7a) Lat and lng are removed from the displayMarker section and put here at the tip of the method as we need access to it lower down in the scope chain. (Scope chain always looks up!)

```
// 2. if Workout running, create running object
```

```
if (type === 'running') {  
  const cadence = +inputCadence.value;  
  // 3. Check if data is valid  
  if (  
    !validInputs(distance, duration, cadence) ||  
    !allPositive(distance, duration, cadence)  
  )  
    return alert('Inputs have to be positive numbers!');  
  
  // 4. build running workout object  
  workout = new Running([lat, lng], distance, duration, cadence);  
}
```

2) We have if statements to handle running or Cycling.

We get the cadence value from the form.

3) We use a validInputs helper function which takes an array of parameters and checks if they are numbers or allPositive values.

4) The workout object (for running) can be built now. The workout variable needs to be initialised at the top of the method as a blank.

```
// 5. if Workout Cycling, create cycling object
```

```
if (type === 'cycling') {  
  const elevation = +inputElevation.value;  
  // 6. Check if data is valid  
  if (  
    !validInputs(distance, duration, elevation) ||  
    !allPositive(distance, duration)  
  )  
    return alert('Inputs have to be positive numbers!');  
  
  // 7. build cycling workout object  
  workout = new Cycling([lat, lng], distance, duration, elevation);  
}
```

5) We have an if statement to handle cycling.

We get elevationGain from the form.

6) And validate data. Note that elevation gain can be negative, i.e. cycling downhill so it does not need to be validated as positive.

7) The workout object (for cycling) can be built now. Note that lat and lng come from a variable defined at the top of the method.

// 8. Add new object to workout array

```
this.#workouts.push(workout);  
console.log(workout);
```

8) We can push the running or cycling object to the workouts array. The workouts array needs to be initialised at the top of the method.

// 9 Render workout on map as marker

```
this._renderWorkoutMarker(workout);
```

9) Render the map marker becomes a separate method that we pass workout into.

// 10. Render workout on list

// 11. Hide form and clear input fields

```
inputDistance.value = inputDuration.value = inputCadence.value =  
inputElevation.value = '';
```

```
}
```

```
_renderWorkoutMarker(workout) {
```

```
  L.marker(workout.coords)
```

```
    .addTo(this.#map)
```

```
    .bindPopup({
```

```
      L.popup({
```

```
        maxWidth: 250,
```

```
        minWidth: 100,
```

```
        autoClose: false,
```

```
        closeOnClick: false,
```

```
        className: `${workout.type}-popup`,
```

```
      })
```

```
    )
```

```
    .setPopupContent('workout')
```

```
    .openPopup();
```

```
}
```

```
}
```

8a) We need to initialise workouts as a blank array in the App class to be able to push to it later in the code.

```
class App {  
  #map;  
  #mapEvent;  
  #workouts = [];
```

9a) In the css file there is a class for cycling-popup and running-popup so we can use an object literal to define the className of the leaflet map popup.

```
class Running extends Workout {  
  type = 'running';
```

But we need to specify type in the running class and the cycling class.

Rendering Workouts

```
// 10. Render workout on list  
this._renderWorkout(workout);
```

10) We can call a protected method within the `_newWorkout` event handler.

```
_renderWorkout(workout) {  
  let html = `  
    <li class="workout workout--${workout.type}" data-id="${workout.id}">  
      <h2 class="workout__title">${workout.description}</h2>  
      <div class="workout__details">  
        <span class="workout__icon">${  
          workout.type === 'running' ? '🏃♂️' : '🚴♀️'  
        }</span>  
        <span class="workout__value">${workout.distance}</span>  
        <span class="workout__unit">km</span>  
      </div>  
      <div class="workout__details">  
        <span class="workout__icon">⌚</span>  
        <span class="workout__value">${workout.duration}</span>  
        <span class="workout__unit">min</span>  
      </div>  
    `;  
  ;
```

10a) we define a let variable called `html` and set the contents to a template literal.

10b) inserting the values from the workout object.

```
if (workout.type === 'running')
```

```
html += `
```

```
<div class="workout__details">
```

```
<span class="workout__icon">⚡</span>
```

```
<span class="workout__value">${workout.pace.toFixed(1)}</span>
```

```
<span class="workout__unit">min/km</span>
```

```
</div>
```

```
<div class="workout__details">
```

```
<span class="workout__icon">🚴</span>
```

```
<span class="workout__value">${workout.cadence}</span>
```

```
<span class="workout__unit">spm</span>
```

```
</div>
```

```
</li>
```

```
`;
```

```
if (workout.type === 'cycling')
```

```
html += `
```

```
<div class="workout__details">
```

```
<span class="workout__icon">⚡</span>
```

```
<span class="workout__value">${workout.speed.toFixed(1)}</span>
```

```
<span class="workout__unit">km/h</span>
```

```
</div>
```

```
<div class="workout__details">
```

```
<span class="workout__icon">⬆️</span>
```

```
<span class="workout__value">${workout.elevationGain}</span>
```

```
<span class="workout__unit">m</span>
```

```
</div>
```

```
</li>
```

```
`;
```

10c) To handle running we append a block of html and adjust values. We use toFixed to round the calculated values to one decimal place.

10d) To handle cycling we append an alternative block of html and adjust values. We use toFixed to round the calculated values to one decimal place.


```
form.insertAdjacentHTML('afterend', html);
```

← 10e) Use insertAdjacentHTML afterend of the form.

```
// 11. Hide form and clear input fields
```

```
this._hideForm();
```

11) Once the workout has been rendered to the list we need to hide the input form.

```
_setDescription() {
```

```
// prettier-ignore
```

```
const months = ['January', 'February', 'March', 'April', 'May', 'June', 'July',  
'August', 'September', 'October', 'November', 'December'];
```

12) The workout description is defined in the Workout class.

```
this.description = `${this.type[0].toUpperCase()}${this.type.slice(1)} on ${  
months[this.date.getMonth()]
```

12a) capitalise first letter of workout type

12b) this.date.getMonth() will return a value from 0 to 11 for the month which can be used to identify a position in the months array.

```
constructor(coords, distance, duration,  
cadence) {
```

```
super(coords, distance, duration);
```

```
this.cadence = cadence;
```

```
this.calcPace();
```

```
this._setDescription();
```

13a) we need to call the _setDescription method from the constructor of the running and cycling objects.

```

constructor(coords, distance, duration,
elevationGain) {
    super(coords, distance, duration);
    this.elevationGain = elevationGain;
    this.calcSpeed();
    this._setDescription();
}

```

13b) we need to call the `_setDescription` method from the constructor of the running and cycling objects.

```

_renderWorkoutMarker(workout) {
    L.marker(workout.coords)
        .addTo(this.#map)
        .bindPopup(
            L.popup({
                maxWidth: 250,
                minWidth: 100,
                autoClose: false,
                closeOnClick: false,
                className: `${workout.type}-popup`,
            })
        )
        .setPopupContent(
            `${workout.type} === 'running' ? '🏃♂️' : '🚴♀️' } ${workout.description}`
        )
        .openPopup();
}

```

13) Workout popup marker for the map needs to be modified to include the workout description-

Move to marker on click

```
class App {  
  #map;  
  #mapZoomLevel = 15;  
  #mapEvent;  
  #workouts = [];
```

```
  constructor() {  
    this._getPosition();  
    form.addEventListener('submit', this._newWorkout.bind(this));  
    inputType.addEventListener('change', this._toggleElevationField);  
    containerWorkouts.addEventListener('click', this._moveToPopup.bind(this));  
  }
```

2) We call the `moveToPopup` method from the app class constructor binding this.

```
  _moveToPopup(e) {  
    const workoutEl = e.target.closest('.workout');  
    if (!workoutEl) return;  
    const workout = this.#workouts.find(  
      work => work.id === workoutEl.dataset.id  
    );
```

1) We want the map to move to a marker location when we click on a workout in the `containerWorkouts` so for that we will create a method in the App class.

3) We use event delegation to find the closest element to the target with a class of `'workout'`

4) From the element we can find the `dataset.id` and use this to find the marker on the map.

```
    this.#map.setView(workout.coords, this.#mapZoomLevel, {  
      animate: true,  
      pan: {  
        duration: 1,  
      },  
    });  
  }
```

5) We can now use the leaflet `setView` method as before passing in `coords` and `zoom level`. Note that now `#mapZoomLevel` is a private property of class App. `Animate`, `pan` and `duration` are properties of the `setView` leaflet Method.

Working with local storage

Local storage is a persistent memory in the browser that can be used to store small amounts of data from a website so that it is not lost when the browser is closed or reloaded.

```
// 12. Set local Storage to all workouts
```

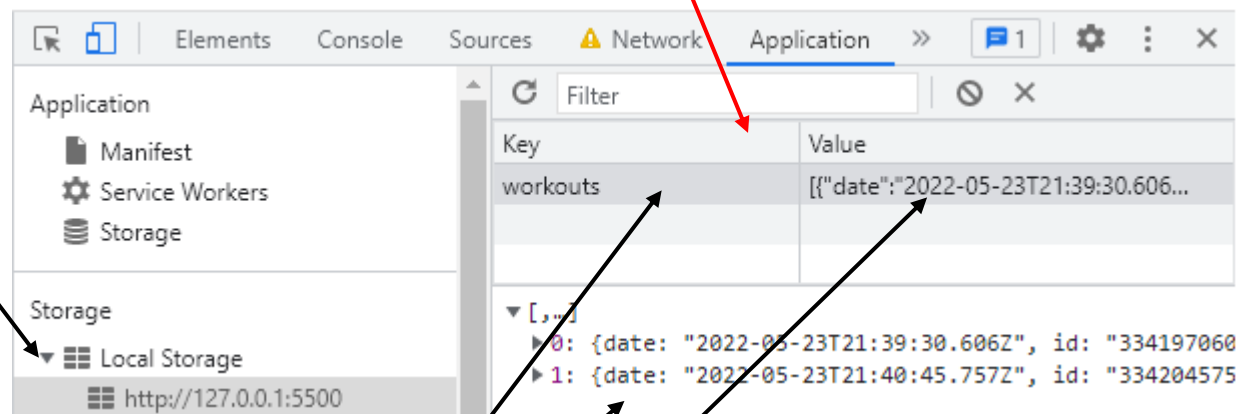
```
this._setLocalStorage();
```

As part of the `_newWorkout` method we can set the local storage to 'this' which will be the workout object and all of it's data.

```
_setLocalStorage() {  
  localStorage.setItem('workouts', JSON.stringify(this.#workouts));  
}
```

`localStorage` is a JavaScript API where we can set an item that is a **key value pair**. In this case we set the key to `workouts` and the value as a JSON stringified array with all the workout objects in it.

In dev tools we can see the local storage in application->local Storage then select the URL for this website.



Here we see a key of 'workouts' which has a value that contains an array of the workouts objects.

```
class App {  
  #map;  
  #mapZoomLevel = 15;  
  #mapEvent;  
  #workouts = [];  
  
  constructor() {  
    // GEt user position  
    this._getPosition();  
  
    // Get data from local storage  
    this._getLocalStorage();  
  
    // Attach event handlers  
    form.addEventListener('submit', this._newWorkout.bind(this));  
    inputType.addEventListener('change', this._toggleElevationField);  
    containerWorkouts.addEventListener('click', this._moveToPopup.bind(this));  
  }  
}
```

To retrieve the data from local storage we can create a function called `getLocalStorage` and call it from the App constructor.



```
_getLocalStorage() {  
  const data = JSON.parse(localStorage.getItem('workouts'));  
  
  if (!data) return;  
  
  this.#workouts = data;  
  
  this.#workouts.forEach(work => {  
    this._renderWorkout(work);  
  });  
}
```

The inverse of JSON stringify is parse so we can get the data from the key:value pair in local storage and assign it to the workouts array.

The guard clause returns if there is no data in local storage.

Now we can loop over the #workouts array with a foreach and call the render workout function previously created.

Because we are calling the get local storage method from the constructor of the app it will be the second function to get executed after get position. Once get position is complete the `_loadMap` starts and this takes time so when the `getLocalStorage` is complete the map is still loading so the markers do not get loaded. This is because JavaScript is asynchronous.

```
// Load workouts from local storage after map has finished loading  
this.#workouts.forEach(work => {  
  this._renderWorkoutMarker(work);  
});
```

To solve this we also need to add the foreach loop of the workouts in the `_loadMap` method after the code that handles the map loading so that it will execute only when the map is loaded.

An important note about getting data from local storage. When we use JSON stringify the prototype chain is removed. When we then parsed it the prototype chain is no longer there so they cannot inherit any of their methods. This can be a big problem when working with local storage and OOP.

```
▶ toString: f toString()
▶ valueOf: f valueOf()
▶ __defineGetter__: f __defineGetter__()
▶ __defineSetter__: f __defineSetter__()
▶ __lookupGetter__: f __lookupGetter__()
▶ __lookupSetter__: f __lookupSetter__()
▶ proto : Object
▼ get __proto__: f __proto__()
  length: 0
  name: "get __proto__"
  arguments: (...)
  caller: (...)
  ▼ [[Prototype]]: f ()
    ▶ apply: f apply()
      arguments: (...)
    ▶ bind: f bind()
    ▶ call: f call()
      caller: (...)
    ▶ constructor: f Function()
      length: 0
      name: ""
    ▶ toString: f toString()
    ▶ Symbol(Symbol.hasInstance): f [Symbol.hasInstance]()
    ▶ get arguments: f ()
    ▶ set arguments: f ()
    ▶ get caller: f ()
    ▶ set caller: f ()
      [[FunctionLocation]]:
    ▶ [[Prototype]]: Object
    ▶ [[Scopes]]: Scopes[0]
    ▶ [[Scopes]]: Scopes[0]
  ▶ set __proto__: f __proto__()
```

```
▼ {date: '2022-05-23T21:39:30.606Z', id: '3341970606', coords: Array(2), distance: 2, duration: 2, ...} ⓘ
  cadence: 2
  ▶ coords: (2) [41.38096258902881, 2.130961418151856]
  date: "2022-05-23T21:39:30.606Z"
  description: "Running on May"
  distance: 2
  duration: 2
  id: "3341970606"
  pace: 1
  type: "running"
  ▼ [[Prototype]]: Object
    ▶ constructor: f Object()
    ▶ hasOwnProperty: f hasOwnProperty()
    ▶ isPrototypeOf: f isPrototypeOf()
    ▶ propertyIsEnumerable: f propertyIsEnumerable()
    ▶ toLocaleString: f toLocaleString()
    ▶ toString: f toString()
    ▶ valueOf: f valueOf()
    ▶ __defineGetter__: f __defineGetter__()
    ▶ __defineSetter__: f __defineSetter__()
    ▶ __lookupGetter__: f __lookupGetter__()
    ▶ __lookupSetter__: f __lookupSetter__()
    __proto__: (...)
    ▶ get __proto__: f __proto__()
    ▶ set __proto__: f __proto__()
```

We can see that the `__proto__` only contains the inbuilt methods and none of our created methods in the classes.

```
reset() {  
  localStorage.removeItem('workouts');  
  location.reload();  
}
```



The reset method in the App class will erase local storage and reload the page.

This method can be called from the console with `App.reset()`.

Asynchronous JavaScript: Promises, Async/Await, AJAX

Asynchronous JavaScript, AJAX and APIs

SYNCRONOUS



```
const p = document.querySelector('p');  
p.textContent = 'My name is Jonas!';  
alert('Text set!');  
p.style.color = 'red';
```

- Most code is **synchronous**
- Synchronous code is **executed line by line**
- Each line of code **waits** for previous line to finish

This can create problems when one line of code takes a long time to run. For example the alert window will block the execution of the code until the user clicks OK and the window disappears and the code can continue running.

THREAD OF EXECUTION



part of execution context
that actually executes the
code in computer's CPU

ASYNCRONOUS

Asynchronous

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

Example: timer with call-back

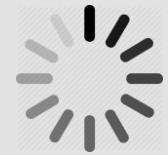
THREAD OF
EXECUTION



Executed after
all other code

BACKGROUND

Timer Running



- Asynchronous code is executed **after** a task that runs in the background finishes.
- Asynchronous code is **non-blocking**
- Execution does not wait for an asynchronous task to finish
- Call-back functions alone do **NOT** make code asynchronous

`image.src` attribute is implemented in an asynchronous way. when the image finishes loading then JavaScript emits a load event. We can use a call back function to listen for the load event and perform actions such as `fadeIn`.

ASYNCHRONOUS

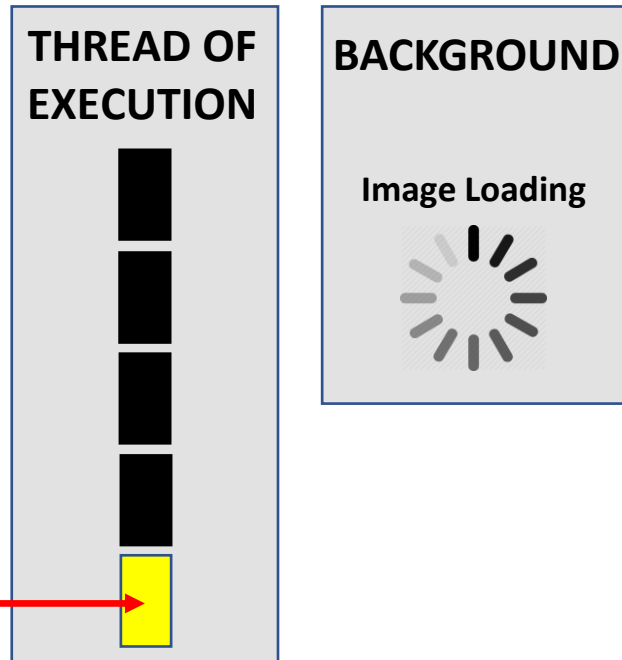
Asynchronous

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

Example: Asynchronous image load with event and call-back

`addEventListener` does not automatically make code asynchronous!

Executed after all other code

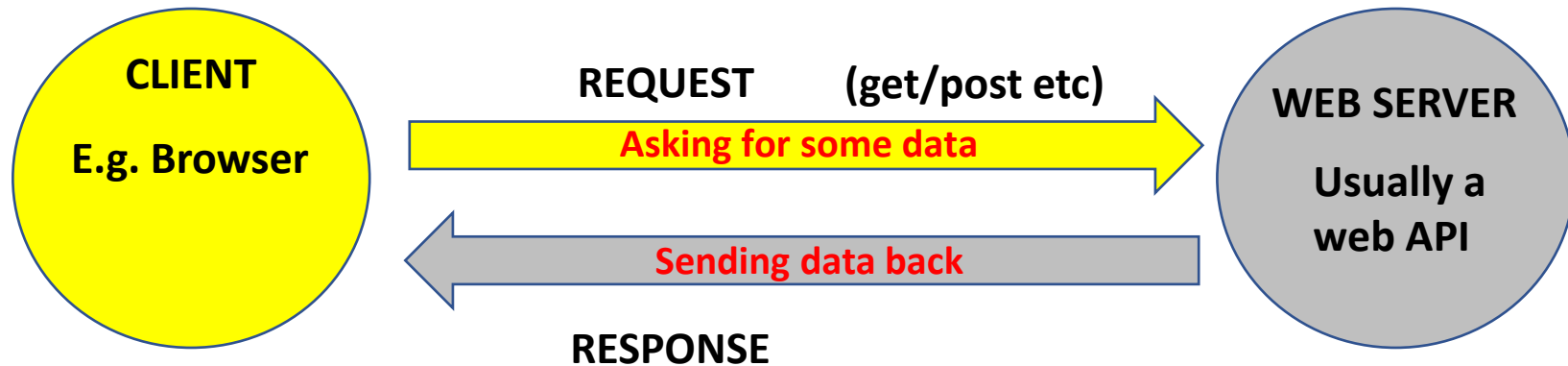


We deferred the action of `fadeIn` into the future making the code asynchronous. An event listener waiting for a click in the background is not asynchronous. What makes this code asynchronous is the fact that the image is loading in the background.

- Asynchronous code is executed **after** a task that runs in the background finishes.
- Asynchronous code is **non-blocking**
- Execution does not wait for an asynchronous task to finish
- Call-back functions alone do **NOT** make code asynchronous

AJAX

Asynchronous JavaScript And XML: Allows us to communicate with remote web servers in an **asynchronous way**. With AJAX calls we can **request data** from web servers dynamically.



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 many types of API in software development

- DOM API**
- Geolocation API**
- Own class API**

“Online” API: Application running on a server that receives requests for data, and sends back a response.

WE can build our own web APIs (requires back-end development eg with node JS) or use 3rd party APIs.

There is an API for everything:

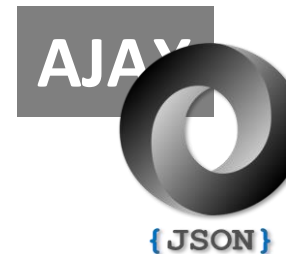
- Weather data
- Data about countries
- Flights Data
- Currency conversion data
- APIs for sending email or SMS
- Google Maps

Millions of possibilities....

In the early days of the web data was transferred using the XML format but this has become obsolete



API Data transfer uses the modern JSON format but the name AJAX has stuck. JSON format is stringified JavaScript object. **JavaScript Object Notation**.



AJAX call: XMLHttpRequest

<https://github.com/public-apis/public-apis> A list of free public APIs

We can open a XML http request to restcountriesAPI specifying the country then send the request.

```
const request = new XMLHttpRequest();
request.open('GET', 'https://restcountries.com/v3.1/name/spain');
request.send();
```

Then use an event listener to wait for the load event.

```
request.addEventListener('load', function () {
    console.log(this.responseText);
});
```

```
[{"name":{"common":"Spain","official":"Kingdom of Spain","nativeName":{"spa":{"official":"Reino de España","common":"España"}}},"tld": "script.js:13
[\".es\"],\"cca2\":\"ES\",\"ccn3\":\"724\",\"cca3\":\"ESP\",\"cioc\":\"ESP\",\"independent\":true,\"status\":\"officially-assigned\",\"unMember\":true,\"currencies\":{\"EUR\":
{\"name\":\"Euro\",\"symbol\":\"€\"}},\"idd\":{\"root\":\"+3\",\"suffixes\":[\"4\"]},\"capital\":[\"Madrid\"],\"altSpellings\":[\"ES\",\"Kingdom of Spain\",\"Reino de
España\"],\"region\":\"Europe\",\"subregion\":\"Southern Europe\",\"languages\":{\"spa\":\"Spanish\"},\"translations\":{\"ara\":{\"official\":\"مملكة
إسبانيا\",\"common\":\"إسبانيا\"},\"ces\":{\"official\":\"Španělské království\",\"common\":\"Španělsko\"},\"cym\":{\"official\":\"Kingdom of
Spain\",\"common\":\"Spain\"},\"deu\":{\"official\":\"Königreich Spanien\",\"common\":\"Spanien\"},\"est\":{\"official\":\"Hispaania
Kuningriik\",\"common\":\"Hispaania\"},\"fin\":{\"official\":\"Espanjan kuningaskunta\",\"common\":\"Espanja\"},\"fra\":{\"official\":\"Royaume
d'Espagne\",\"common\":\"Espagne\"},\"hrv\":{\"official\":\"Kraljevina Španjolska\",\"common\":\"Španjolska\"},\"hun\":{\"official\":\"Spanyol
Királyság\",\"common\":\"Spanyolország\"},\"ita\":{\"official\":\"Regno di Spagna\",\"common\":\"Spagna\"},\"jpn\":{\"official\":\"スペイン王国\",\"common\":\"スペイン\"},\"kor\":
{\"official\":\"에스파냐 왕국\",\"common\":\"스페인\"},\"nld\":{\"official\":\"Koninkrijk Spanje\",\"common\":\"Spanje\"},\"per\":{\"official\":\"پادشاهی
اسپانیا\",\"common\":\"اسپانیا\"},\"pol\":{\"official\":\"Królestwo Hiszpanii\",\"common\":\"Hiszpania\"},\"por\":{\"official\":\"Reino de
Espanha\",\"common\":\"Espanha\"},\"rus\":{\"official\":\"Королевство Испания\",\"common\":\"Испания\"},\"slk\":{\"official\":\"Španielske
kráľovstvo\",\"common\":\"Španielsko\"},\"spa\":{\"official\":\"Reino de España\",\"common\":\"España\"},\"swe\":{\"official\":\"Konungariket
Spanien\",\"common\":\"Spanien\"},\"urd\":{\"official\":\"مملکت ہسپانیہ\",\"common\":\"ہسپانیہ\"},\"zho\":{\"official\":\"西班牙王国\",\"common\":\"西班牙\"},\"latlng\":
[40.0,-4.0],\"landlocked\":false,\"borders\":[\"AND\",\"FRA\",\"GIB\",\"PRT\",\"MAR\"],\"area\":505992.0,\"demonyms\":{\"eng\":{\"f\":\"Spanish\",\"m\":\"Spanish\"},\"fra\":
{\"f\":\"Espagnole\",\"m\":\"Espagnol\"}},\"flag\":\"\u2192D83C\u2192DDEA\u2192D83C\u2192DDF8\",\"maps\":{\"googleMaps\":\"https://goo.gl/maps/138JaXW8EZzRVitY9\", \"openStreetMaps\":\"http
s://www.openstreetmap.org/relation/1311341\"},\"population\":47351567,\"gini\":{\"2018\":34.7},\"fifa\":\"ESP\",\"car\":{\"signs\":[\"E\"],\"side\":\"right\"},\"timezones\":
[\"UTC\",\"UTC+01:00\"],\"continents\":[\"Europe\"],\"flags\":{\"png\":\"https://flagcdn.com/w320/es.png\",\"svg\":\"https://flagcdn.com/es.svg\"},\"coatOfArms\":{\"png\":\"ht
tps://mainfacts.com/media/images/coats of arms/es.png\",\"svg\":\"https://mainfacts.com/media/images/coats of arms/es.sv
g\"},\"startOfWeek\":\"monday\",\"capitalInfo\":{\"latlng\":[40.4,-3.68]},\"postalCode\":{\"format\":\"####\",\"regex\":\"^\\d{5}$\"}}]
```

3) Wrap the whole lot in a function

```
const getCountryData = function (country) {  
  const request = new XMLHttpRequest();  
  request.open('GET', `https://restcountries.com/v3.1/name/${country}`);  
  request.send();
```

```
  request.addEventListener('load', function () {  
    const [data] = JSON.parse(this.responseText);
```

```
    const name = data.name.common;  
    const flag = data.flags.svg;  
    const region = data.region;  
    const language = Object.values(data.languages)[0];  
    const currency = Object.values(data.currencies)[0].name;
```

1) We can define the values as variables

```
    const html = `  
<article class="country">  
    
  <div class="country__data">  
    <h3 class="country__name">${name}</h3>  
    <h4 class="country__region">${region}</h4>  
    <p class="country__row"><span>👤</span>${(+data.population / 1000000).toFixed(1)} people</p>  
    <p class="country__row"><span>🗣️</span>${language}</p>  
    <p class="country__row"><span>💰</span>${currency}</p>  
  </div>  
</article>  
  `;
```

2) Insert the values into the html

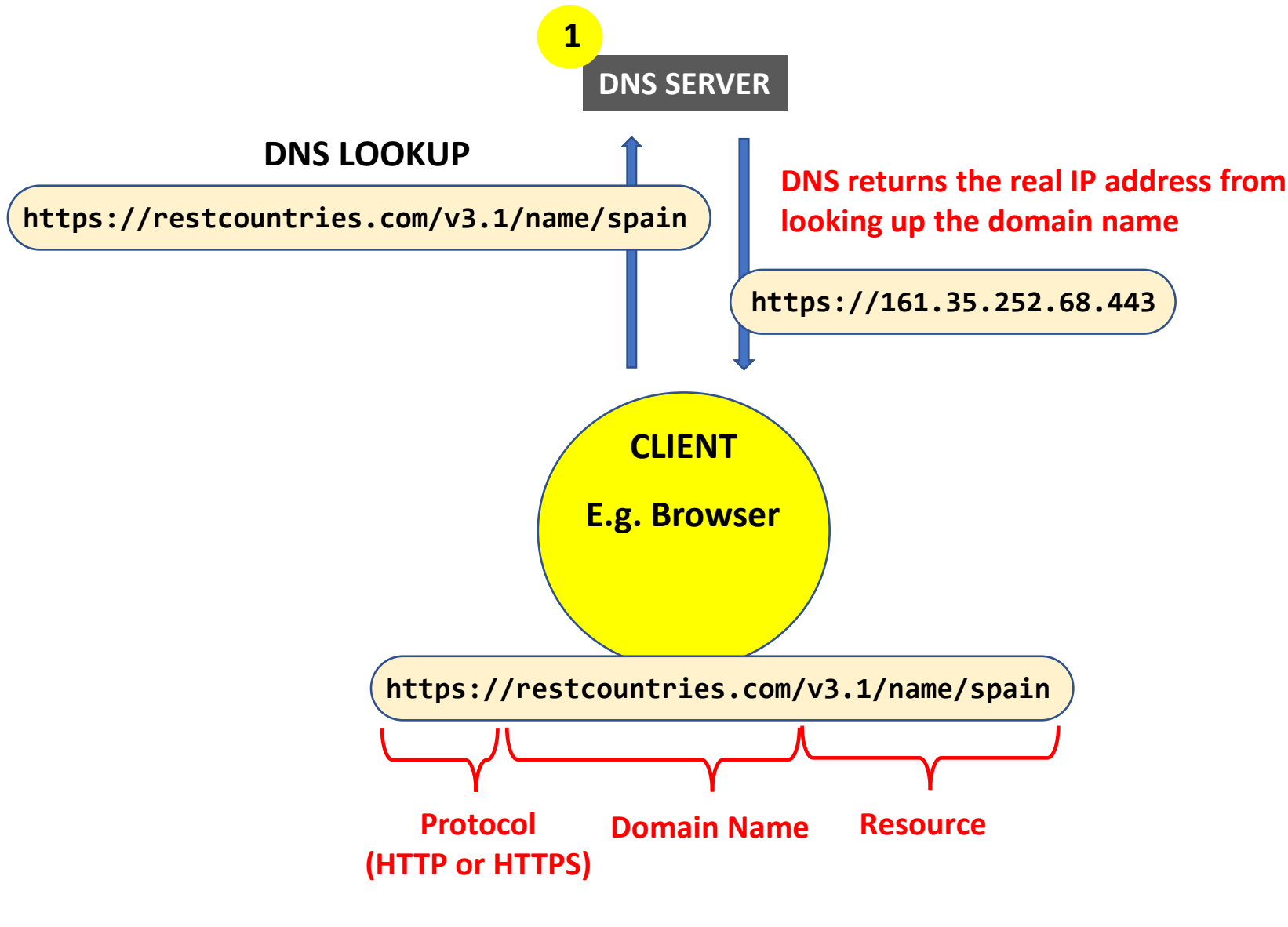
```
    countriesContainer.insertAdjacentHTML('beforeend', html);  
    countriesContainer.style.opacity = 1;
```

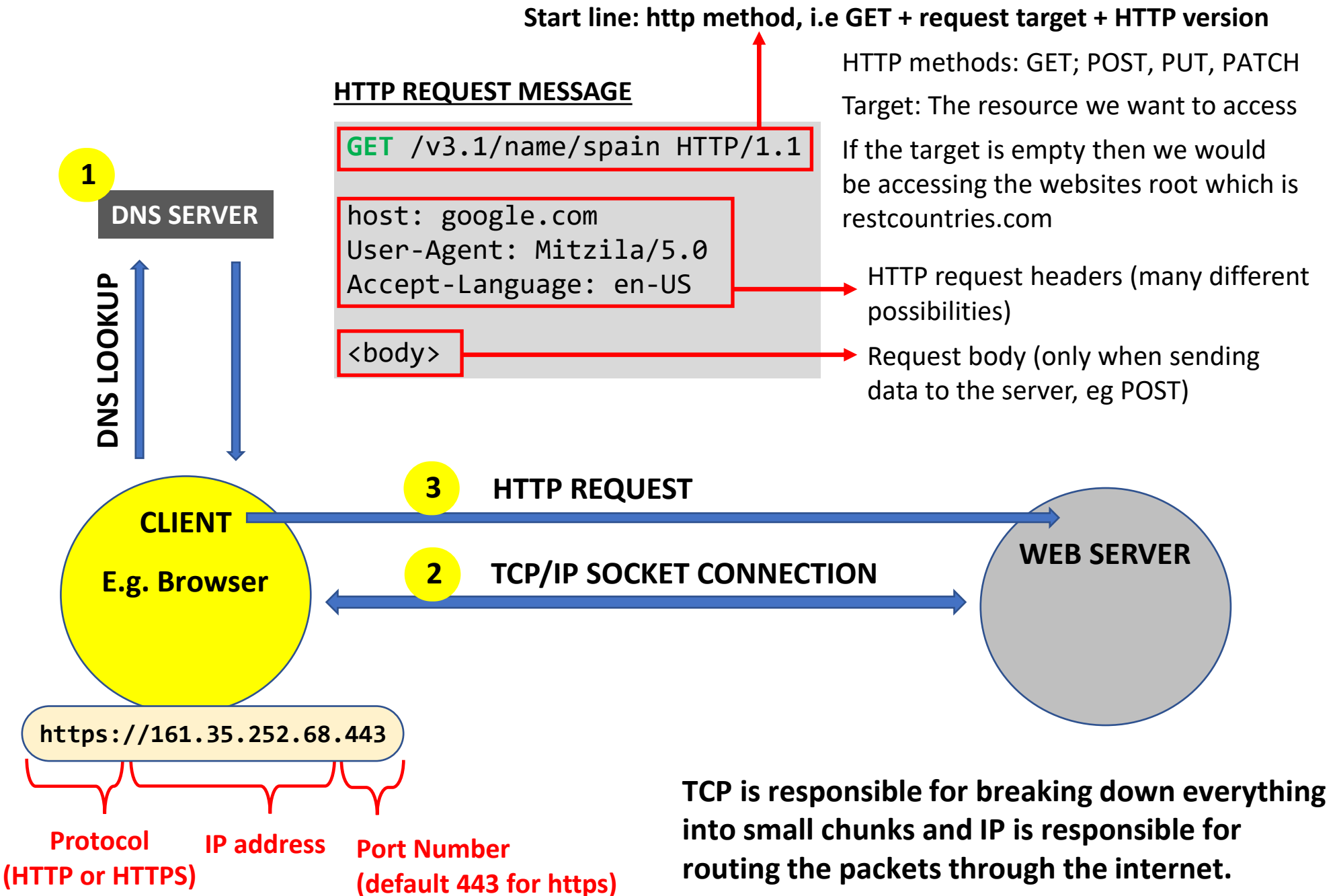
4) And call the function however many times we want to use it.

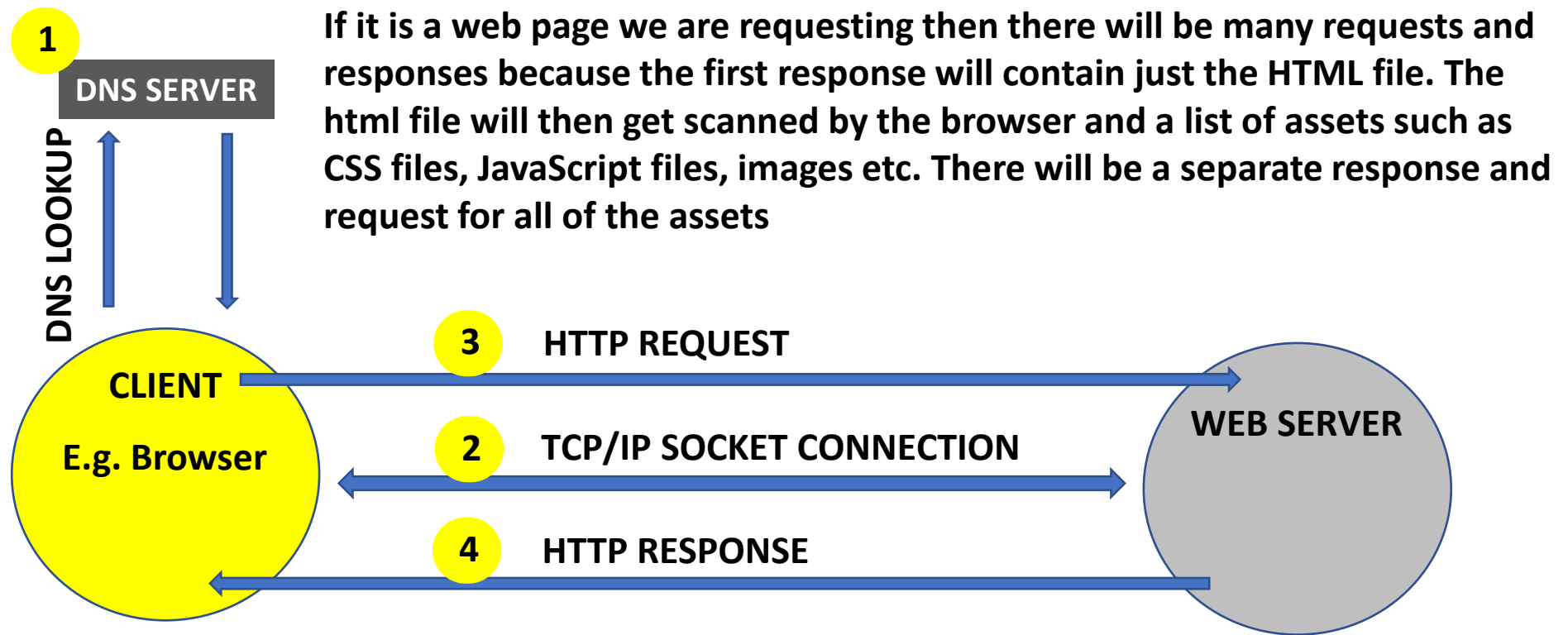
```
getCountryData('spain');  
getCountryData('andorra');
```


How the Web Works: Request & Response

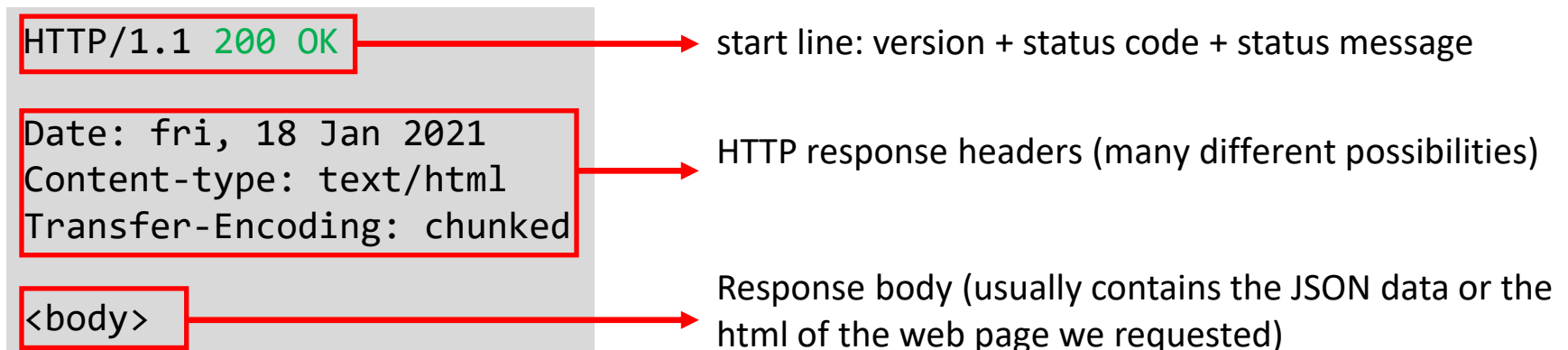
Request-response model or
Client-Server architecture







HTTP RESPONSE MESSAGE



Welcome to Call-back Hell

In the last lesson we created a simple AJAX call to get countries data from an API but we could not control which API call completed first and loaded. Both AJAX calls were running in parallel and the first one to complete would be the one that loaded first.

```
const renderCountry = function (data, className = '') {
  const name = data.name.common;
  const flag = data.flags.svg;
  const region = data.region;
  const language = Object.values(data.languages)[0];
  const currency = Object.values(data.currencies)[0].name;

  const html = `
    <article class="country ${className}">
      
      <div class="country__data">
        <h3 class="country__name">${name}</h3>
        <h4 class="country__region">${region}</h4>
        <p class="country__row"><span>👤</span>${(+data.population / 1000000).toFixed(1)} people</p>
        <p class="country__row"><span>🗣️</span>${language}</p>
        <p class="country__row"><span>💰</span>${currency}</p>
      </div>
    </article>
  `;

  countriesContainer.insertAdjacentHTML('beforeend', html);
  countriesContainer.style.opacity = 1;
};
```

1) The part of the code that renders the country is passed into its own function so it can be reused multiple times.

```
const getCountryAndNeighbour = function (country) {  
  const request = new XMLHttpRequest();  
  request.open('GET', `https://restcountries.com/v3.1/name/${country}`);  
  request.send();
```

2) The function is renamed to
getCountryAndNeighbour

```
request.addEventListener('load', function () {  
  const [data] = JSON.parse(this.responseText);  
  // console.log(data);  
  renderCountry(data);
```

```
  // get neighbour
```

```
  const [neighbour] = data.borders;
```

```
  if (!neighbour) return;
```

```
  // ajax call for neighbour
```

```
  const request2 = new XMLHttpRequest();
```

```
  request2.open(  
    'GET',  
    `https://restcountries.com/v3.1/alpha/${neighbour}`  
  );
```

```
};
```

```
request2.send();
```

```
request2.addEventListener('load', function () {  
  const [data2] = JSON.parse(this.responseText);
```

```
  //render country:
```

```
  renderCountry(data2, 'neighbour');
```

```
});
```

```
});
```

```
};
```

```
getCountryAndNeighbour('spain');
```

3) The first XMLHttpRequest gets the
country data including the neighbours.

4) Within the first callback there is
a nested callback.

The nested callback will not
execute until the first call back has
finished execution fully i.e. the
response has been received and
the render country is complete.

This ensures that the first country
is loaded first then the second
country however can become
callback hell if there are multiple
nested items.

```
setTimeout(() => {  
  console.log('1 second passed');  
  setTimeout(() => {  
    console.log('2 seconds passed');  
    setTimeout(() => {  
      console.log('3 second passed');  
      setTimeout(() => {  
        console.log('4 second passed');  
      }, 1000);  
    }, 1000);  
  }, 1000);  
}, 1000);
```

This makes for messy code, hard to maintain and hard to understand – bad code!

Another simplified example of callback hell or nested callbacks would be a set timeout nested within a set timeout like a Russian doll.

Callbacks within call backs are used to make asynchronous tasks work in order, i.e in a synchronous manner.

Promises and the fetch API (ES6 - since 2015)

```
// const request = new XMLHttpRequest();  
// request.open('GET', `https://restcountries.com/v3.1/name/${country}`);  
// request.send();
```

```
const request = fetch('https://restcountries.com/v3.1/name/portugal');  
console.log(request);
```

The above code of a request, then open then send can be replaced with a fetch.

```
▼ Promise {<pending>} ⓘ  
  ► [[Prototype]]: Promise  
  ► [[PromiseState]]: "fulfilled"  
  ► [[PromiseResult]]: Response
```

PROMISE

1. **Promise:** An object that is used as a placeholder for the future result of an asynchronous operation.
↓ Less formal
2. **Promise:** A container for an asynchronous delivery value.
↓ Less formal
3. **promise:** A container for a future value. ← **Example: response from an 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 by lottery ticket (promise) right now

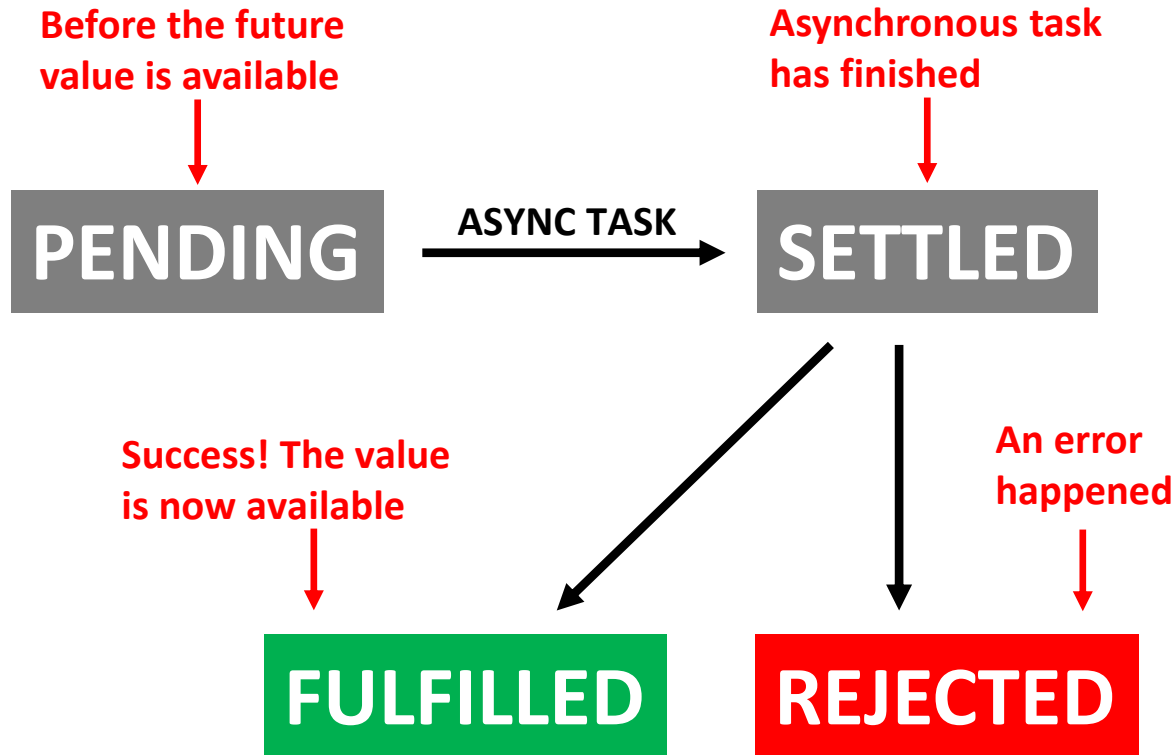


Lottery draw happens asynchronously



If correct outcome, I receive money, because I was promised

The Promise Lifecycle



- We are able to **handle** these different states in our code
- Promises are **only settled once**
- The promise is either fulfilled or rejected but we are **unable to change this state**

In order for a promise to be consume we must first **build** the promise. e.g. fetch API returns a promise.

**BUILD
PROMISE**

**CONSUME
PROMISE**

When we use a promise the get a result we **consume** a promise.

Consuming Promises

```
const getCountryData = function (country) {  
  fetch(`https://restcountries.com/v3.1/name/${country}`)  
    .then(function (response) {  
      console.log(response);  
    });  
};
```

```
getCountryData('portugal');
```

The fetch immediately returns an empty promise. We can use the .then method on the fetch and create a callback function, in this example named response.

Note the response message is a http message.

```
Response {type: 'cors', url: 'https://restcountries.com/v3.1/name/portugal', redirected: false, status: 200, ok: true, ...} t  
  body: (...)  
  bodyUsed: false  
  headers: Headers {}  
  ok: true  
  redirected: false  
  status: 200  
  statusText: "OK"  
  type: "cors"  
  url: "https://restcountries.com/v3.1/name/portugal"  
  [[Prototype]]: Response
```

```
const getCountryData = function (country) {  
  fetch(`https://restcountries.com/v3.1/name/${country}`)  
    .then(function (response) {  
      return response.json();  
    })  
    .then(function (data) {  
      console.log(data);  
    });  
};
```

```
getCountryData('portugal');
```

We can use the .json method on the response. Note that the .json method also is a promise so we need to return it.

Now we can use the .then method again on the .json response to add another function for the data.

```
{...} t  
  0: {name: {...}, tld: Array(1), cca2: 'PT', ccn3: length: 1}  
  [[Prototype]]: Array(0)
```

```
const getCountryData = function (country) {  
  fetch(`https://restcountries.com/v3.1/name/${country}`)  
    .then(function (response) {  
      return response.json();  
    })  
    .then(function (data) {  
      console.log(data);  
      renderCountry(data[0]);  
    });  
};
```

Now all we have to do is renderCountry(Data[0]
using the function previously created!

```
getCountryData('portugal');
```

```
const getCountryData = function (country) {  
  fetch(`https://restcountries.com/v3.1/name/${country}`)  
    .then(response => response.json())  
    .then(data => renderCountry(data[0]));  
};
```

If we remove the console logs,
we can convert these to arrow
functions to make a very tidy
simple and clean block of code.

```
getCountryData('portugal');
```

Chaining Promises

```
const getCountryData = function (country) {  
  fetch(`https://restcountries.com/v3.1/name/${country}`)  
    .then(response => response.json())  
    .then(data => {  
      renderCountry(data[0]);  
      const neighbour = data[0].borders[0];  
  
      if (!neighbour) return;  
  
      return fetch(`https://restcountries.com/v3.1/alpha/${neighbour}`);  
    })  
    .then(response => response.json())  
    .then(data => renderCountry(data[0], 'neighbour'));  
};  
  
getCountryData('portugal');
```

From the results of the first promise we can get the neighbour.

And we can return a second fetch within the first one.

And use the json method on the second promise.

And call the renderCountry method again.

```
const getCountryData = function (country) {  
  fetch(`https://restcountries.com/v3.1/name/${country}`)  
    .then(response => response.json())  
    .then(data => {  
      renderCountry(data[0]);  
      const neighbour = data[0].borders[0];  
  
      if (!neighbour) return;  
      return fetch(`https://restcountries.com/v3.1/alpha/${neighbour}`).then(  
        response => response.json()  
      );  
    })  
    .then(data => renderCountry(data[0], 'neighbour'));  
};  
  
getCountryData('portugal');
```



A common mistake is to call the then method on the second fetch before returning it. This will create nested callbacks and we are back in callback hell again!

Always close the first promise before the .then of the second promise.

Handling Rejected Promises

```
btn.addEventListener('click', function () {  
  getCountryData('portugal');  
});
```

The most common failure for promises is that the internet connection is temporarily lost when the data is being sent to the browser. To simulate this we can trigger the fetch from an event listener on a button and set the network status to offline on the browser.

The screenshot shows a web browser window with a page titled 'Asynchronous JavaScript'. The page displays information for Portugal and Spain, including flags, names, and some statistics. A button labeled 'Where am I?' is at the bottom. The browser's developer tools are open, showing the Network tab. A dropdown menu is open, showing the 'Offline' status. The console shows an error: 'Uncaught (in promise) TypeError: Failed to fetch'.

Name	Status	Type	Initiator	Size	T..	Waterfall
16-Asynchronous-JavaScript-Promises-A...	304	document		296 B	2...	
style.css	304	stylesheet	(index)	296 B	2...	
script.js	200	script	(index)	8.0 kB	2...	
ws	101	websocket	(index):39	0 B	P...	
portugal	200	fetch	script.js:238	1.6 kB	2...	
ESP	200	fetch	script.js:246	1.7 kB	2...	
pt.svg	200	svg+xml	Other	(disk cac...	1...	

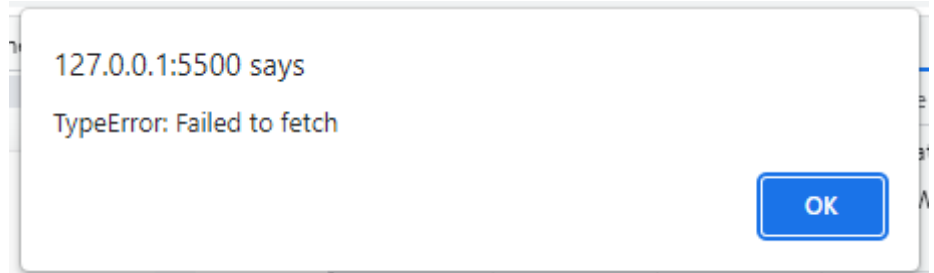
```
✖ ▶ Uncaught (in promise) TypeError: Failed to fetch  
  at getCountryData (script.js:238:3)  
  at HTMLButtonElement.<anonymous> (script.js:253:3)
```

Now we have an uncaught promise that has a state of rejected.

There are two ways of handling rejected promises:

```
const getCountryData = function (country) {  
  fetch(`https://restcountries.com/v3.1/name/${country}`)  
    .then(  
      response => response.json(),  
      err => alert(err) ←  
    )  
    .then(data => {  
      renderCountry(data[0]);  
      const neighbour = data[0].borders[0];  
  
      if (!neighbour) return;  
  
      return fetch(`https://restcountries.com/v3.1/alpha/${neighbour}`);  
    })  
    .then(response => response.json())  
    .then(data => renderCountry(data[0], 'neighbour'));  
};
```

1) We can pass a second callback function into the then method which will be called with the error argument itself.



```
btn.addEventListener('click', function () {  
  getCountryData('portugal');  
});
```

```
✖ Uncaught (in promise) TypeError: Cannot read properties of undefined (reading '0')  
  at script.js:244:25  
    (anonymous) @ script.js:244  
    Promise.then (async)  
    getCountryData @ script.js:252  
    (anonymous) @ script.js:256
```

Now we are handling the error

```
const getCountryData = function (country) {  
  fetch(`https://restcountries.com/v3.1/name/${country}`)  
    .then(  
      response => response.json(),  
      err => alert(err) ← We would need to handle the error for both  
                           fetches using this method.  
    )  
    .then(data => {  
      renderCountry(data[0]);  
      const neighbour = data[0].borders[0];  
  
      if (!neighbour) return;  
  
      return fetch(`https://restcountries.com/v3.1/alpha/${neighbour}`);  
    })  
    .then(  
      response => response.json(),  
      err => alert(err) ←  
    )  
    .then(data => renderCountry(data[0], 'neighbour'));  
};  
  
btn.addEventListener('click', function () {  
  getCountryData('portugal');  
});
```

There is a better way of handling all the errors in one place.

```
const renderError = function (msg) {
  countriesContainer.insertAdjacentText('beforeend', msg);
  countriesContainer.style.opacity = 1;
};
```

4) A function that takes a message as a parameter and inserts it into the html document.

```
const getCountryData = function (country) {
  fetch(`https://restcountries.com/v3.1/name/${country}`)
    .then(response => response.json())
    .then(data => {
      renderCountry(data[0]);
      const neighbour = data[0].borders[0];

      if (!neighbour) return;

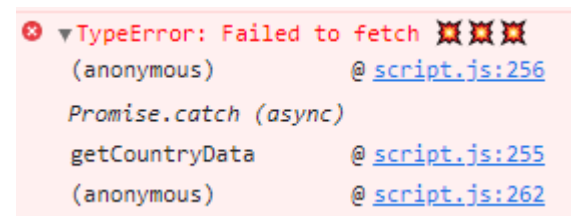
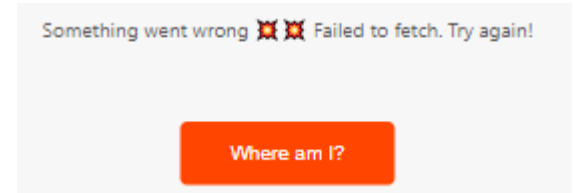
      return fetch(`https://restcountries.com/v3.1/alpha/${neighbour}`);
    })
    .then(response => response.json())
    .then(data => renderCountry(data[0], 'neighbour'))
    .catch(err => {
      console.error(`${err} ❌❌❌`);
      renderError(`Something went wrong ❌❌ ${err.message}. Try again!`);
    });
};
```

1) At the end of the promise chain we add a method called catch which is only executed if the promise is rejected.

2) Console terminal has built in functionality to display custom error messages

3) Construct the message string and call the render error function

```
btn.addEventListener('click', function () {
  getCountryData('portugal');
});
```




```

const getCountryData = function (country) {
  fetch(`https://restcountries.com/v3.1/name/${country}`)
    .then(response => response.json())
    .then(data => {
      renderCountry(data[0]);
      const neighbour = data[0].borders[0];

      if (!neighbour) return;

      return fetch(`https://restcountries.com/v3.1/alpha/${neighbour}`);
    })
    .then(response => response.json())
    .then(data => renderCountry(data[0], 'neighbour'))
    .catch(err => {
      console.error(`${err} ❌❌❌`);
      renderError(`Something went wrong ❌❌ ${err.message}. Try again!`);
    })
    .finally(() => {
      countriesContainer.style.opacity = 1;
    });
};

```



The finally method call back function will always be called regardless of whether the promise is fulfilled or rejected. This can be useful for tasks such as removing a spinning loading gif. In this example we use it to add opacity to the countries container. Because we are handling this action with a finally, we can remove it from the renderCountry and renderError functions.

Throwing Errors Manually

```
getCountryData( 'gdh' );
```

Something went wrong 🚨🚨 Cannot read properties of undefined (reading 'name'). Try again!

Where am I?

```
✖ GET https://restcountries.com/v3.1/name/gdh 404 (Not Found)
  getCountryData @ script.js:241
  (anonymous) @ script.js:266

✖ ▼ TypeError: Cannot read properties of undefined (reading 'name') 🚨🚨🚨
  (anonymous) @ script.js:254
  Promise.catch (async)
  getCountryData @ script.js:253
  (anonymous) @ script.js:266
```

In this scenario I am trying to find a country that does not exist. The actual error message that is handled in the catch method is that the API cannot find flag for that country because flag is the first item we are trying to deconstruct from the returned API object. But the real issue is that the API has returned a 404 not found which is more along the lines of what we want to log to the user as an error message i.e. country not found 404.

The promise does not get rejected straight away upon receipt of a 4xx response but continues to run through all the methods till it gets to the catch and returns the error of cannot read properties of undefined.

We can solve this issue by manually throwing the error at the first then method.

```
const getCountryData = function (country) {  
  fetch(`https://restcountries.com/v3.1/name/${country}`)  
    .then(response => {  
      console.log(response);  
      return response.json();  
    })  
    .then(data => {  
      renderCountry(data[0]);  
      const neighbour = data[0].borders[0];  
    })  
}
```

1) At the first then method we expand the simplified arrow function with curly braces to console.log the response. We also now have to explicitly return the response.

```
▼ Response {type: 'cors', url: 'https://restcountries.com/v3.1/name/gdh', redirected: false, status: 404, o  
  k: false, ...} ⓘ  
  body: (...)  
  bodyUsed: true  
  ▶ headers: Headers {}  
  ok: false  
  redirected: false  
  status: 404  
  statusText: "Not Found"  
  type: "cors"  
  url: "https://restcountries.com/v3.1/name/gdh"  
  ▶ [[Prototype]]: Response
```

2) The response status is Ok: false, status: 404

```

const getCountryData = function (country) {
  fetch(`https://restcountries.com/v3.1/name/${country}`)
    .then(response => {
      console.log(response);

      if (!response.ok)
        throw new Error(`Country not found (${response.status})`);

      return response.json();
    })
    .then(data => {
      renderCountry(data[0]);
      const neighbour = data[0].borders[0];

      if (!neighbour) return;

      return fetch(`https://restcountries.com/v3.1/alpha/${neighbour}`);
    })
    .then(response => response.json())
    .then(data => renderCountry(data[0], 'neighbour'))
    .catch(err => {
      console.error(`${err} ❌❌❌`);
      renderError(`Something went wrong ❌❌ ${err.message}. Try again!`);
    })
    .finally(() => {
      countriesContainer.style.opacity = 1;
    });
};

```

3) We can now write a custom error handler that creates a new Error message if the response is not OK.



4) If the response is not 200 then this code will cause the promise to be rejected here at this stage of code and propagate the error down to the catch where it is passed to the renderError function.

Something went wrong ❌❌ Country not found (404). Try again!

Where am I?

```

const getCountryData = function (country) {
  fetch(`https://restcountries.com/v3.1/name/${country}`)
    .then(response => {

      if (!response.ok)
        throw new Error(`Country not found (${response.status})`);

      return response.json();
    })
    .then(data => {
      renderCountry(data[0]);
      // const neighbour = data[0].borders[0];
      const neighbour = 'gdgd';

      if (!neighbour) return;

      return fetch(`https://restcountries.com/v3.1/alpha/${neighbour}`);
    })
    .then(response => response.json())
    .then(data => renderCountry(data[0], 'neighbour'))
    .catch(err => {
      console.error(`${err} ❌❌❌`);
      renderError(`Something went wrong ❌❌ ${err.message}. Try again!`);
    })
    .finally(() => {
      countriesContainer.style.opacity = 1;
    });
};

```

5) But lets say that the country requested is valid but the API returns a neighbour country that does not exist.



6) The second promise does not get rejected but the code runs down to the catch and displays the 4xx incorrectly.

```

const getCountryData = function (country) {
  fetch(`https://restcountries.com/v3.1/name/${country}`)
    .then(response => {
      if (!response.ok) throw new Error(`Country not found (${response.status})`);

      return response.json();
    })
    .then(data => {
      renderCountry(data[0]);
      // const neighbour = data[0].borders[0];
      const neighbour = 'gdgd';

      if (!neighbour) return;

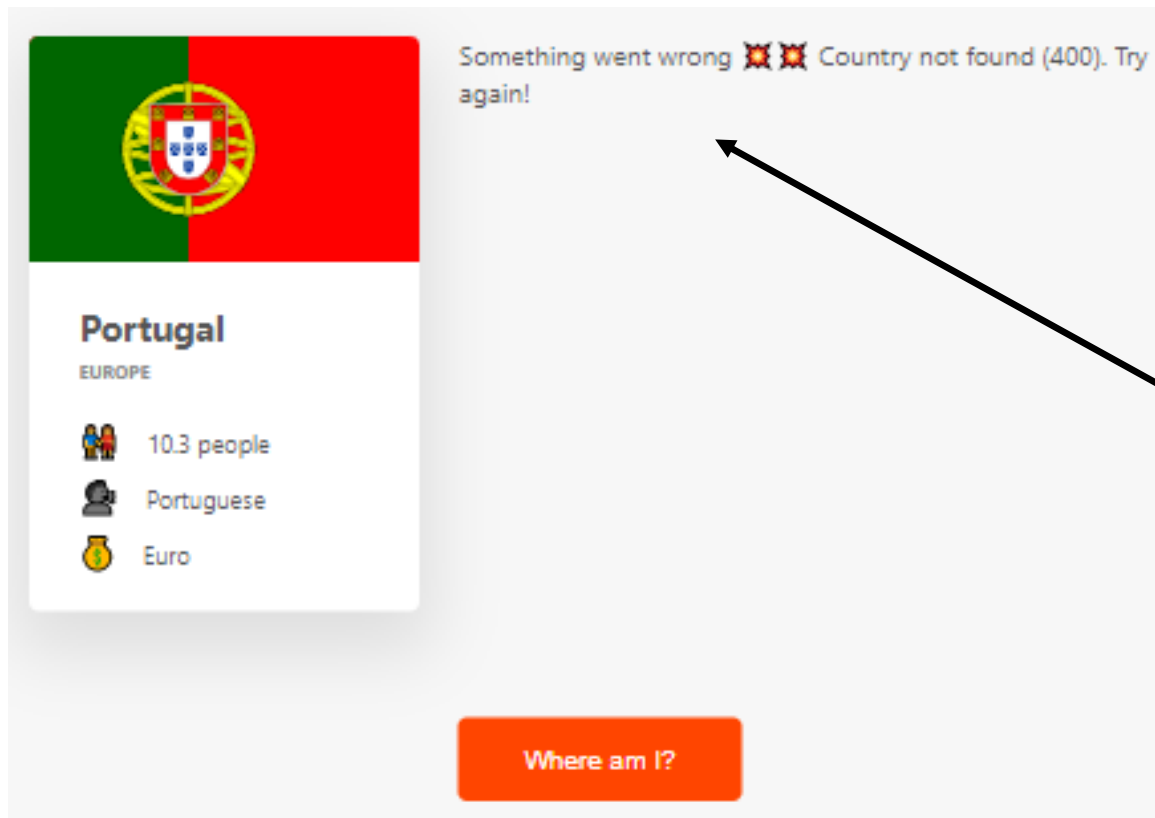
      return fetch(`https://restcountries.com/v3.1/alpha/${neighbour}`);
    })
    .then(response => {
      if (!response.ok) throw new Error(`Country not found (${response.status})`);

      return response.json();
    })
    .then(data => renderCountry(data[0], 'neighbour'))
    .catch(err => {
      console.error(`${err} ❌❌❌`);
      renderError(`Something went wrong ❌❌ ${err.message}. Try again!`);
    })
    .finally(() => {
      countriesContainer.style.opacity = 1;
    });
};

```

7) The code to throw a manual error needs to be duplicated in the second promise





8) Now the correct error message with the 4xx status is displaying correctly.

9) Which is OK and we could leave it here as working code but we have broken the DRY principle because we have almost duplicate code for the manual error handling sections of the first then of both promises.

This would get really tedious if we had more than 2 promises in a chain.

The code can be refactored to put the promises into a separate, reusable function then call that function on both occasions.


```
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();
  });
};
```

10) The getJSON function takes a URL and errorMsg as parameters and returns the fetch, manual error throw and JSON.

11) Now we can call the getJSON function passing in url and errorMsg

```
const getCountryData = function (country) {
  getJSON(`https://restcountries.com/v3.1/name/${country}`, 'Country not found')
    .then(data => {
      renderCountry(data[0]);
      // const neighbour = data[0].borders[0];
      const neighbour = 'gdgd';

      if (!neighbour) return;

      return getJSON(
        `https://restcountries.com/v3.1/alpha/${neighbour}`, 'Country not found');
    })
    .then(data => renderCountry(data[0], 'neighbour'))
    .catch(err => {
      console.error(`${err} ❌❌❌`);
      renderError(`Something went wrong ❌❌ ${err.message}. Try again!`);
    })
    .finally(() => {
      countriesContainer.style.opacity = 1;
    });
};
```



Portugal

EUROPE



10.3 people



Portuguese



Euro

Something went wrong 🚨🚨 Country not found (400). Try again!



Something went wrong 🚨🚨 Country not found (404). Try again!



Where am I?

12) And now it works in the two simulated failure scenarios and the success scenario.



Portugal

EUROPE



10.3 people



Portuguese



Euro



Spain

EUROPE



47.4 people



Spanish



Euro

Where am I?



Australia

OCEANIA



25.7 people



English



Australian dollar

Something went wrong 🚨🚨 Cannot read properties of undefined (reading '0'). Try again!

```
if (!neighbour) return;
```

13) But now we need to handle the case that there is a country that has no neighbours. At the moment all our code does is return but a custom message on screen would be more helpful.

Where am I?

```

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) {
  getJSON(`https://restcountries.com/v3.1/name/${country}`, 'Country not found')
    .then(data => {
      renderCountry(data[0]);
      const neighbour = data[0].borders;
      // const neighbour = 'gdgd';

      if (!neighbour) throw new Error('No neighbour found!');

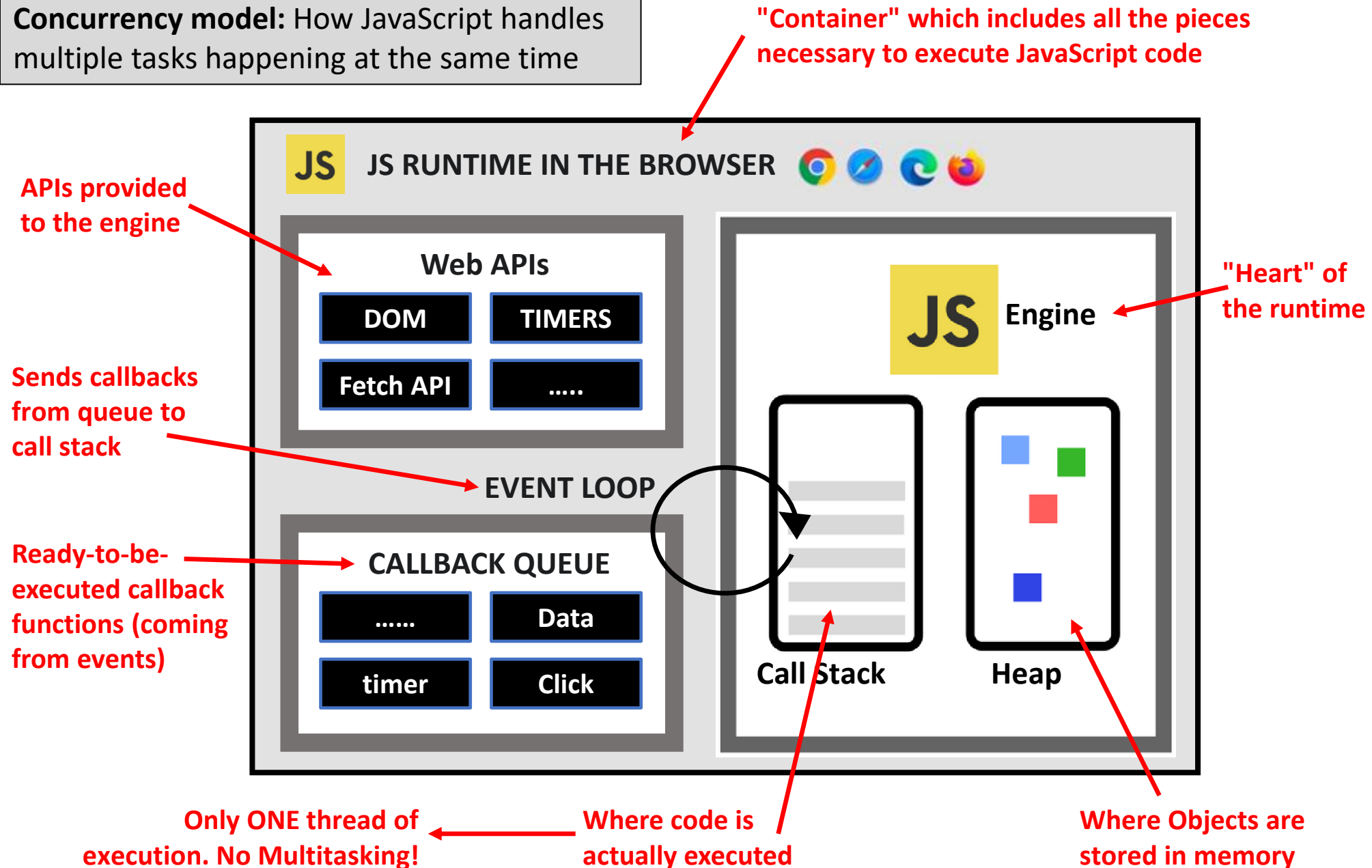
      return getJSON(
        `https://restcountries.com/v3.1/alpha/${neighbour}`,
        'Country not found'
      );
    })
    .then(data => renderCountry(data[0], 'neighbour'))
    .catch(err => {
      console.error(`${err} ❌❌❌`);
      renderError(`Something went wrong ❌❌ ${err.message}. Try again!`);
    })
    .finally(() => {
      countriesContainer.style.opacity = 1;
    });
};

```

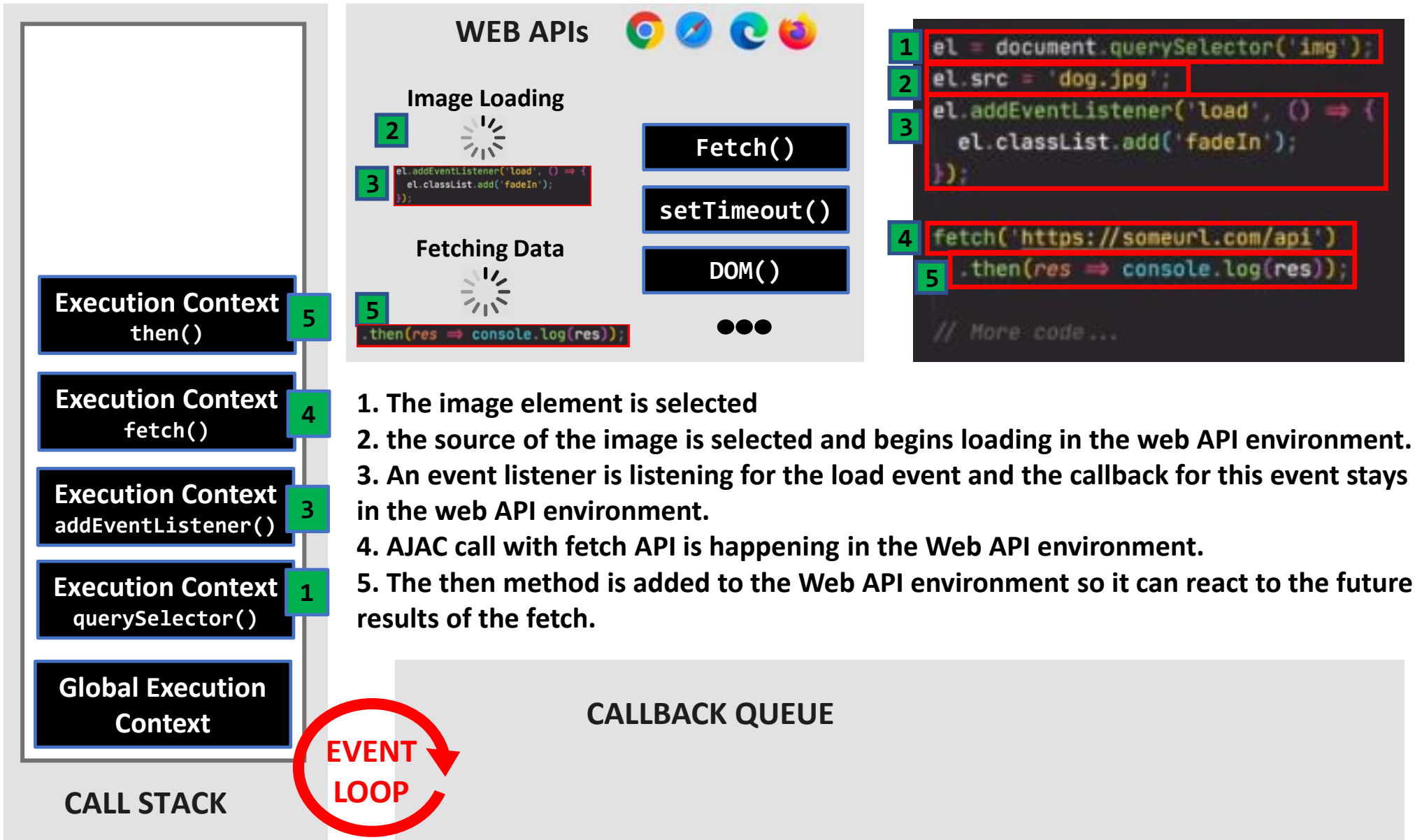
14) Instead of returning we can throw new Error with a custom message which will get propagated down the catch.

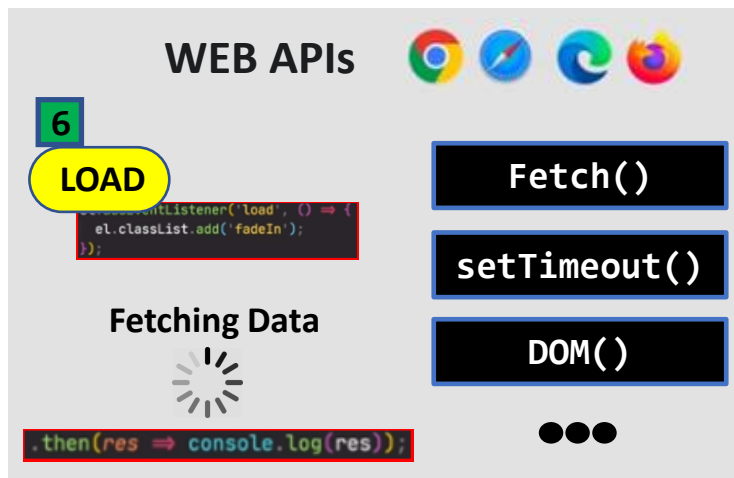
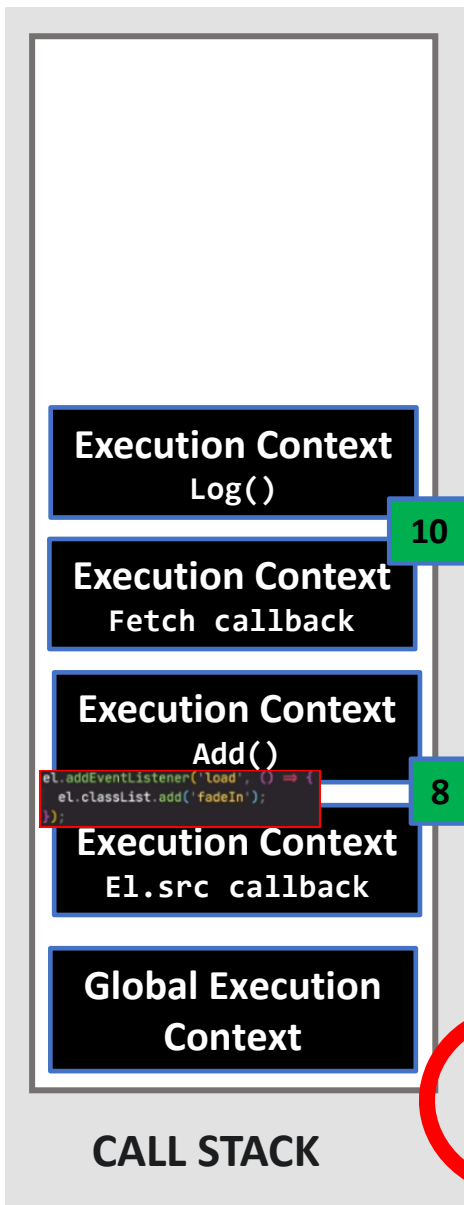
Asynchronous Behind the Scenes: The Event Loop

Concurrency model: How JavaScript handles multiple tasks happening at the same time

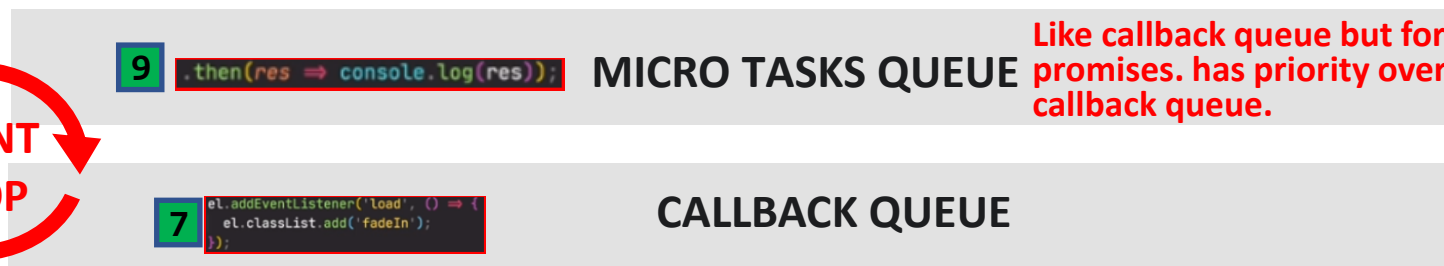


How can asynchronous code be executed in a non-blocking way, if there is only one thread of execution in the engine?





6. The image has finished loading and the load event is emitted
7. The image callback is put in the callback queue waiting in line to be executed in the call stack.
8. The event loop is looking at the call stack and when it is empty (except for the global execution context) it will load the next item in the call back queue into the call stack for execution. This is called and event loop tick – the dequeuing of an item from the callback queue.
9. The fetch is done and the data has arrived. The callback is coming from a promise which has a special queue called the microtasks queue. Tasks in the microtasks queue have priority over tasks in the callback queue.
10. MicroTask for the promise response is executed in the call stack.



Asynchronous tasks run in the Web API environment of the browser. If they ran in the call stack they would be blocking.

Micro Tasks always have priority over the call back queue. So if new tasks are added the microtask queue then the call back queue could potentially be starved.

The Event Loop in practice

```
console.log('Test start');
setTimeout(() => console.log('0 sec timer'), 0);
Promise.resolve('Resolved promise 1').then(res =>
console.log(res));

Promise.resolve('Resolved promise 2').then(res => {
  for (let i = 0; i < 1000000000; i++) {}
  console.log(res);
});

console.log('Test end');
```

Test start

Test end

Resolved promise 1

Resolved promise 2

0 sec timer

- 1) Test start is executed first because it goes straight to the call stack due to not having any call back event.
- 2) Test End is second in the queue because it is later on in the code than "Test start" and also does not have any call back events.
- 3) "Promise 1" is put in the microtasks queue rapidly because it executes a setTimeout after 0 seconds. It has priority over the call back queue.
- 4) "Promise 2" has a for loop that counts to a very large number before being put in the microTasks queue and priority loaded into the call stack.
- 5) "0 sec timer" is loaded last because it is a non promise callback which goes in the non priority call back loop.

Building a Simple Promise

A fulfilled promise means to win the lottery
a rejected promise is to loose the lottery.

The Promise takes 1 argument and that is the executor function which runs automatically when the promise is called. The new promise is stored in lotteryPromise variable

```
const lotteryPromise = new Promise(function (resolve, reject) {
```

The executor function takes two arguments of resolve and reject.

```
if (Math.random() >= 0.5) {
```

```
    resolve('You WIN 💰');
```

```
} else {
```

```
    reject('You lost your money 💩');
```

```
};
```

Lets say we win in 50% of cases so we can simply generate a random number between zero and 1.

If the random number is greater than 0.5 we want to call the resolve function and the promise is marked as fulfilled. Else we call the reject function and mark the promise as rejected.

The new promise calls an executor function with resolved or rejected functions so that the promise will end up in either one of these two states.

Now we will CONSUME the promise with a '.then' on the lotteryPromise object. The then method needs a callback function which is going to be called with the resolved value of the promise.

```
lotteryPromise.then(res => console.log(res)).catch(err => console.error(err));
```

We also need to catch any errors and we can log it to the console as an error.

↓
You WIN 💰

↓
❌ You lost your money 💩

However the simple lottery promise is not asynchronous because it runs in real time so lets at a simple timeoutFunction. This is how we encapsulate any asynchronous behaviour in a promise.

```
const lotteryPromise = new Promise(function (resolve, reject) {
```

This will log to the console immediately because it is outside of the timeout function.

```
console.log('Lottery draw is happening 🎰');
```

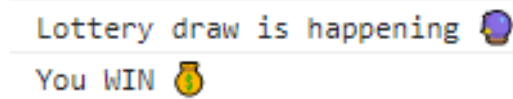
The promise is now encapsulated in a timeoutFunction with a 2000ms delay.

```
setTimeout(function () {  
  if (Math.random() >= 0.5) {  
    resolve('You WIN 💰');  
  } else {  
    reject(new Error('You lost your money 💩'));  
  }  
}, 2000);  
});
```

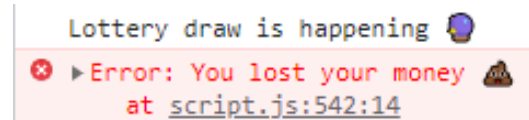
The promise is Consumed outside of the timeout function.

```
lotteryPromise.then(res => console.log(res)).catch(err => console.error(err));
```

Lottery draw is happening is logged to the console immediately than after 2 seconds the promise fulfilled or rejected is logged.



```
Lottery draw is happening 🎰  
You WIN 💰
```



```
✖ Error: You lost your money 💩  
  at script.js:542:14
```

Most of the time we only consume promises. We only build promises to wrap old callback asynchronous behaviour into promise based asynchronous behaviour. This is called promisifying.

We can promisify the setTimeout function and create a wait function that will take in a number of seconds.

```
const wait = function (seconds) {  
  return new Promise(function (resolve) {  
    setTimeout(resolve, seconds * 1000);  
  });  
};
```

From within the function we return a new promise like a fetch does.

The executor of the promise does not even need a reject function because it is impossible for the timer to fail so it does not need the reject parameter specified.

We use SetTimeout and the callback function we want to be called after a certain time is the resolve function but we are not going to pass any value into the resolve function because it is not necessary. All we want to do is make our code wait so we run the timer for a certain amount of seconds.

```
wait(1)  
  .then(() => {  
    console.log('1 second passed');  
    return wait(1);  
  })  
  .then(() => {  
    console.log('2 seconds passed');  
    return wait(1);  
  })  
  .then(() => {  
    console.log('3 seconds passed');  
    return wait(1);  
  })  
  .then(() => console.log('4 seconds passed'));
```

Now we consume the promise by calling the wait function and with the then method we log time waited and return a new promise of wait(1).

We are chaining multiple wait's and then's with each one returning a new promise.

We have a nice chain of sequential asynchronous behaviour without callback hell.

```
1 second passed  
2 seconds passed  
3 seconds passed  
4 seconds passed
```

We can create a fulfilled or rejected promise immediately.

```
Promise.resolve('abc').then(x => console.log(x));  
Promise.reject(new Error('Problem!')).catch(x => console.error(x));
```

abc

✖ ▶ Error: Problem!
at script.js:584:16

Promisify the Geolocation API

```
navigator.geolocation.getCurrentPosition(  
  position => console.log(position),  
  err => console.log(err)  
);  
console.log('Getting position');
```

We used `navigator.geolocation.getCurrentPosition` which accepts two callbacks. The first actually gets action to the position object. The second callback has the error in case the user blocks access to browser location.

This is asynchronous behaviour because the code is not blocking while it is finding the position, we can even log to the console “getting position” while it is in the background (the web API environment) finding the position.

Getting position

► `GeolocationPosition {coords: GeolocationCoordinates, timestamp: 1653573210721}`

```
const getPosition = function () {  
  console.log('Getting position');
```

Now to promisify the callback based API

```
  return new Promise(function (resolve, reject) {  
    navigator.geolocation.getCurrentPosition(  
      position => resolve(position),  
      err => reject(err)  
    );  
  });  
};  
getPosition().then(pos => console.log(pos));
```

Getting position

► `GeolocationPosition {coords: GeolocationCoordinates, timestamp: 1653573801043}`

```
const getPosition = function () {  
  console.log('Getting position');
```

```
  return new Promise(function (resolve, reject) {  
    navigator.geolocation.getCurrentPosition(resolve, reject);  
  });  
};
```

We can simplify it because the resolve is automatically passing in the position and the reject is automatically passing in the error.

```
getPosition().then(pos => console.log(pos));
```

Getting position

► *GeolocationPosition {coords: GeolocationCoordinates, timestamp: 1653573994749}*

```
const getPosition = function () {  
  return new Promise(function (resolve, reject) {  
    navigator.geolocation.getCurrentPosition(resolve, reject);  
  });  
};
```

```
const whereAmI = function () {  
  getPosition().then(pos => {  
    console.log(pos.coords);  
  });  
};
```

*GeolocationCoordinates {latitude: 41.4941184, longitude: 2.12992,
 ► altitude: null, accuracy: 2971.657996266222, altitudeAccuracy: nul
 l, ...}*

```

const whereAmI = function () {
  getPosition()
    .then(pos => {
      We can de-structure the pos.coords object
      const { latitude: lat, longitude: lng } = pos.coords;
      return fetch(`https://geocode.xyz/${lat},${lng}?geoit=json`);
    })
    And return them in the fetch
    .then(res => {
      if (!res.ok) throw new Error(`Problem with geocoding ${res.status}`);
      return res.json();
    })
    .then(data => {
      console.log(data);
      console.log(`You are in ${data.city}, ${data.country}`); You are in Barcelona, Spain

      return fetch(`https://restcountries.com/v3.1/name/${data.country}`);
    })
    .then(res => {
      if (!res.ok) throw new Error(`Country not found (${res.status})`);

      return res.json();
    })
    .then(data => renderCountry(data[0]))
    .catch(err => console.error(`${err.message} ❌`))
    .finally(() => {
      countriesContainer.style.opacity = 1;
    });
};

btn.addEventListener('click', whereAmI);

```

Consuming Promises with Async/Await (ES2017)

There is now an even better way to consume promises called `async/await`.

```
const whereAmI = async function (country) {
```

The `whereAmI` function takes a `country` parameter. Note that the function is prefixed with **async**. This means that the function is automatically an asynchronous function that runs in the background and returns a promise.

```
    const res = await fetch(`https://restcountries.com/v3.1/name/${country}`);
```

We use `await` to stop the code execution on this line while the function is awaiting the response from the promise. But is this blocking the execution?

No. Stopping execution in an `async` function is not a problem because it is running in the background and not the main execution thread of the call stack.

As soon as the promise is resolved the whole of the value of the variable `"res"` is the response.

```
    console.log(res);  
};
```

We can test this by console logging the response from within the function. We can also call the function and then console log `'first'`.

```
whereAmI('portugal');  
console.log('First');
```

First

```
▶ Response {type: 'cors', url: 'https://restcountries.com/v3.1/name/portugal', redirected: false, status: 200, ok: true, ...}
```

We see that the `first` is displayed first while the `async` function is awaiting the response to the promise. This looks like normal synchronous code when it is asynchronously running. It is syntactic sugar over the `fetch()`, `then()`, `catch()` methods.


```
const whereAmI = async function (country) {  
  const res = await  
  fetch(`https://restcountries.com/v3.1/name/${country}`);  
  const data = await res.json();  
  renderCountry(data[0]);  
};  
whereAmI('portugal');
```

All we have to do is store the data in a variable and call the json() method on the response of the await.

This is a lot easier because we can store the response directly in a variable rather than chaining methods like then() and catch().

WE can now combine the aysnc/await with the get geo location API promise and reverse geocoding API to rewritre our chain of promises using then(), catch() and finally().

```
const getPosition = function () {
  return new Promise(function (resolve, reject) {
    navigator.geolocation.getCurrentPosition(resolve, reject);
  });
};
```

The creation of the promise for obtaining the geolocation from the browser remains the same.

```
const whereAmI = async function () {
  // Geolocation
  const pos = await getPosition();
  const { latitude: lat, longitude: lng } = pos.coords;
```

The whereAmI is now an async function which is awaiting the response of the promise from the getPosition function.

```
  // Reverse Geocoding
  const resGeo = await fetch(`https://geocode.xyz/${lat},${lng}?geoit=json`);
  const dataGeo = await resGeo.json();
  console.log(dataGeo);
```

We then chain an await to the API call to get location from coords.

```
  // Country Data
  const res = await fetch(
    `https://restcountries.com/v3.1/name/${dataGeo.country}`
  );
  const data = await res.json();
  console.log(data);
  renderCountry(data[0]);
};
whereAmI();
```

We then chain an await to the API call to the countries API from the response of the first API location call.

Now the whole app is in one nice clean async function which relies on the response of three promises to execute in sequence. But what about error handling?

Error handling with Try...Catch

```
let y = 1;  
const x = 2;  
x = 3;
```

✖ ▶ Uncaught TypeError: Assignment to constant variable.
at script.js:801:3

In this example a deliberate error is created to provoke an error message.

```
try {  
  let y = 1;  
  const x = 2;  
  x = 3;  
} catch (err) {  
  alert(err.message);  
}
```

127.0.0.1:5500 says
undefined

OK

Our code with the error can be wrapped in a try catch clause and the error alerted to the screen.

```

const whereAmI = 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`);
    const dataGeo = await resGeo.json();
    console.log(dataGeo);

    // Country Data
    const res = await fetch(
      `https://restcountries.com/v3.1/name/${dataGeo.country}`
    );
    const data = await res.json();
    console.log(data);
    renderCountry(data[0]);
  } catch (err) {
    console.log(`${err} ❌`);
    renderError(`Something Went Wrong ❌ ${err.message}`);
  }
};

```

Our whereAmI function
can be wrapped in a
try...catch block.

whereAmI();

But this code does not take into account the response of an API promise being a 404 or 403 code because a 404 or 403 is considered as promise fulfilled.

```

const whereAmI = 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
    const res = await fetch(`https://restcountries.com/v3.1/name/${dataGeo.country}`);
    if (!res.ok) throw new Error('Problem getting country data');

    const data = await res.json();
    console.log(data);
    renderCountry(data[0]);
  } catch (err) {
    console.log(`${err} ❌`);
    renderError(`Something Went Wrong ❌ ${err.message}`);
  }
};

```

The first promise does not need a manual throw because it is already built with the reject included.

We can manually code a throw of new error if the response is anything but 200 OK

Never ignore handling errors because it can cause code to stop and log an error to the console. Which might disable other features further down in the code.

whereAmI();

Returning Values from Async functions

```
const whereAmI = 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
    const res = await fetch(`https://restcountries.com/v3.1/name/${dataGeo.country}`);
    if (!res.ok) throw new Error('Problem getting country data');
    const data = await res.json();
    // console.log(data);
    renderCountry(data[0]);
    return `You are in ${dataGeo.city}, ${dataGeo.country}`;
  } catch (err) {
    console.log(`${err} ✖`);
    renderError(`Something Went Wrong ✖ ${err.message}`);
  }
};

console.log('1: Will get location');
const city = whereAmI();
console.log(city);
console.log('3: Finished getting location');
```

Lets say we want to return a string from the whereAmI function.

An async function always returns a promise



```
1: Will get location
Promise {<pending>}
3: Finished getting location
```

When we call an async function it always returns a promise because at this point in the code JavaScript has no way of knowing about the string or what will be returned so it will always return a promise. The fulfilled value of the promise will be the string (if there is no error in the function.)

```
console.log('1: Will get location');
whereAmI().then(city => console.log(city));
console.log('3: Finished getting location');
```

1: Will get location
3: Finished getting location
You are in Sabadell, Spain

As we know the return of whereAmI() is a promise we can use the then method with a callback function to get the string but this does not take into account a failure situation where something goes wrong and the return string line of code is not reached, but immediately jumps to the catch block.

// Country Data

```
const res = await fetch(
  `https://restcountries.com/v3.1/name/${dataGeoxxx.country}`
);
if (!res.ok) throw new Error('Problem getting country data');
const data = await res.json();
// console.log(data);
renderCountry(data[0]);

return `You are in ${dataGeo.city}, ${dataGeo.country}`;
} catch (err) {
  console.log(`${err} ❌`);
  renderError(`Something Went Wrong ❌ ${err.message}`);
}
};
```

Here I have

← provoked an error.

The code jumps straight to the catch clause and returns undefined.

1: Will get location
3: Finished getting location
ReferenceError: dataGeoxxx is not defined ❌
undefined

```

console.log('1: Will get location');
whereAmI()
  .then(city => console.log(`2: ${city}`))
  .catch(err => console.error(`2: ${err.message}`))
  .finally(() => console.log('3: Finished getting location'));

```

```

1: Will get location
3: Finished getting location
ReferenceError: dataGeoxxx is not defined
2: undefined

```

```

console.log('3: Finished getting location');

```

The undefined comes from this line of code which means the console log was executed. If we add a catch method into this promise function we still get undefined.

This means that the promise was fulfilled even though there was an error in the async function.

```

console.log('1: Will get location');
whereAmI()
  .then(city => console.log(`2: ${city}`))
  .catch(err => console.error(`2: ${err.message}`))
  .finally(() => console.log('3: Finished getting location'));

```

```

1: Will get location
ReferenceError: dataGeoxxx is not defined
2: dataGeoxxx is not defined
3: Finished getting location

```

Additionally, The third item is loading before the async function has completed. We can fix this so the items load in sequence of 1, 2 and 3 by adding the 3 into a finally callback function. If we remove the deliberately introduced error or with the error they load in sequence.

```

1: Will get location
2: You are in Sabadell, Spain
3: Finished getting location

```


To solve this we have to re-throw the error in the whereAml function.

```
const whereAmI = 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
    const res = await fetch(`https://restcountries.com/v3.1/name/${dataGeoxxx.country}`);
    if (!res.ok) throw new Error('Problem getting country data');
    const data = await res.json();
    // console.log(data);
    renderCountry(data[0]);

    return `You are in ${dataGeo.city}, ${dataGeo.country}`;
  } catch (err) {
    console.log(`${err} ✖`);
    renderError(`Something Went Wrong ✖ ${err.message}`);

    // reject promise returned from async function
    throw err;
  }
};
```

re-throw the error in the whereAml function, by rejecting the promise.

```
1: Will get location
ReferenceError: dataGeoxxx is not defined ✖
✖ ▶ 2: dataGeoxxx is not defined ✖
3: Finished getting location
```

However using then() and catch() with async/await is mixing old and new methods.

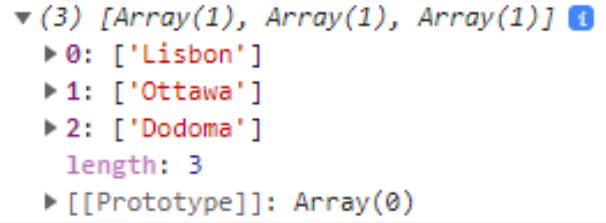
```
(async function () {  
  try {  
    const city = await whereAmI();  
    console.log(`2: ${city}`);  
  } catch (err) {  
    console.error(`2: ${err.message} 💣`);  
  }  
  console.log('3: Finished getting location');  
})();
```

An Immediately Invoked Function Expression (IIFE) can be used with a try catch block in it. We have an async function calling an async function.

```
1: Will get location  
ReferenceError: dataGeoxxx is not defined 💣  
❌ ▶ 2: dataGeoxxx is not defined 💣  
3: Finished getting location
```

Running Promises in parallel

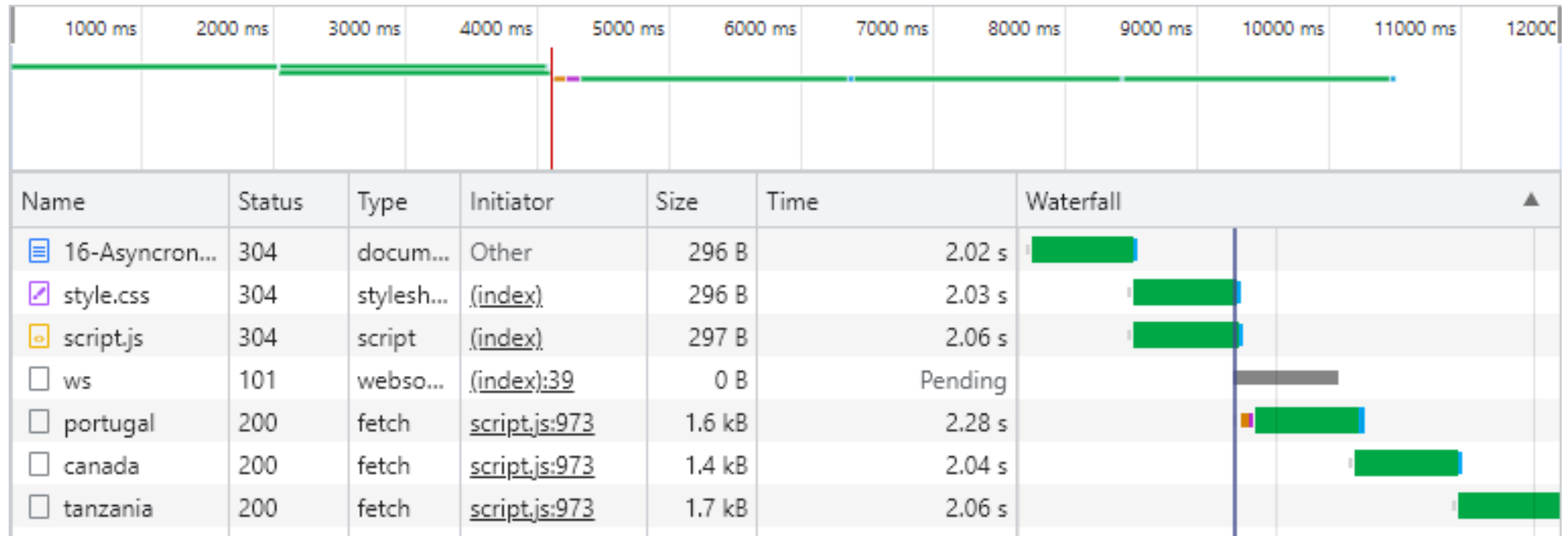
```
const getJSON = function (url, errorMsg = 'Something went wrong') {  
  return fetch(url).then(response => {  
    if (!response.ok) throw new Error(`${errorMsg} (${response.status})`);  
  
    return response.json();  
  });  
};  
  
const get3Countries = async function (c1, c2, c3) {  
  try {  
    const [data1] = await getJSON(`https://restcountries.com/v3.1/name/${c1}`);  
    const [data2] = await getJSON(`https://restcountries.com/v3.1/name/${c2}`);  
    const [data3] = await getJSON(`https://restcountries.com/v3.1/name/${c3}`);  
  
    console.log([data1.capital, data2.capital, data3.capital]);  
  } catch (err) {  
    console.log(err);  
  }  
};  
  
get3Countries('portugal', 'canada', 'tanzania');
```



```
▼ (3) [Array(1), Array(1), Array(1)] ⓘ  
  ▶ 0: ['Lisbon']  
  ▶ 1: ['Ottawa']  
  ▶ 2: ['Dodoma']  
    length: 3  
  ▶ [[Prototype]]: Array(0)
```

We are running three await functions within the try block to output an array of capitals from the results of the countries API.

But this does not make any sense. why should the second AJAX call have to wait for the first AJAX call?



If we look in the network tab we can see that the first to load is the HTML of the page followed by the style.css then the script.js. After that Portugal, Canada, and Tanzania from the countries API one after the other in sequence. The total loading time for a slow 3g connection is 11000ms of 1.1 seconds total.

We can save time by loading these items in parallel rather than one after the other in sequence.

```

const get3Countries = async function (c1, c2, c3) {
  try {
    // const [data1] = await getJSON(`https://restcountries.com/v3.1/name/${c1}`);
    // const [data2] = await getJSON(`https://restcountries.com/v3.1/name/${c2}`);
    // const [data3] = await getJSON(`https://restcountries.com/v3.1/name/${c3}`);

    const data = await Promise.all([
      getJSON(`https://restcountries.com/v3.1/name/${c1}`),
      getJSON(`https://restcountries.com/v3.1/name/${c2}`),
      getJSON(`https://restcountries.com/v3.1/name/${c3}`),
    ]);

    // console.log([data1.capital, data2.capital, data3.capital]);
    console.log(data);
  } catch (err) {
    console.log(err);
  }
};

get3Countries('portugal', 'canada', 'tanzania');

```

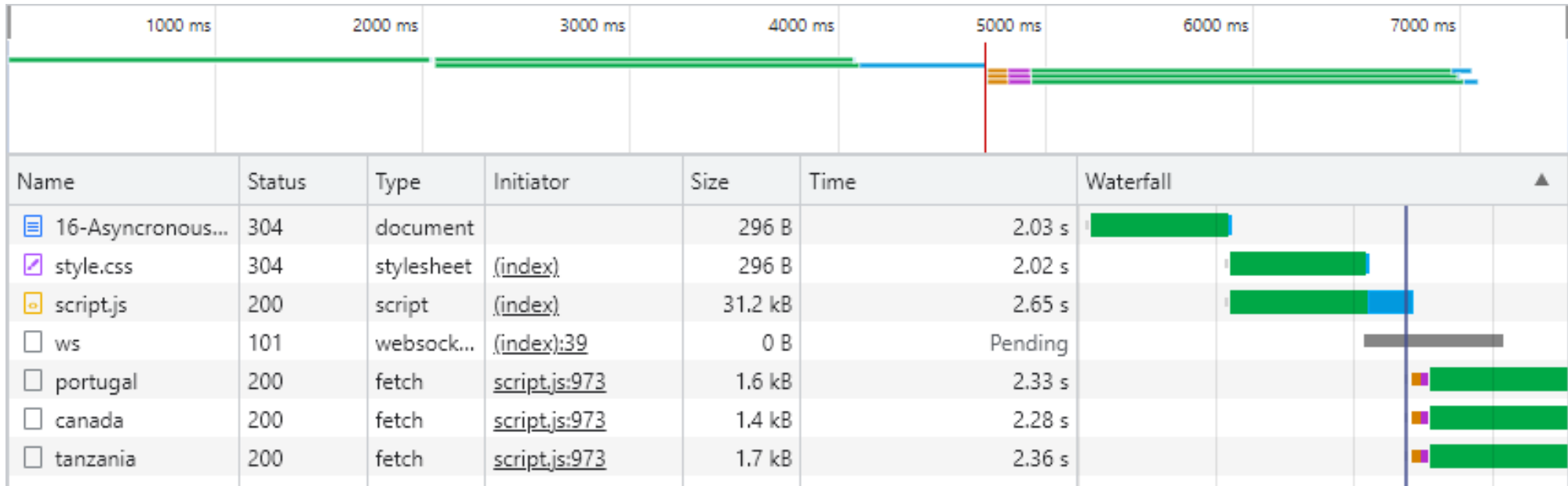
Promise.all receives an array and returns an array.

```

▼ (3) [Array(1), Array(1), Array(1)] ⓘ
  ► 0: [{...}]
  ► 1: [{...}]
  ► 2: [{...}]
    length: 3
  ► [[Prototype]]: Array(0)

```

Now in the network tab we see the three async functions loading in parallel which reduces the loading time from 1.1 seconds to 0.7 seconds.



```
console.log(data.map(d => d[0].capital));  
} catch (err) {  
  console.log(err);  
}
```

To get a new array of capital cities we can just loop over the data and use map to make the new array.

However, Promise.all will reject if one of the async functions within it fail. that is to say it will short circuit and return an error.

Promise.all is called a combinator function because it combines multiple functions into one.

Other Promise Combinators: Race, allSettled and Any.

```
(async function () {  
  const res = await Promise.race([  
    getJSON(`https://restcountries.com/v3.1/name/italy`),  
    getJSON(`https://restcountries.com/v3.1/name/egypt`),  
    getJSON(`https://restcountries.com/v3.1/name/mexico`),  
  ]);  
  console.log(res[0]);  
})();  
  
▶ {name: {...}, tld: Array(1), cca2: 'IT', ccn3: '380', cca3: 'ITA', ...}  
▶ {name: {...}, tld: Array(2), cca2: 'EG', ccn3: '818', cca3: 'EGY', ...}
```

These three promises will race against each other and the winner will be the one that returns first. If we run this function multiple times then it will give a different result each time depending on who the winner is. Promise.race returns if the fastest is a fulfilled or rejected.

```
(async function () {  
  const res = await Promise.race([  
    getJSON(`https://restcountries.com/v3.1/name/xx`),  
    getJSON(`https://restcountries.com/v3.1/name/egypt`),  
    getJSON(`https://restcountries.com/v3.1/name/mexico`),  
  ]);  
  console.log(res[0]);  
})();
```

Xx country does not exist but returns first, thus winning the race even though it returns a rejected promise.

```
✖ ▶ GET https://restcountries.com/v3.1/name/xx 404 (Not Found)  
✖ ▶ Uncaught (in promise) Error: Something went wrong (404)  
   at script.js:1009:29  
   at async script.js:1017:15
```

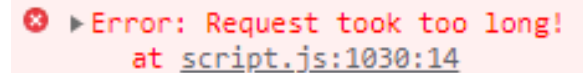
```
const timeout = function (sec) {  
  return new Promise(function (_, reject) {  
    setTimeout(function () {  
      reject(new Error('Request took too long!'));  
    }, sec * 1000);  
  });  
};
```

Here we build a new promise that returns a reject if the promise does not return within the timeout function time.

```
Promise.race([  
  getJSON(`https://restcountries.com/v3.1/name/tanzania`),  
  timeout(5),  
)  
  .then(res => console.log(res[0]))  
  .catch(err => console.error(err));
```

This is useful for loading resources that are only useful for a period of time.

Here we have a Promise.race where the get Tanzania JSON data is actually racing against a timeout set to 5 seconds. If the timeout value is set to 1 second the timeout function wins the Promise.race.

A screenshot of a console error message. It features a red 'x' icon in a circle, followed by the text 'Error: Request took too long!' and 'at script.js:1030:14'.


```

Promise.allSettled([
  Promise.resolve('Success'),
  Promise.reject('ERROR'),
  Promise.resolve('Another success'),
]).then(res => console.log(res));

```

```

▼ (3) [{...}, {...}, {...}] ⓘ
  ► 0: {status: 'fulfilled', value: 'Success'}
  ► 1: {status: 'rejected', reason: 'ERROR'}
  ► 2: {status: 'fulfilled', value: 'Another success'}
    length: 3
  ► [[Prototype]]: Array(0)

```

Promise.allSettled will return all of the promises even if one or more of them gets rejected. (Promise.all will return a rejected if only one of the promises fails.)

```

(async function () {
  const res = await Promise.allSettled([
    getJSON(`https://restcountries.com/v3.1/name/xx`),
    getJSON(`https://restcountries.com/v3.1/name/egypt`),
    getJSON(`https://restcountries.com/v3.1/name/mexico`),
  ]).then(res => console.log(res));
})();

```

✖ **GET** <https://restcountries.com/v3.1/name/xx> `script.js:1008` 
404 (Not Found)

`script.js:1056`

```

▼ (3) [{...}, {...}, {...}] ⓘ
  ► 0: {status: 'rejected', reason: Error: Something went wrong}
  ► 1: {status: 'fulfilled', value: Array(1)}
  ► 2: {status: 'fulfilled', value: Array(1)}
    length: 3
  ► [[Prototype]]: Array(0)

```

```
// Promise.any [ES2021]
Promise.any([
  Promise.resolve('Success'),
  Promise.reject('ERROR'),
  Promise.resolve('Another success'),
])
  .then(res => console.log(res))
  .catch(err => console.error(err));
```

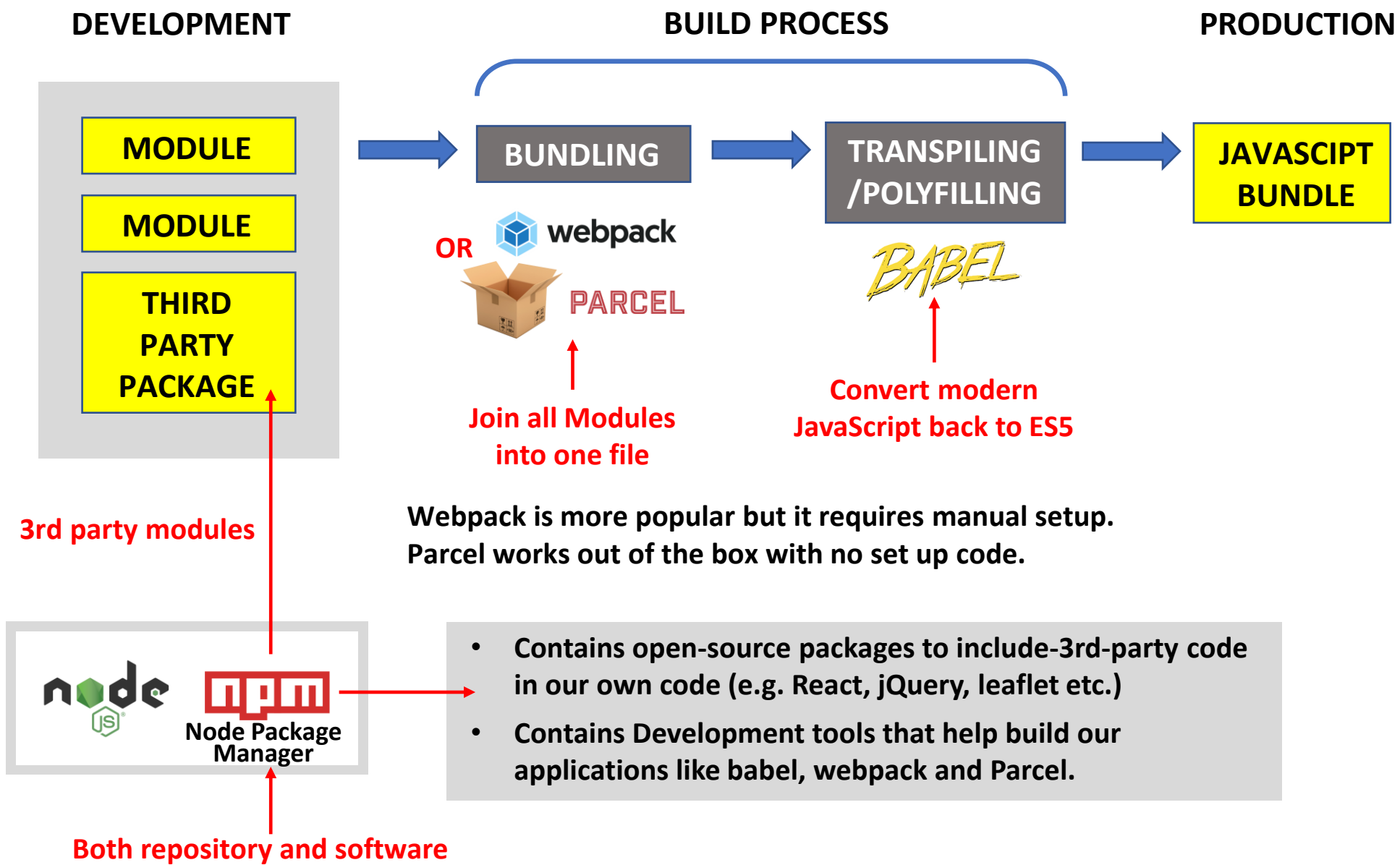
Success

Promise.any will return the fastest loading promise that returns a fulfilled. Unlike **Promise.race** that will return the fastest loading promise regardless of if it is fulfilled or rejected.

Of course if all promises are rejected then it will return rejected.

Modern JavaScript Development: Modules, Tooling, and Functional

An Overview of Modern JavaScript Development



An Overview Modules

MODULE

1. Reusable piece of code that encapsulates implementation details.
2. Usually a **standalone** file but it doesn't have to be.

- **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.
- **Organise code:** Modules naturally lead to a more organised code base.
- **Reuse Code:** Modules allow us to easily reuse the same code, even across multiple projects.

Import (Dependency)

```
MODULE
import { rand } from './math.js';
const diceP1 = rand(1, 6, 2);
const diceP2 = rand(1, 6, 2);
const scores = { diceP1, diceP2 };
export { scores };
```

Module code

Export (public API)

ES6 Modules

ES6 MODULE

Modules stored in files,
exactly one module per file

Import and
export
Syntaxes

```
import { rand } from './math.js';  
const diceP1 = rand(1, 6, 2);  
const diceP2 = rand(1, 6, 2);  
const scores = { diceP1, diceP2 };  
export { scores };
```

ES6 MODULE

SCRIPT

Top-level Variables

Scoped to module

Global

Default mode

Strict Mode

"sloppy" mode

Top-level this

undefined

window

Imports & Exports

YES 

NO 

HTML linking

<script type="module">

<script>

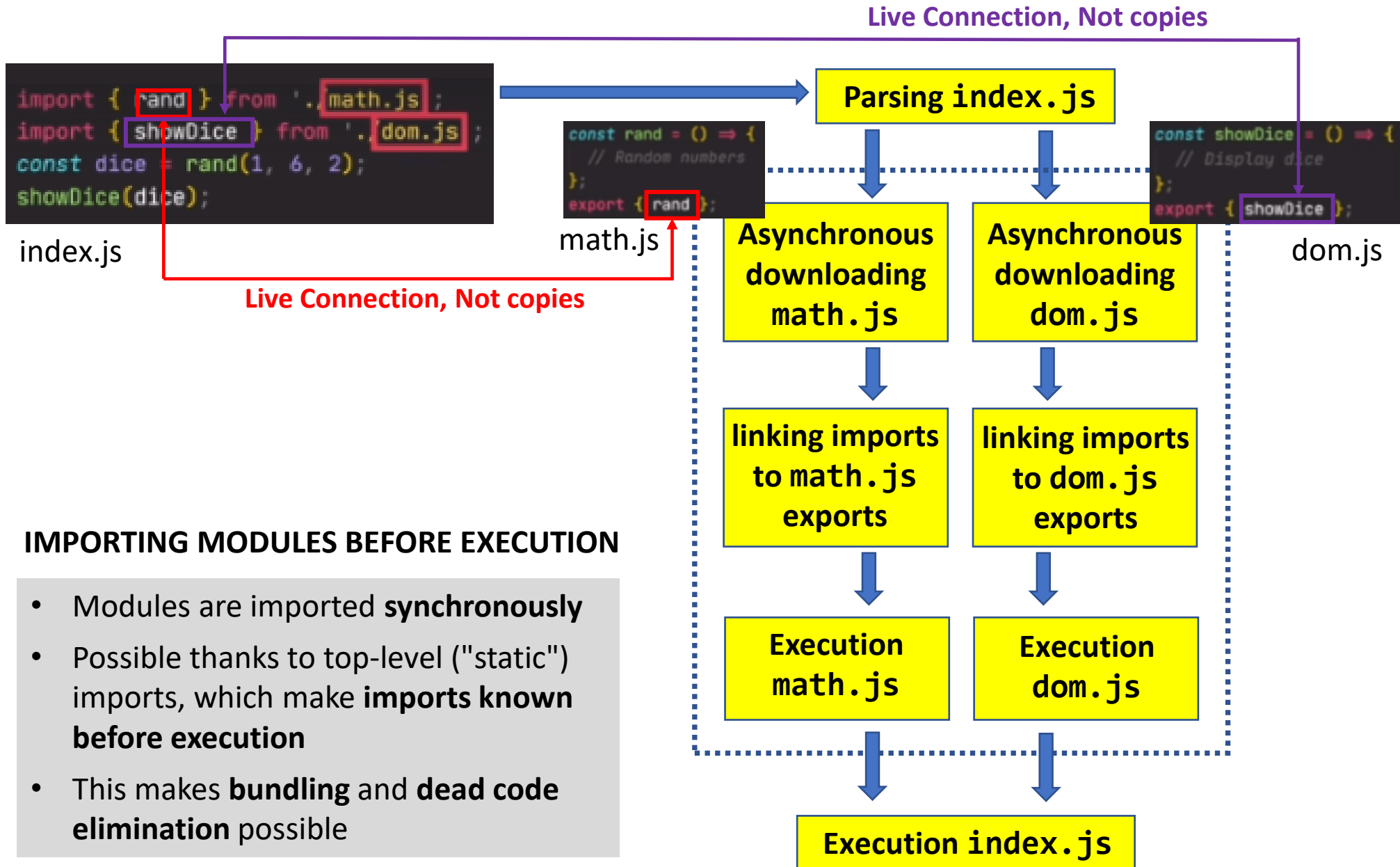
File Downloading

Asynchronous

Synchronous

Need to happen
at top-level:
Imports are
hoisted

How ES6 Modules are Imported



Exporting & Importing ES6 Modules

```
// script.js file: Importing module
import './shoppingCart.js';
console.log('importing module');
```

✖ Uncaught SyntaxError: Cannot use import statement outside a module (at script.js:2:1)

```
<!DOCTYPE html>
<html lang="en">
  <head>
```

```
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <script type="module" defer src="script.js"></script>
```

To connect a module to a html file we have to specify the type.

```
// ShoppingCart.js file: Exporting Module
console.log('Exporting Module');
```

Exporting Module	<u>shoppingCart.js:2</u>
importing module	<u>script.js:3</u>

Note how the Exporting module appears in the console first proving that the ShoppingCart.js file is executed first before the script.js file. The script.js file is parsed then all the code in the files that it imports is executed first.

`console.log('importing module');` It would not matter if the console log was before the import because all the imports are hoisted to the top.

Notice that at the top of the JavaScript files strict mode is not set. This is because all modules use strict mode by default.


```
// shoppingCart.js: Exporting Module
console.log('Exporting Module');
const shippingCost = 10;
const cart = [];
```

```
// script.js: Importing module
import './shoppingCart.js';
console.log('importing module');
console.log(shippingCost);
```

```
// shoppingCart.js: Exporting Module
const shippingCost = 10;
const cart = [];
```

```
export const addToCart = function (product, quantity) {
  cart.push({ product, quantity });
  console.log(`${quantity} ${product} added to cart`);
};
```

```
// script.js: Importing module
import { addToCart } from './shoppingCart.js';

addToCart('bread', 5);
```

```
✖ ▶ Uncaught ReferenceError:
  shippingCost is not defined
    at script.js:5:13
```

All the variables in a module have local scope to that module or file and are not pushed into the global of the importing module.

Now shoppingCart.js contains a simple function that pushes a product and quantity into an array and logs the action.

If we want to be able to use this function in script.js then we preface the function with 'export'.

Now in script.js we can import that function from shoppingCart.js but we need to wrap it in curly braces as a named export.

```
5 bread added to cart    shoppingCart.js:8
```

addToCart can be called from script.js

```
// shoppingCart.js: Exporting Module
const shippingCost = 10;
const cart = [];
```

✖ Uncaught SyntaxError: shoppingCart.js:7
Unexpected token 'export' (at shoppingCart.js:7:3)

Exports must happen at top level code so this would not work.

```
if (true) {
  export const addToCart = function (product, quantity) {
    cart.push({ product, quantity });
    console.log(`${quantity} ${product} added to cart`);
  };
}
```

```
// shoppingCart.js: Exporting Module
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;
const totalQuantity = 23;
```

```
export { totalPrice, totalQuantity };
```

We can also export variables
like they were an object...

```
// script.js: Importing module
```

```
import { addToCart, totalPrice, totalQuantity } from './shoppingCart.js';
```

...And Import them by name.

```
addToCart('bread', 5);
```

```
console.log(totalPrice, totalQuantity);
```

```
5 bread added to cart
```

```
shoppingCart.js:8
```

```
237 23
```

```
script.js:5
```

```
// script.js: Importing module
```

```
import { addToCart, totalPrice as price, totalQuantity, } from './shoppingCart.js';
```

```
addToCart('bread', 5);
```

```
console.log(price, totalQuantity);
```

Imported items can be renamed.

```
// shoppingCart.js: Exporting Module
```

```
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;
```

```
const totalQuantity = 23;
```

```
export { totalPrice, totalQuantity as qty }; Exported items can also be renamed.
```

```
// script.js: Importing module
import { addToCart, totalPrice as price, totalQuantity, } from './shoppingCart.js';
```

```
addToCart('bread', 5);
console.log(price, totalQuantity);
```

✖ Uncaught SyntaxError: The requested `script.js:5` module `./shoppingCart.js` does not provide an export named `'totalQuantity'` (at `script.js:5:3`)

```
// script.js: Importing module
import { addToCart, totalPrice as price, qty } from './shoppingCart.js';
```

```
addToCart('bread', 5);
console.log(price, qty);
```

5 bread added to cart	<code>shoppingCart.js:8</code>
237 23	<code>script.js:5</code>

The name exported must match the name imported

```
// script.js: Importing module
```

```
// import { addToCart, totalPrice as price, qty } from './shoppingCart.js';
import * as ShoppingCart from './shoppingCart.js';
```

```
// addToCart('bread', 5);
ShoppingCart.addToCart('bread', 5);
```

We can import all as an Object from another JS file.

And call methods within that object

```
// console.log(price, qty);
console.log(ShoppingCart.totalPrice, ShoppingCart.qty);
```

Or call properties of the object

// Export Named

```
export const addToCart = function (product, quantity) {  
  cart.push({ product, quantity });  
  console.log(`${quantity} ${product} added to cart`);  
};
```

```
const totalPrice = 237;  
const totalQuantity = 23;
```

```
export { totalPrice, totalQuantity as qty };
```

There are named Exports
(Examples that we have done
so far) or default exports
when there is only one thing
exported from a module.

Here we are
exporting by name.

// Export default

```
export default function (product, quantity) {  
  cart.push({ product, quantity });  
  console.log(`${quantity} ${product} added to cart`);  
};
```

Here we are
exporting the output
of the function. It has
no name.

Import add from will get all
default exports from the
specified file.

```
// script.js: Importing module  
// import { addToCart, totalPrice as price, qty } from  
// './shoppingCart.js';  
import * as ShoppingCart from './shoppingCart.js';  
// addToCart('bread', 5);  
ShoppingCart.addToCart('bread', 5);  
// console.log(price, qty);  
console.log(ShoppingCart.totalPrice, ShoppingCart.qty);
```

```
5 bread added to cart      shoppingCart.js:9  
237 23                    script.js:7  
2 pizza added to cart     shoppingCart.js:20
```

```
// Import Default  
import add from './shoppingCart.js';  
add('pizza', 2);
```

```
// Mixing named and default imports
import add, { addToCart, totalPrice as price, qty } from './shoppingCart.js';
console.log(price);
```

We can mix the import of named and default but this is not normally done.

```
// shoppingCart.js: Exporting Module
```

```
const shippingCost = 10;
```

```
export const cart = [];
```

← We export the variable of cart which is an empty array

```
// Export default
```

```
export default function (product, quantity) {
  cart.push({ product, quantity });
  console.log(`${quantity} ${product} added to cart`);
}
```

← We default export the add to cart function.

```
// script.js: Importing module
```

```
// Mixing named and default imports
```

```
import add, { cart } from './shoppingCart.js';
```

←

```
add('pizza', 2);
```

```
add('bread', 5);
```

```
add('tomatoes', 4);
```

```
console.log(cart);
```

←

Because this is a live connection
the 'empty' cart is now full.

→

We can now import the cart which
is a blank array. And also import the
default.

```
2 pizza added to cart      shoppingCart.js:20
5 bread added to cart      shoppingCart.js:20
4 tomatoes added to cart   shoppingCart.js:20
```

script.js:18

```
▼ (3) [{...}, {...}, {...}] ⓘ
  ▶ 0: {product: 'pizza', quantity: 2}
  ▶ 1: {product: 'bread', quantity: 5}
  ▶ 2: {product: 'tomatoes', quantity: 4}
    length: 3
  ▶ [[Prototype]]: Array(0)
```

IMPORTS ARE NOT COPIES BUT A LIVE CONNECTION!!!!!!

Top Level Await (ES2022)

Top level await only works in a module so this line of html must have the type="module" parameter set

```
<script type="module" defer src="script.js"></script>
```

```
// script.js: Importing module
const res = await fetch('https://jsonplaceholder.typicode.com/posts');
const data = await res.json();
console.log(data);
```

Fetch dummy of data from <https://jsonplaceholder.typicode.com> API for testing using the await keyword.

Note that in modules we do not write the `async` keyword nor write it as a function `async function x (){};` like we would have to do if this was not a module. **However, This now blocks t**

However, This now blocks the execution of the entire module.

```
console.log('start fetching');
const res = await fetch('https://jsonplaceholder.typicode.com/posts');
const data = await res.json();
console.log(data);
console.log('Something');
```

Something is not logged until the API has finished.

[illegible]

```
const getLastPost = async function () {  
  const res = await  
fetch('https://jsonplaceholder.typicode.com/posts');  
  const data = await res.json();  
  console.log(data);  
  
  return { title: data.at(-1).title, text: data.at(-1).body  
};
```

```
const lastPost = getLastPost();  
console.log(lastPost);  
  
lastPost.then(last => console.log(last));
```

```
▶ {title: 'at nam consequatur ea labore ea harum', text: 'cupiditate quo est a modi nesciunt  
soluta\nipsa vol...nam et distinctio eum\naccusamus ratione error aut'}
```

script.js:37

```
const lastPost2 = await getLastPost();  
console.log(lastPost2);
```

```
▶ {title: 'at nam consequatur ea labore ea harum', text: 'cupiditate quo est a modi nesciunt  
soluta\nipsa vol...nam et distinctio eum\naccusamus ratione error aut'}
```

script.js:40

This will return a promise, not the actual data because at the time of the console.log the data is still in transit. Async functions always return a promise.

► *Promise {<pending>}*

The workaround is to use the then() method but this is not very clean.

However if we use top-level await we do not need the then() method.

If one module imports a module which has a top level await, then the importing module will wait until the blocking code is finished.

```
// shoppingCart.js: Exporting Module
```

```
console.log('Start Fetching Users');  
await fetch('https://jsonplaceholder.typicode.com/users');  
console.log('Finish fetching users');
```

Blocking code in the
exporting JS Module.

```
// Export Named
```

```
export const addToCart = function (product, quantity) {  
  cart.push({ product, quantity });  
  console.log(`${quantity} ${product} added to cart`);  
};
```

```
// script.js: Importing module
```

```
import * as ShoppingCart from './shoppingCart.js';  
ShoppingCart.addToCart('bread', 5);
```

Start Fetching Users	shoppingCart.js:3
Finish fetching users	shoppingCart.js:5
5 bread added to cart	shoppingCart.js:13

We see that the blocking code does block in the importing module. The code in Script.js has to wait for the code from shoppingCart.js to complete. This is something to be careful of when using top level await.

The Module Pattern (Pre ES6 modules)

```
const ShoppingCart2 = (function () {  
  const cart = [];  
  const shippingCost = 10;  
  const totalPrice = 237;  
  const totalQuantity = 23;
```

The implementation of the module pattern is to wrap something in a function and only return what we want to be public and it was how modular functionality was implemented in JS pre ES6.

```
  const addToCart = function (product, quantity) {  
    cart.push({ product, quantity });  
    console.log(`${quantity} ${product} added to cart (shipping cost is ${shippingCost})`);  
  };  
};
```

```
const orderStock = function (product, quantity) {  
  console.log(`${quantity} ${product} ordered from supplier`);  
};
```

Note how this IIFE was invoked once and immediately so why does the cart array have values from calling it outside of the function?

The answer is closures: Allow a function to have access to all the variables at it's birthplace. The add to cart function was born in the shoppingCart function so it has access to all the variables within shoppingCart.

As we are not returning shipping cost from ShoppingCart2 it cannot be accessed from outside the function or module pattern.

```
  return {  
    addToCart,  
    cart,  
    totalPrice,  
    totalQuantity,  
  };  
})();
```

```
ShoppingCart2.addToCart('apple', 4);  
ShoppingCart2.addToCart('pizza', 2);  
console.log(ShoppingCart2);  
console.log(ShoppingCart2.shippingCost);
```

4 apple added to cart (shipping cost is 10)	script.js:52
2 pizza added to cart (shipping cost is 10)	script.js:52
▶ {cart: Array(2), totalPrice: 237, totalQuantity: 23, addToCart: f}	script.js:71
undefined	script.js:72

Common JS Modules

Besides native ES modules and the module pattern there are other module systems that have been used by JavaScript in the past but they are not native to JavaScript.

Two examples are AMD modules and Common Js Modules. CommonJs modules are important because they have been used in NodeJs for almost all of its existence. Only recently have ES6 modules been implemented in NodeJS.

As a consequence, almost all of the modules in the NPM repository still use the commonJS module system because NPM was originally only intended for NodeJs. Only later did NPM become the standard repository for the whole JavaScript world so now we are basically stuck with commonJS.

```
// Export
export.addToCart = function (product, quantity) {
  cart.push({ product, quantity });
  console.log(
    `${quantity} ${product} added to cart (shipping cost is ${shippingCost})`
  );
};

// import
const {addToCart} = require('./shoppingCart.js');
```

One example of commonJs is the export keyword which would not work in the browser but will work in NodeJs.

Require is the commonJs equivalent of ES6 import.

Hopefully commonJs will eventually be replaced by ES6 JS in NPM but meanwhile we still need to be aware of commonJS.

A Brief Introduction to the Command Line

All the NPM tools are only available via the command line so we need to understand a little about it.

From VS code we can open a terminal from the terminal menu at the top. The first thing to know about terminal is that we are always in a folder, in this case the folder where the course project files are located.

```
ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Comple
e-Javascript-Course/course-exercices (main)
$
```

```
ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Comple
e-Javascript-Course/course-exercices (main)
$ ls
'~$complete-javascript-course.pptx'
01-fundamentals-part-1/
02-fundamentals-part-2/
03-developer-skills-vs-studio-setup/
04-html-css-crash.course/
05-guess-my-number/
06-Modal/
07-Pig-Game/
08-javascript-behind-the-scenes/
09-Data-Structures-Operators/
10-Closer-Look-at-Functions/
11-Working-with-Arrays/
12-Numbers-Dates-Timers-Bankist/
13-Advanced-DOM-and-Events/
14-Object-Orientated-Programming/
15-Mapty-App-OOP-Geolocation-External-Libraries/
16-Asynchronous-JavaScript-Promises-Async-Await-AJAX/
17-Modern-JavaScript-Development-Modules-Tooling-Functional/

ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Comple
e-Javascript-Course/course-exercices (main)
$
```

ls will show the contents of the folder. Or in windows machines dir.

```
ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course/course-exercices (main)
$ cd ..

ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course
$ ls
'~$complete-javascript-course.pptx'  course-exercices/
complete-javascript-course.pptx      course-material/
```

```
ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course
$ cd ../../

ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript
$ ls
05-javascript-lessons/  06-jquery/
```

cd can be used to change directory, in this case **cd ..** Takes us up one level in the directory structure

cd ../../ Will take us up two levels in the directory structure.

clear will remove everything from the terminal window.

- mkdir** – Make directory (new directory)
- touch index.html** – new file called index.html
- touch script.js** – new file called script.js
- cd xx [tab]** – tab will autocomplete
- rm [filename]** – delete file
- mv [filename] [new location]** – move file to

For more information about the command line see course notes from <https://www.udemy.com/course/git-and-github-complete-guide/>

Introduction to NPM

Lets now finally use Node Package Manager (NPM). It is both software on our computer and a repository.

Why do we need a way of managing packages in our project? In the old days we used an external library like we did in the Mapty project and we simply included the leaflet file by adding an external script.

This could create problems in a large project or with lots of developers. Also if this script points to a locally stored version of the external library then we would need to update this file when a new version came out.

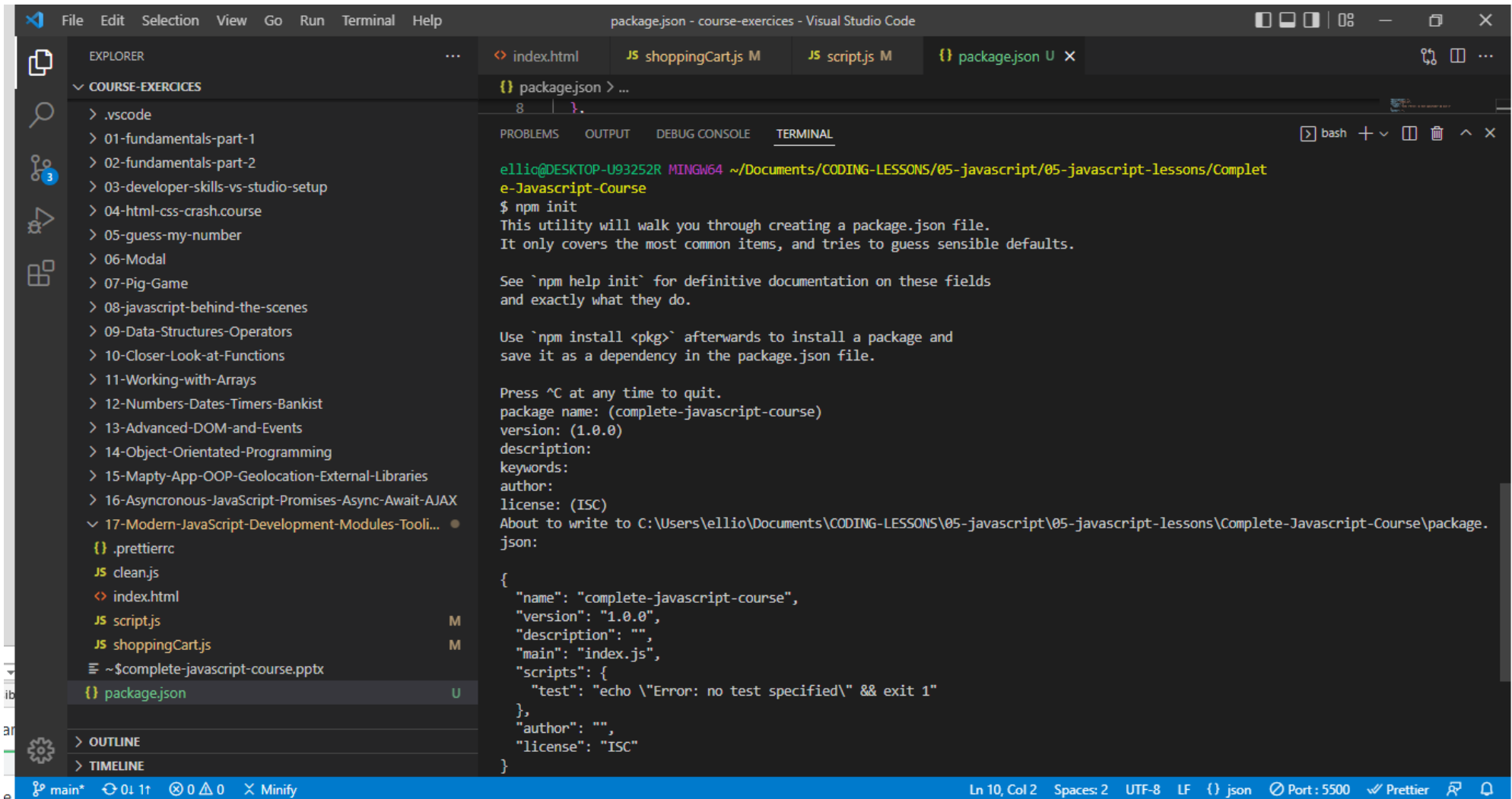
This can become tedious if we have to update multiple files and dependencies.

NPM solves this by having one location for all JavaScript files.

```
ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Comple
e-Javascript-Course
$ npm -v
8.3.1
```

From the terminal an
`npm -v` command will
tell us what version we
have installed.

We can use `npm init` from the command line to create a `package.json` file.



The screenshot shows the Visual Studio Code interface with the Explorer sidebar on the left displaying a project structure for 'COURSE-EXERCISES'. The file explorer lists various folders and files, including `.vscode`, `01-fundamentals-part-1`, `02-fundamentals-part-2`, `03-developer-skills-vs-studio-setup`, `04-html-css-crash.course`, `05-guess-my-number`, `06-Modal`, `07-Pig-Game`, `08-javascript-behind-the-scenes`, `09-Data-Structures-Operators`, `10-Closer-Look-at-Functions`, `11-Working-with-Arrays`, `12-Numbers-Dates-Timers-Bankist`, `13-Advanced-DOM-and-Events`, `14-Object-Orientated-Programming`, `15-Mapty-App-OOP-Geolocation-External-Libraries`, `16-Asynchronous-JavaScript-Promises-Async-Await-AJAX`, and `17-Modern-JavaScript-Development-Modules-Tools...`. The file `package.json` is highlighted in the Explorer sidebar.

The main editor area shows the `package.json` file being created. The terminal output shows the command `npm init` being executed, which prompts the user to enter package details. The resulting `package.json` file is displayed in the editor:

```
{
  "name": "complete-javascript-course",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

The terminal output also includes the following text:

```
ellie@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Comple
e-Javascript-Course
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

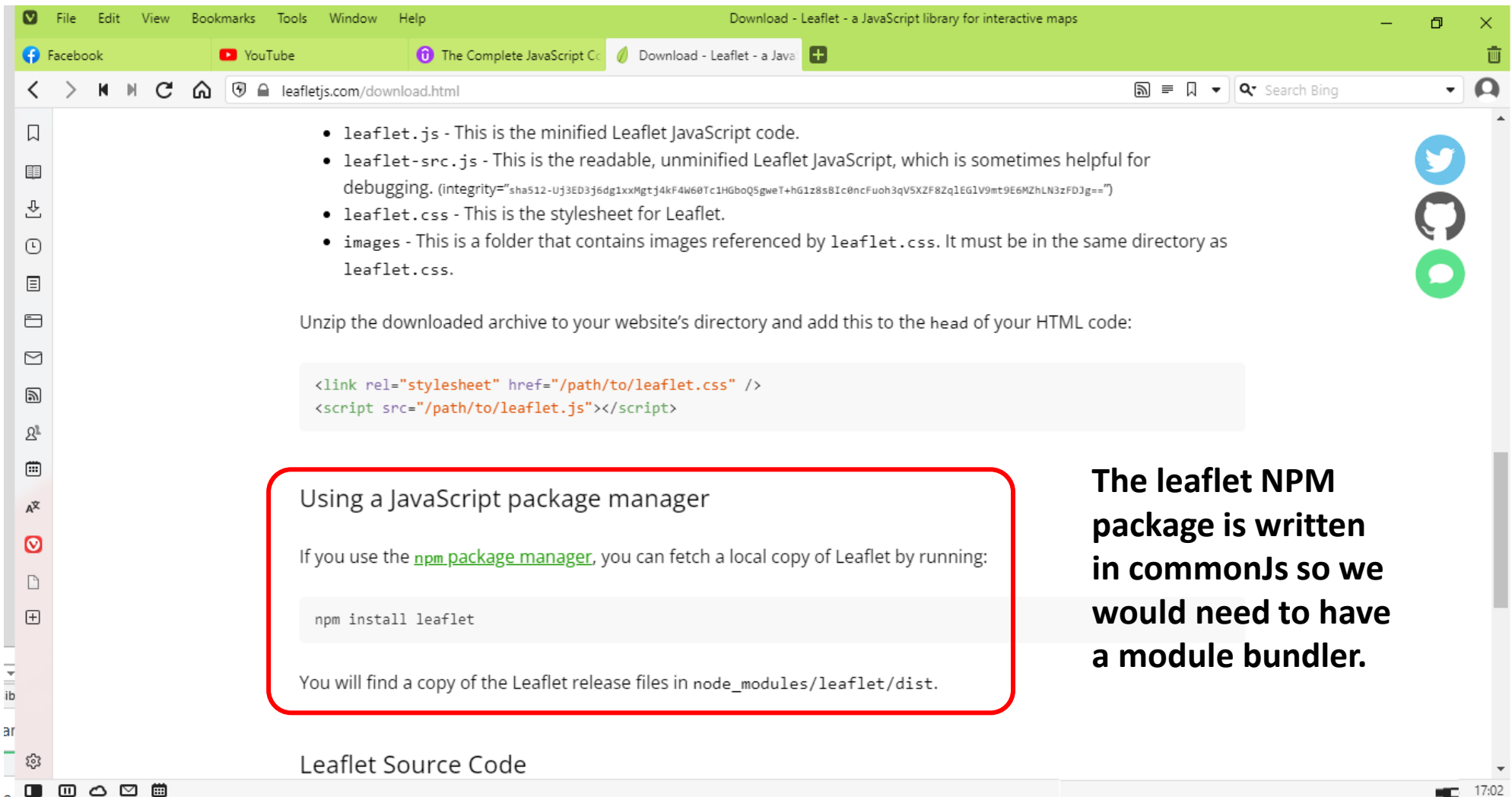
See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (complete-javascript-course)
version: (1.0.0)
description:
keywords:
author:
license: (ISC)
About to write to C:\Users\ellio\Documents\CODING-LESSONS\05-javascript\05-javascript-lessons\Complete-Javascript-Course\package.
json:
```

The status bar at the bottom of the editor shows the current file is `package.json`, the encoding is `UTF-8`, and the line and column numbers are `Ln 10, Col 2`.

We can install leaflet in NPM.



The screenshot shows the Leaflet download page. A red box highlights the section titled "Using a JavaScript package manager". To the right of this box, a text block states that the Leaflet NPM package is written in commonJS and requires a module bundler. The page content includes a list of files to download, instructions to unzip the archive, and HTML code snippets for including Leaflet in a web page.

- `leaflet.js` - This is the minified Leaflet JavaScript code.
- `leaflet-src.js` - This is the readable, unminified Leaflet JavaScript, which is sometimes helpful for debugging. (integrity="sha512-Uj3ED3j6dg1xxMgtj4kF4W60Tc1HGboQ5gweT+hG1z8s8Ic0ncFuoh3qV5XZF8Zq1EG1V9mt9E6MZLN3zFDJg==")
- `leaflet.css` - This is the stylesheet for Leaflet.
- `images` - This is a folder that contains images referenced by `leaflet.css`. It must be in the same directory as `leaflet.css`.

Unzip the downloaded archive to your website's directory and add this to the head of your HTML code:

```
<link rel="stylesheet" href="/path/to/leaflet.css" />
<script src="/path/to/leaflet.js"></script>
```

Using a JavaScript package manager

If you use the [npm package manager](#), you can fetch a local copy of Leaflet by running:

```
npm install leaflet
```

You will find a copy of the Leaflet release files in `node_modules/leaflet/dist`.

The leaflet NPM package is written in commonJs so we would need to have a module bundler.

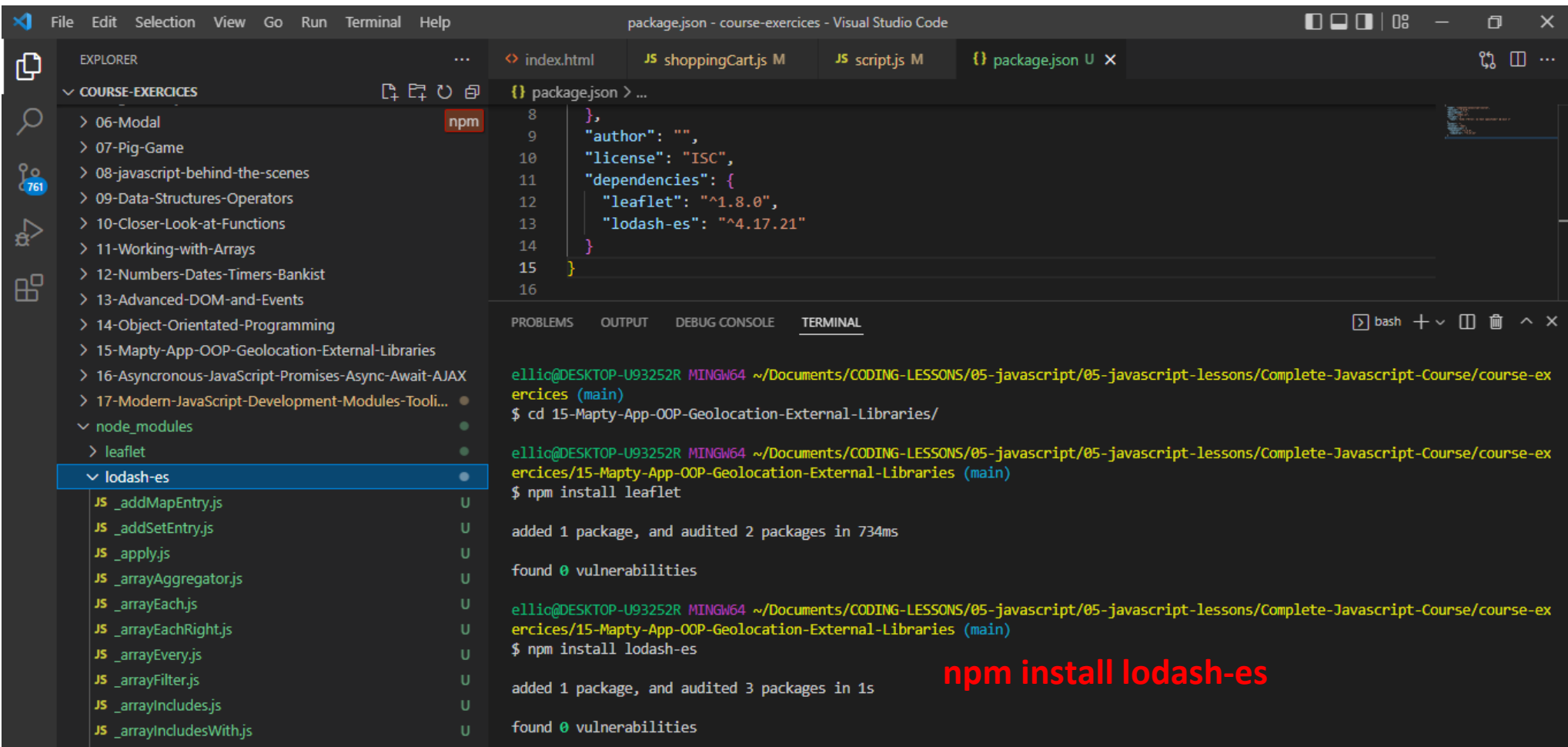
Leaflet Source Code

```
ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course
$ npm install leaflet

added 1 package, and audited 2 packages in 746ms

found 0 vulnerabilities
```


<https://lodash.com> is a collection of useful functions for arrays, dates and more that should be included in JavaScript but are not. The default lodash is built in commonJs but there is a version that works for javascriptES <https://www.npmjs.com/package/lodash-es>



```
File Edit Selection View Go Run Terminal Help package.json - course-exercises - Visual Studio Code

EXPLORER
  COURSE-EXERCISES
    06-Modal
    07-Pig-Game
    08-javascript-behind-the-scenes
    09-Data-Structures-Operators
    10-Closer-Look-at-Functions
    11-Working-with-Arrays
    12-Numbers-Dates-Timers-Bankist
    13-Advanced-DOM-and-Events
    14-Object-Orientated-Programming
    15-Mapty-App-OOP-Geolocation-External-Libraries
    16-Asynchronous-JavaScript-Promises-Async-Await-AJAX
    17-Modern-JavaScript-Development-Modules-Tools...
    node_modules
      leaflet
      lodash-es
        JS _addMapEntry.js
        JS _addSetEntry.js
        JS _apply.js
        JS _arrayAggregator.js
        JS _arrayEach.js
        JS _arrayEachRight.js
        JS _arrayEvery.js
        JS _arrayFilter.js
        JS _arrayIncludes.js
        JS _arrayIncludesWith.js

index.html JS shoppingCart.js M JS script.js M package.json U x

package.json > ...
  8 },
  9 "author": "",
 10 "license": "ISC",
 11 "dependencies": {
 12   "leaflet": "^1.8.0",
 13   "lodash-es": "^4.17.21"
 14 }
 15 }
 16

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
  bash + v [ ] [ ] ^ x

ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course/course-exercises (main)
$ cd 15-Mapty-App-OOP-Geolocation-External-Libraries/

ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course/course-exercises/15-Mapty-App-OOP-Geolocation-External-Libraries (main)
$ npm install leaflet

added 1 package, and audited 2 packages in 734ms

found 0 vulnerabilities

ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course/course-exercises/15-Mapty-App-OOP-Geolocation-External-Libraries (main)
$ npm install lodash-es

added 1 package, and audited 3 packages in 1s

found 0 vulnerabilities
```

Will add the lodash-es files to the node_module directory

```
import cloneDeep from '../node_modules/lodash-es/cloneDeep.js';
```

```
const state = {  
  cart: [  
    { product: 'bread', quantity: 5 },  
    { product: 'pizza', quantity: 5 },  
  ],  
  user: { loggedIn: true },  
};  
const stateClone = Object.assign({}, state);  
const stateDeepClone = cloneDeep(state);  
state.user.loggedIn = false;  
console.log(stateClone);  
console.log(stateDeepClone);
```

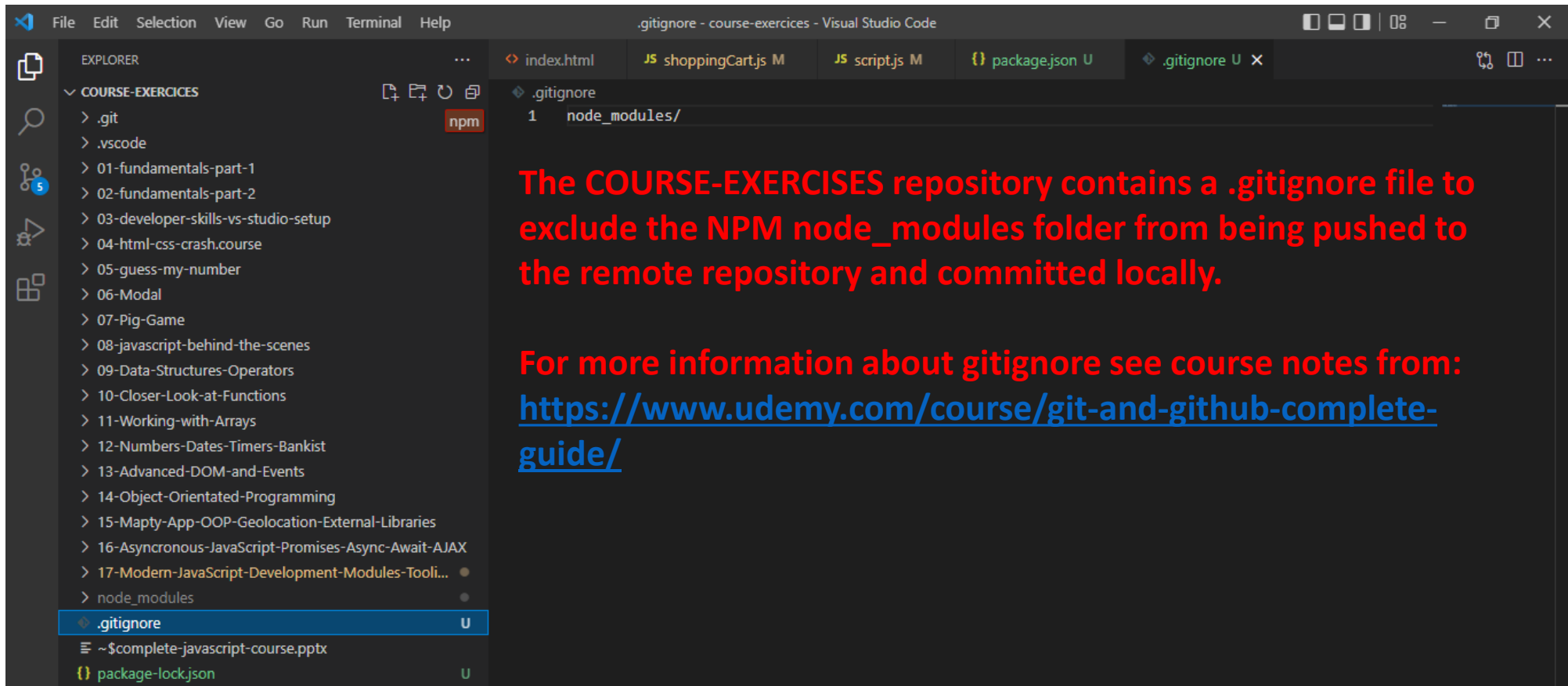
```
▼ {cart: Array(2), user: {...}} ⓘ  
  ▼ cart: Array(2)  
    ► 0: {product: 'bread', quantity: 5}  
    ► 1: {product: 'pizza', quantity: 5}  
    length: 2  
    ► [[Prototype]]: Array(0)  
  ▼ user:  
    loggedIn: false  
    ► [[Prototype]]: Object  
    ► [[Prototype]]: Object
```

Here is an object with property of user.loggedIn set to true which we are then setting the property to false. With a normal JS clone the object would retain its new values but with a deep clone we clone the original.

```
▼ {cart: Array(2), user: {...}} ⓘ  
  ▼ cart: Array(2)  
    ► 0: {product: 'bread', quantity: 5}  
    ► 1: {product: 'pizza', quantity: 5}  
    length: 2  
    ► [[Prototype]]: Array(0)  
  ▼ user:  
    loggedIn: false  
    ► [[Prototype]]: Object  
    ► [[Prototype]]: Object
```

```
▼ {cart: Array(2), user: {...}} ⓘ  
  ▼ cart: Array(2)  
    ► 0: {product: 'bread', quantity: 5}  
    ► 1: {product: 'pizza', quantity: 5}  
    length: 2  
    ► [[Prototype]]: Array(0)  
  ▼ user:  
    loggedIn: true  
    ► [[Prototype]]: Object  
    ► [[Prototype]]: Object
```

Exclude NPM node_modules Folder from Git Repository.



Restore NPM Packages from Cloned Git Repository

```
{  
  "name": "complete-javascript-course",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "leaflet": "^1.8.0",  
    "lodash-es": "^4.17.21"  
  }  
}
```

The package.json file contains a list of all the npm software that we have installed so if we add our project to a git folder and correctly gitignore the node-modules folder all we have to do when we clone that repository is from the terminal 'npm Install' command and it will install all the dependencies listed in the package.json file.

Bundling with parcels and NPM scripts

Lets install NPM parcel <https://parceljs.org/getting-started/webapp/> Using command: `npm install --save-dev parcel` Note: `--save-dev parcel` indicates that it is a tool that we use to build our application but not something we include in our code.

```
{
  "name": "complete-javascript-course",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "leaflet": "^1.8.0",
    "lodash-es": "^4.17.21"
  },
  "devDependencies": {
    "parcel": "^2.6.0"
  }
}
```

Now we have parcel installed in the devDependencies section of the package.json file

```
ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course/course-exercices (main)
$ cd 17-Modern-JavaScript-Development-Modules-Tooling-Functional/

ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course/course-exercices/17-Modern-JavaScript-Development-Modules-Tooling-Functional (main)
$ npx parcel index.html
Server running at http://localhost:1234
✨ Built in 1.18s
```

I can use `cd` to change to the directory where the index file is that contains the html tag identifying the script files.

I use `npx parcel index.html` command and parcel will analyse the JavaScript files to convert all the modules into one file.

Note the output of this command contains a link to <http://localhost:1234> which is like live server but for the parcel compiled version of code and we can open it in our browser to test it.

```
if (module.hot) {
  module.hot.accept();
}
```

But if we change something in one of the JavaScript modules than I would also like that to change to be reflected in the parcel environment. What the `module.hot` lines of code do is permit any changes made in the source js files to be immediately reflected in the browser without a page reload which could become tiresome when developing a new app.

What parcel did was create a new folder called dist which contains the new js and html files for our project.

The screenshot shows the Visual Studio Code interface with the Explorer sidebar on the left and the Terminal at the bottom. The Explorer sidebar shows a project structure for 'COURSE-EXERCICES' with a folder '17-Modern-JavaScript-Development-Modules-Tooling-...' expanded to show a subfolder 'dist'. The 'dist' folder contains three files: 'index.26045220.js', 'index.26045220.js.map', and 'index.html'. The main editor shows the 'package.json' file with the following content:

```
{
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "leaflet": "^1.8.0",
    "lodash-es": "^4.17.21"
  },
  "devDependencies": {

```

The Terminal at the bottom shows the following output:

```
-LESSONS/05-javascript/05-javascript-lessons/Complete-J
avascript-Course/course-exercices (main)
$ cd 17-Modern-JavaScript-Development-Modules-Tooling-F
unctional/

ellio@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSON
S/05-javascript/05-javascript-lessons/Complete-Javascr
ipt-Course/course-exercices/17-Modern-JavaScript-Develo
pment-Modules-Tooling-Functional (main)
$ npx parcel index.html
Server running at http://localhost:1234
🌟 Built in 1.18s
```

The status bar at the bottom shows the file is 'main*', has 0 errors and 0 warnings, and is using 'UTF-8' encoding and 'LF' line endings.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <script defer="" src="/index.26045220.js"></script>
    <title>Modern JavaScript Development: Modules and Tooling</title>
    <style>body {
height: 100vh;
background: linear-gradient(to top left, #28b487, #7dd56f);
align-items: center;
display: flex;
}
```

Note how the script tag of the new HTML file has been modified from
`<script type="module" defer src="script.js"></script>`
To `<script defer="" src="/index.26045220.js"></script>`

It is no longer of type module because the JS modules have been merged into one file.

The new index.26054220.js and index.26045220.js.map files can only be interpreted by parcel and are not the final version that would be uploaded to the server because a browser would not understand this code.


```

{
  "name": "complete-javascript-course",
  "version": "1.0.0",
  "description": "",
  "scripts": {
    "start": "parcel 17-Modern-JavaScript-Development-Modules-Tooling-Functional/index.html",
    "build": "parcel build 17-Modern-JavaScript-Development-Modules-Tooling-Functional/index.html"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "leaflet": "^1.8.0",
    "lodash-es": "^4.17.21"
  },
  "devDependencies": {
    "parcel": "^2.6.0"
  }
}

```

Instead of running the `npx parcel index.html` command each time we want to update the parcel output files in the dist folder we can add a script in the package.json file that names the package we want to run – parcel and the relative path from package.json to the file that we want to run the script on.

When we want to update the parcel package then we just type `npm start`.

```

ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course/course-exercices (main)
$ npm start

> complete-javascript-course@1.0.0 start
> parcel 17-Modern-JavaScript-Development-Modules-Tooling-Functional/index.html

Server running at http://localhost:1234
🌟 Built in 1.18s

```

```
ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course/course-exercices (main)
$ npm run build

> complete-javascript-course@1.0.0 build
> parcel build 17-Modern-JavaScript-Development-Modules-Tooling-Functional/index.html

✨ Built in 2.17s

dist\index.html      645 B    772ms
dist\index.14e1e450.js 14 KB    536ms
```

In the package.json file I also added another script that runs npm build on the index.html file.

This compresses the files and removes any unused code.

Configuring Babel and Pollyfilling

Configure babel to convert our super modern code back to ES5 standards because older browsers running on windows XP or windows7 may not support post ES6 code.

Babel is highly configurable up to what browsers can be supported and fortunately parcel comes with babel pre-integrated supporting babel defined defaults called presets. Additionally, we can define with plugins what JavaScript features we want to convert. Note that a preset is a list of plugins that are supported in the Babel install. <https://babeljs.io/docs/en/plugins-list>

We will use the <https://babeljs.io/docs/en/babel-preset-env>.

Note that this preset may only support JavaScript that is part of the official release standard, New features not yet released are not converted to ES5 standard. Babel does not transpile all new features, it only does ES6 to ES5 syntax transpilation.

For full feature conversion we need to Polyfill with another library like coreJs.

<https://babeljs.io/docs/en/caveats>

```
ellio@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course/course-exercices (main)
$ npm install core-js@3 --save

added 1 package, and audited 171 packages in 4s

72 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Polyfill is installed with:
npm install core-js@3 -save
command.

`import "core-js";` Then we need to import the polyfill into our main script.js file.

Writing Clean and Modern JavaScript

READABLE CODE

- write code so that **others** can understand it
- write code so that **you** can understand it in one 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 only do 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

OBJECT ORIENTATED PROGRAMMING

- Use ES6 Classes
- Encapsulate data and don't mutate it from outside the class
- Implement method changing
- Do not use arrow functions as methods (in regular objects)

AVOID NESTED CODE

- use early return (guard clause)
- 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

ASYNCRONOUS CODE

- Consume promises with `async/await` for best readability
- Whenever possible, run promises in parallel(`promise.all`)
- Handle errors and promise rejections

Lets fix some bad code part 1

```
var budget = [
  { 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' },
];

var limits = {
  jonas: 1500,
  matilda: 100,
};

var add = function (value, description, user) {
  if (!user) user = 'jonas';
  user = user.toLowerCase();

  var lim;
  if (limits[user]) {
    lim = limits[user];
  } else {
    lim = 0;
  }

  if (value <= lim) {
    budget.push({ value: -value, description: description, user: user });
  }
};

add(10, 'Pizza 🍕');
add(100, 'Going to movies 🎬', 'Matilda');
add(200, 'Stuff', 'Jay');
console.log(budget);
```

This is the starting code which is full of coding bad practices which we must find and fix.

It is based on a budget which has two users. The users have limits for how much they can spend.

```

var check = function () {
  for (var el of budget) {
    var lim;
    if (limits[el.user]) {
      lim = limits[el.user];
    } else {
      lim = 0;
    }

    if (el.value < -lim) {
      el.flag = 'limit';
    }
  }
};

check();

console.log(budget);

var bigExpenses = function (limit) {
  var output = '';
  for (var el of budget) {
    if (el.value <= -limit) {
      output += el.description.slice(-2) + ' / '; // Emojis are 2 chars
    }
  }
  output = output.slice(0, -2); // Remove last ' / '
  console.log(output);
};

```

This is the starting code which is full of coding bad practices which we must find and fix.

```
var add = function (value, description, user) {  
  if (!user) user = 'jonas';  
  user = user.toLowerCase();  
  var lim;
```

```
  if (limits[user]) {  
    lim = limits[user];  
  } else {  
    lim = 0;  
  }  
}
```

```
  if (value <= lim) {  
    budget.push({ value: -value, description: description,  
user: user });  
  }  
};
```

This function has an ambiguous name so we could change add to addExpense throughout the code.

```
var budget = [  
  { 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' },  
];
```

Next issue is the use of the var variable declaration when we should be using const or let. These are all changed throughout the code.


```
const limits = {  
  jonas: 1500,  
  matilda: 100,  
};
```

This variable has an ambiguous name so we could change limits to spendingLimits throughout the code.

```
const addExpense = function (value, description, user) {  
  if (!user) user = 'jonas';  
  user = user.toLowerCase();
```

```
  let lim;  
  if (spendingLimits[user]) {  
    lim = spendingLimits[user];  
  } else {  
    lim = 0;  
  }
```

`if (!user) user = 'jonas';`
We can use a default parameter value and eliminate this line.

This block of nested code can be substituted for a ternary or better still optional chaining and the nullish operator.

```
  if (value <= lim) {  
    budget.push({ value: -value, description: description, user: user });  
  }  
};
```

Here we are saying that `description = description` and `user = user`. With ES6 Enhanced object literal syntax if the property name and the variable name are the same we only need to put in the variable name.

```

const check = function () {
  for (const el of budget) {
    let lim;
    if (spendingLimits[el.user]) {
      lim = spendingLimits[el.user];
    } else {
      lim = 0;
    }

    if (el.value < -lim) {
      el.flag = 'limit';
    }
  }
};
check();

```

The check function checks all of our budgets to ensure none of them exceed spending limits. We can rename this to checkExpenses.

This block of nested code can be substituted for a ternary or better still optional chaining and the nullish operator.

We are looping over element or el. Better to use entry instead of el.

```
const limit = spendingLimits?.[entry.user] ?? 0;
```

But this is almost identical to some code further up so we can refactor this into its own function rather than repeating ourselves.

```

var bigExpenses = function (limit) {
  var output = '';
  for (var el of budget) {
    if (el.value <= -limit) {
      output += el.description.slice(-2) + ' / '; // Emojis are 2 chars
    }
  }
  output = output.slice(0, -2); // Remove last ' / '
  console.log(output);
};

```

Replace with ternary and template literal.

Declarative and functional JavaScript Principles.

IMPERATIVE

Two fundamentally
different ways of writing
code (paradigms)

DECLARATIVE

- Programmer explains "**HOW** to do things"
- We explain to 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;
```

- Programmer tells "**WHAT** to do"
- We simply describe the way the computer should achieve the result
- the **HOW** (step by step instructions) get extracted away
- **Example:** description of a cake

```
const arr = [2, 4, 6, 8];  
const doubled = arr.map(n => n * 2);
```

Functional programming principles.

FUNCTIONAL PROGRAMMING

- **Declarative** programming paradigm
- based on the idea of writing software by combining many **pure functions**, avoiding **side effects** and **mutating data**
- **side effects**: Modification (mutation) of any data outside of the function (mutating external variables, logging to the console, writing of the 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 transforms with methods such as `.map()`, `.filter()` and `.reduce()`
- Try to avoid side effects in functions: This is not always possible!

DECLARATIVE SYNTAX

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

Lets fix some bad code part 2

Using `Object.freeze` makes `spendingLimits` immutable. We cannot put any new data into it.

```
const spendingLimits = Object.freeze ({
  jonas: 1500,
  matilda: 100,
});
```

We can also use `ObjectFreeze` in the budget array but we will not be able to `array.push` to it. Note that `object freeze` will freeze the top layer but not lower layers, i.e. the properties in the array. For instance I can modify the first object in the array but I cannot add a new object.

```
budget[0].value = 10000;
budget[9] = 'jonas';
```

```
const budget = [
  { 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' },
];

const spendingLimits = {
  jonas: 1500,
  matilda: 100,
};

const getLimit = user => spendingLimits?.[user] ?? 0;

const addExpense = function (value, description, user = 'jonas') {
  user = user.toLowerCase();

  if (value <= getLimit(user)) {
    budget.push({ value: -value, description, user });
  }
};

addExpense(10, 'Pizza 🍕');
addExpense(100, 'Going to movies 🎬', 'Matilda');
addExpense(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(1000);
```

```

const budget = [
  { 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' },
];

const spendingLimits = {
  jonas: 1500,
  matilda: 100,
};

const getLimit = user => spendingLimits?.[user] ?? 0;

const addExpense = function (value, description, user = 'jonas') {
  user = user.toLowerCase();

  if (value <= getLimit(user)) {
    budget.push({ value: -value, description, user });
  }
};

addExpense(10, 'Pizza 🍕');
addExpense(100, 'Going to movies 🎬', 'Matilda');
addExpense(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(1000);

```

The `addExpense` function now has a side effect because we cannot push to the budget array because it is trying to mutate an outside object, the budget array. It is said to be an impure function.

We can fix this by first passing in all the data the function needs to work and then ensure that it does not mutate external data.

We can pass in the state (which is the budget object), limits (which is the `spendingLimits`) and value, description, user.

```

const addExpense = function (
  state,
  limits,
  value,
  description,
  user = 'jonas'
) {

```

```
const budget = [
  { 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' },
];

const spendingLimits = {
  jonas: 1500,
  matilda: 100,
};

const getLimit = user => spendingLimits?.[user] ?? 0;

const addExpense = function (value, description, user = 'jonas') {
  user = user.toLowerCase();

  if (value <= getLimit(user)) {
    budget.push({ value: -value, description, user });
  }
};

addExpense(10, 'Pizza 🍕');
addExpense(100, 'Going to movies 🎬', 'Matilda');
addExpense(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(1000);
```

Here we are manipulating the user value of an object in budget so we want to avoid doing this by defining a new variable for user.

```
const cleanUser = user.toLowerCase();

return value <= getLimit(limits,
  cleanUser)
  ? [...state, { value: -value,
    description, user: cleanUser }]
  : state;
};
```

Now we want to return an array that is a copy of the state array but with a new user added.

```
const newBudget1 = addExpense(budget,
  spendingLimits, 10, 'Pizza 🍕');
```

We can store this new array as newBusget1 and call the addExpense function specifying the state (budget), spendingLimits and the three parameters of the budget array.

```
const budget = [
  { 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' },
];

const spendingLimits = {
  jonas: 1500,
  matilda: 100,
};

const getLimit = user => spendingLimits?.[user] ?? 0;

const addExpense = function (value, description, user = 'jonas') {
  user = user.toLowerCase();

  if (value <= getLimit(user)) {
    budget.push({ value: -value, description, user });
  }
};

addExpense(10, 'Pizza 🍕');
addExpense(100, 'Going to movies 🎬', 'Matilda');
addExpense(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(1000);
```

```
const newBudget2 = addExpense(
  newBudget1,
  spendingLimits,
  100,
  'Going to movies 🎬',
  'Matilda'
);
```

Because we are not mutating the original array of budget but creating a new one we have to pass this in the next time we call addExpense.

```
const newBudget3 = addExpense(
  newBudget2,
  spendingLimits,
  200,
  'Stuff',
  'Jay'
);
```

Also here we are trying to check the original budget array when it is immutable and we are now copying it – newBudget2, newBudget3 etc.. Also We are now trying to set a flag with that original array.


```

const budget = [
  { 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' },
];

const spendingLimits = {
  jonas: 1500,
  matilda: 100,
};

const getLimit = user => spendingLimits?.[user] ?? 0;

const addExpense = function (value, description, user = 'jonas') {
  user = user.toLowerCase();

  if (value <= getLimit(user)) {
    budget.push({ value: -value, description, user });
  }
};

addExpense(10, 'Pizza 🍕');
addExpense(100, 'Going to movies 🎬', 'Matilda');
addExpense(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(1000);

```

Instead of looping over the array we can use one of our data transformation methods – map – which will create a brand new array. We just pass in an array (state) and limits (spending limits) then modify the entry with new property of flag if condition getLimits is true.

```

const checkExpenses = (state, limits) =>
  state.map(entry =>
    entry.value < -getLimit(limits, entry.user)
      ? { ...entry, flag: 'limit' }
      : entry
  );

const finalBudget = checkExpenses(newBudget3,
  spendingLimits);
console.log(finalBudget);

```

We can then call that function storing it in a variable of final budget.

```

const budget = [
  { 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' },
];

const spendingLimits = {
  jonas: 1500,
  matilda: 100,
};

const getLimit = user => spendingLimits?.[user] ?? 0;

const addExpense = function (value, description, user = 'jonas') {
  user = user.toLowerCase();

  if (value <= getLimit(user)) {
    budget.push({ value: -value, description, user });
  }
};

addExpense(10, 'Pizza 🍕');
addExpense(100, 'Going to movies 🎬', 'Matilda');
addExpense(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)} / ` :
      '';

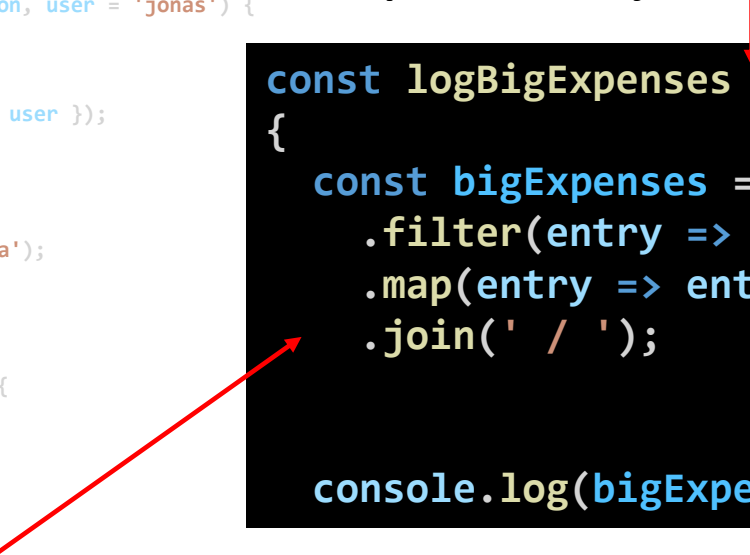
  output = output.slice(0, -2); // Remove last ' / '
  console.log(output);
};

console.log(budget);
logBigExpenses(1000);

```

We have a for loop in the logBigExpenses function and we are always mutating the let variable output by adding to the output the emoji each time we find a big expense.

We first filter the array to find values greater than bigLimit. And build a new array with map of the big expenses then join them into a string.



```

const logBigExpenses = function (state, bigLimit)
{
  const bigExpenses = state
    .filter(entry => entry.value <= -bigLimit)
    .map(entry => entry.description.slice(-2))
    .join(' / ');

  console.log(bigExpenses);
}

```

Forkify App: Building a Modern Application

Part 1 – Core functionality of loading a recipe

Project Overview and planning: User Stories

The goal is to build this website: <https://forkify-v2.netlify.app/>

- **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 have them all organised in the same app
- 4 As a user, I want to be able to **create my own recipes**, so that I have them all organised 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

Project Overview and planning: Features

STORIES

FEATURES

1 Search for Recipes

- Search functionality: input field to send request to API with searched keywords
- Display results with pagination
- display recipe with cooking time, servings and ingredients

2 Update the number of servings

- Change servings functionality: update all ingredients according to current number of servings

3 Bookmark recipes

- Bookmarking functionality: display list of all bookmarked recipes

4 Create my own recipes

- User can upload own recipes
- User recipes will automatically be bookmarked
- user can only see their own recipes, not recipes from other users.

5 See my bookmarks and own recipes when I leave the app and come back

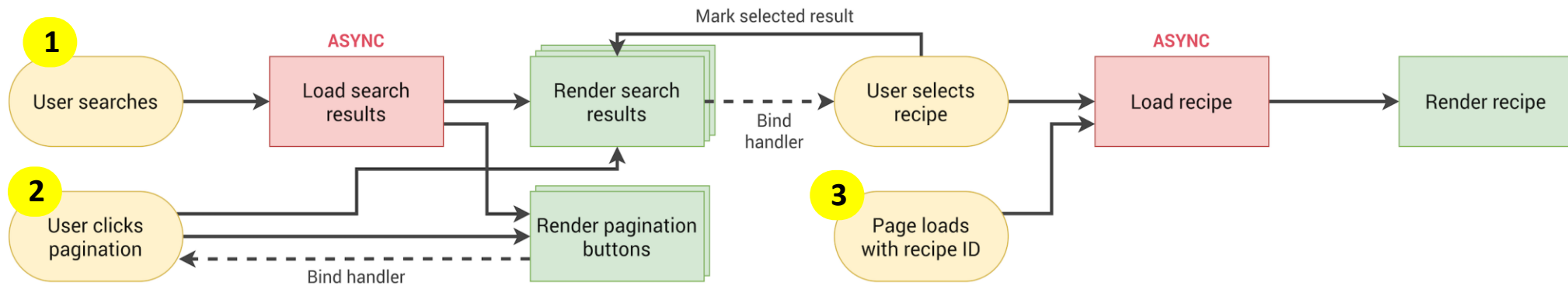
- Store bookmark data in the browser using local storage
- On page load, read saved bookmarks from local storage and display

Project Overview and planning: Flow Chart part 1

1 Search functionality: API search request

2 Results with Pagination

3 Display Recipe



Loading a recipe from API: Setting up parcel

1) I change to the directory where the forkify app is located and initialise NPM filling in the question prompts to produce the package.json file (shown right).

```
ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course/course-exercices (main)
$ cd 18-Foryify-App-Building-A-Modern-Application/

ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course/course-exercices/18-Foryify-App-Building-A-Modern-Application (main)
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.
```

```
{
  "name": "forkify",
  "version": "1.0.0",
  "description": "Recipe application",
  "scripts": {
    "start": "parcel index.html",
    "build": "parcel build index.html"
  },
  "author": "Elliott Farmer",
  "license": "ISC",
  "devDependencies": {
    "@parcel/transformer-sass":
    "^2.6.0",
    "parcel": "^2.6.0"
  }
}
```

2) I next run *npm install parcel@next -d* to load the parcel package and then fix any issues with a *npm audit fix* command as instructed by the onscreen suggestions. AS parcel version two is the latest I update the packages with *npm install parcel@2 -d* (see following slide)
Note: When parcel has finished installing the package.json file will be updated with devDependencies on parcel.

```
ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course/course-exercices/18-Foryify-App-Building-A-Modern-Application (main)
```

```
$ npm install parcel@next -D
```

```
npm WARN deprecated source-map-url@0.4.1: See https://github.com/lydell/source-map-url#deprecated
```

```
npm WARN deprecated request-promise-native@1.0.9: request-promise-native has been deprecated because it extends the now deprecated request package, see https://github.com/request/request/issues/3142
```

```
npm WARN deprecated urix@0.1.0: Please see https://github.com/lydell/urix#deprecated
```

```
npm WARN deprecated har-validator@5.1.5: this library is no longer supported
```

```
npm WARN deprecated source-map-resolve@0.5.3: See https://github.com/lydell/source-map-resolve#deprecated
```

```
npm WARN deprecated resolve-url@0.2.1: https://github.com/lydell/resolve-url#deprecated
```

```
npm WARN deprecated querystring@0.2.1: The querystring API is considered Legacy. new code should use the URLSearchParams API instead.
```

```
npm WARN deprecated uuid@3.4.0: Please upgrade to version 7 or higher. Older versions may use Math.random() in certain circumstances, which is known to be problematic. See https://v8.dev/blog/math-random for details.
```

```
npm WARN deprecated request@2.88.2: request has been deprecated, see https://github.com/request/request/issues/3142
```

```
npm WARN deprecated querystring@0.2.0: The querystring API is considered Legacy. new code should use the URLSearchParams API instead.
```

```
added 791 packages, and audited 792 packages in 45s
```

```
138 packages are looking for funding  
run `npm fund` for details
```

```
46 vulnerabilities (41 moderate, 2 high, 3 critical)
```

```
To address all issues, run:  
npm audit fix
```

```
Run `npm audit` for details.
```

```
ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course/course-exercices/18-Foryify-App-Building-A-Modern-Application (main)
```

```
$ npm audit fix
```

```
added 22 packages, removed 646 packages, changed 53 packages, and audited 168 packages in 16s
```

```
71 packages are looking for funding  
run `npm fund` for details
```

```
found 0 vulnerabilities
```



```

ellio@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course/course-exercices/18-Foryify-App-Building-A-Modern-Application (main)
$ npm run start

> forlify@1.0.0 start
> parcel index.html

Server running at http://localhost:1234
🔴 Build failed.

@parcel/core: Unexpected output file type .html in target "main"

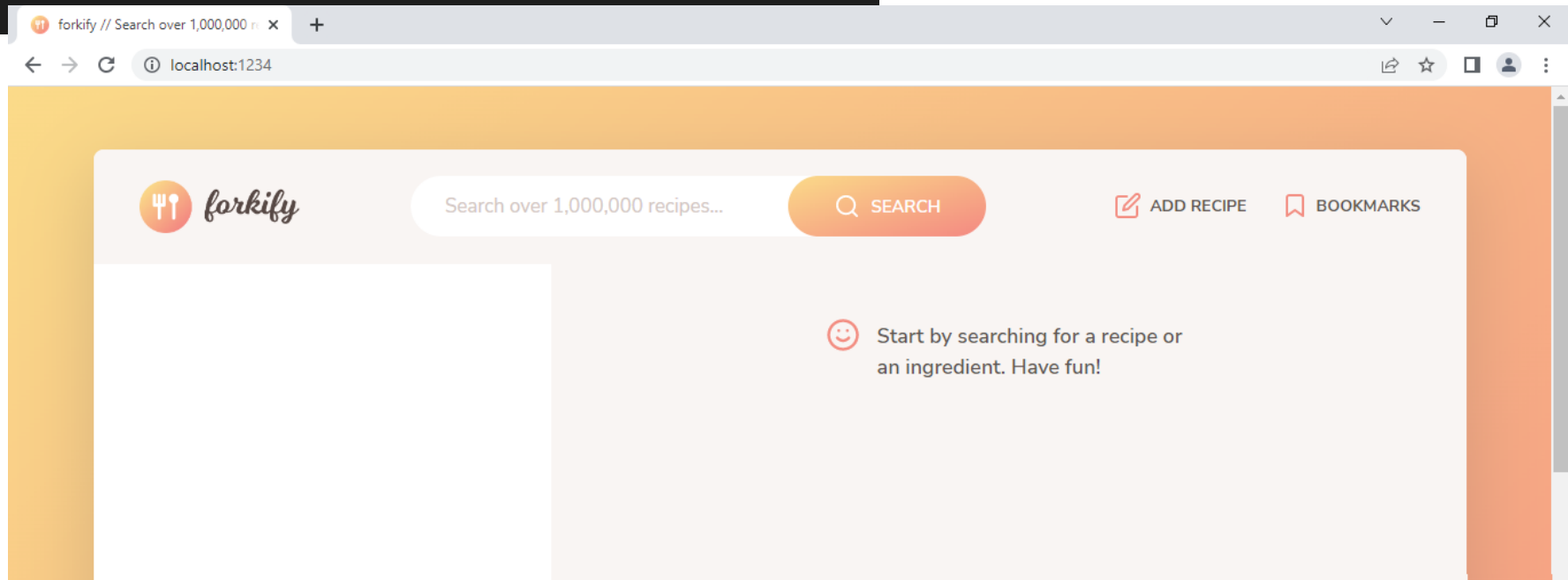
C:\Users\ellio\Documents\CODING-LESSONS\05-javascript\05-javascript-lessons\Complete-Javascript-Course\course-exercices\18-Foryify-App-Building-A-Modern-Application\package.json:5:11
   4 |     "description": "Recipe application",
   > 5 |     "main": "index.html",
     |           ^
Server running at http://localhost:1234
🌟 Built in 800ms

```

3) Next *npm run start* to start parcel. This gives me an error which can be fixed by removing the “main”: index.html line from the package.json file.

Now it is running and gives me an URL which I can use to view the page locally.

To terminate this process use *control + C* in the terminal.



Loading a recipe from API

```
const showRecipe = async function () {
  try {
    const res = await fetch(
      'https://forkify-api.herokuapp.com/api/v2/recipes/5ed6604591c37cdc054bc886'
    );
    const data = await res.json();
    if (!res.ok) throw new Error(`${data.message} (${res.status})`);
    let { recipe } = data.data;
    recipe = {
      id: recipe.id,
      title: recipe.title,
      publisher: recipe.publisher,
      sourceUrl: recipe.source_url,
      image: recipe.image_url,
      servings: recipe.servings,
      cookingTime: recipe.cooking_time,
      ingredients: recipe.ingredients,
    };
    console.log(recipe);
  } catch (err) {
    alert(err);
  }
};

showRecipe();
```

From the API documentation we can find a list of API calls that we can perform.
<https://forkify-api.herokuapp.com/v2>

As in previous lessons we create an async API call using a try catch block and await. We code in the error handling encase the API returns an error.

The API response is put in a variable called data which is then destructured into an object with our defined property names.

Rendering a recipe

A markup variable is created and set to a template literal containing a block of HTML code where we can insert our destructured values from the API call.

```
const markup = `
  <figure class="recipe__fig">
    
    <h1 class="recipe__title">
      <span>${recipe.title}</span>
    </h1>
  </figure>

  <div class="recipe__details">
    <div class="recipe__info">
      <svg class="recipe__info-icon">
        <use href="src/img/icons.svg#icon-clock"></use>
      </svg>
      <span class="recipe__info-data recipe__info-data--minutes">${recipe.cookingTime}</span>
      <span class="recipe__info-text">minutes</span>
    </div>
    <div class="recipe__info">
      <svg class="recipe__info-icon">
        <use href="src/img/icons.svg#icon-users"></use>
      </svg>
      <span class="recipe__info-data recipe__info-data--people">${recipe.servings}</span>
      <span class="recipe__info-text">servings</span>
    </div>
  </div>
`
```

```

<div class="recipe__directions">
  <h2 class="heading--2">How to cook it</h2>
  <p class="recipe__directions-text">
    This recipe was carefully designed and tested by
    <span class="recipe__publisher">${recipe.publisher}</span>. Please check out
    directions at their website.
  </p>
  <a
    class="btn--small recipe__btn"
    href="${recipe.sourceUrl}"
    target="_blank"
  >
    <span>Directions</span>
    <svg class="search__icon">
      <use href="src/img/icons.svg#icon-arrow-right"></use>
    </svg>
  </a>
</div>
`;

```

```
recipeContainer.insertAdjacentHTML('afterbegin', markup);
```

We use insertAdjacentHTML to add the contents of the markup variable to the recipeContainer.

Now the recipe is shown on the page minus the ingredients which are currently in an array and need more work to destructure and display. We also need to fix the icons.

😊 Start by searching for a recipe or an ingredient. Have fun!

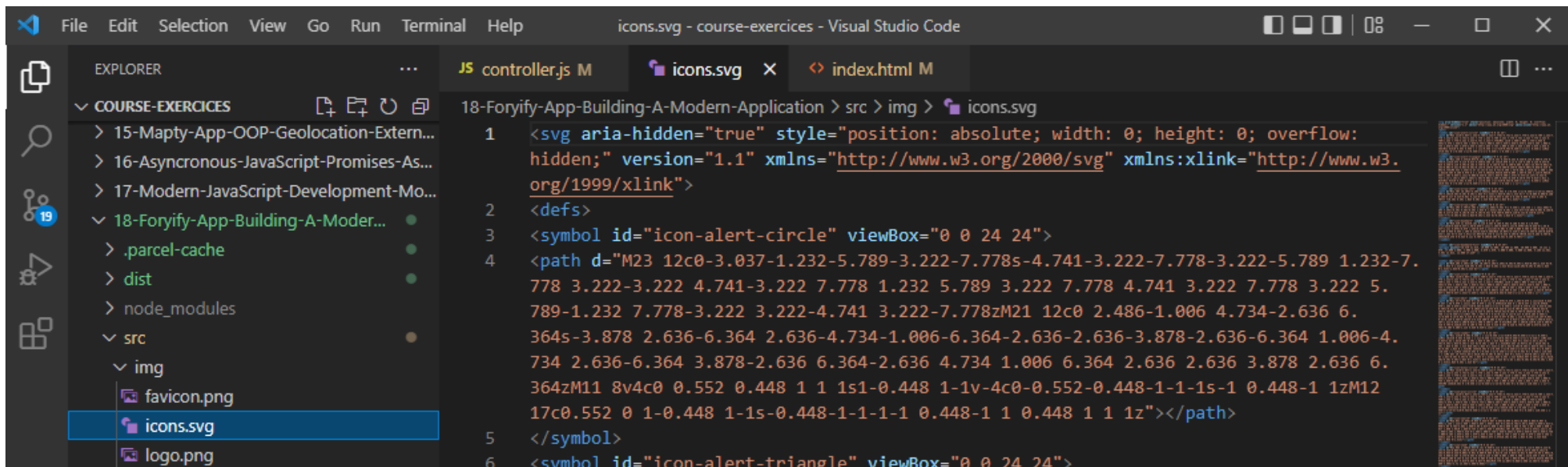
But we still have the original html message in the window so we need to get rid of that by first emptying the recipeContainer before inserting our JavaScript created html block.

```
recipeContainer.innerHTML = '';
recipeContainer.insertAdjacentHTML('afterbegin', markup);
```

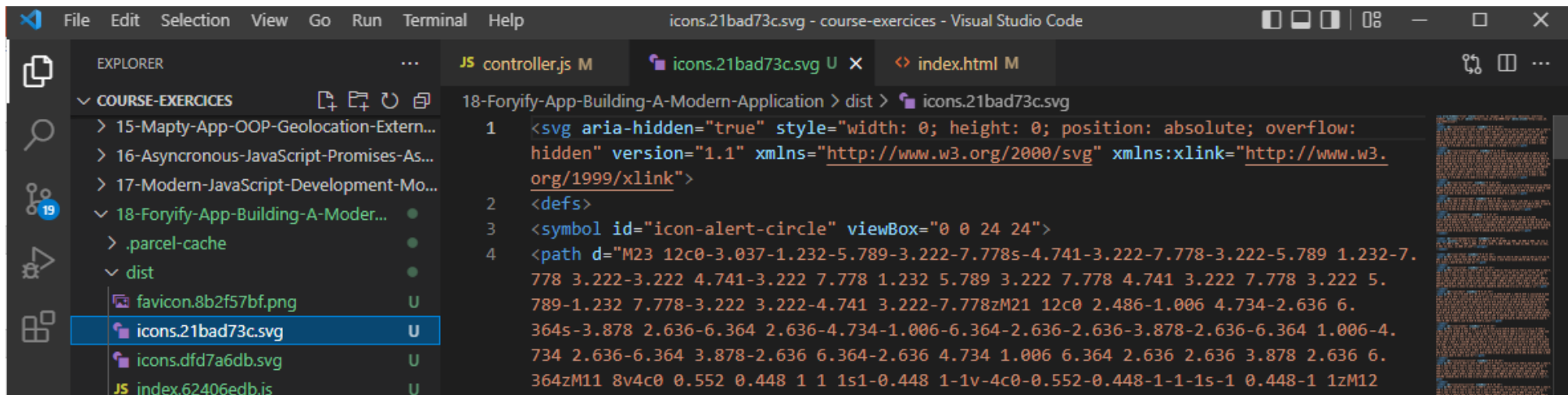
```
<div class="recipe__ingredients">
  <h2 class="heading--2">Recipe ingredients</h2>
  <ul class="recipe__ingredient-list">
    ${recipe.ingredients
      .map(ing => {
        return `
          <li class="recipe__ingredient">
            <svg class="recipe__icon">
              <use href="src/img/icons.svg#icon-check"></use>
            </svg>
            <div class="recipe__quantity">${ing.quantity}</div>
            <div class="recipe__description">
              <span class="recipe__unit">${ing.unit}</span>
              ${ing.description}
            </div>
          </li>
        </ul>
      `;
    })
    .join('')}
  </div>
```

Now we need to loop over the ingredients array and create a block of html for each ingredient.

The .map() method is the best because we can return a template literal then .join() each iteration together into a long string.



The icons come from icons.svg file in the img folder



Which now has a completely different name and path in the dist folder

```
// import icons from '../img/icons.svg'; // parcel.v1
import icons from 'url:../img/icons.svg'; // parcel.v2
```

```
<button class="btn--tiny btn--increase-servings">
  <svg>
    <use href="${icons}#icon-minus-circle"></use>
  </svg>
</button>
<button class="btn--tiny btn--increase-servings">
  <svg>
    <use href="${icons}#icon-plus-circle"></use>
  </svg>
</button>
```

```
const renderSpinner = function (parentEl) {
  const markup = `
    <div class="spinner">
      <svg>
        <use href="${icons}#icon-loader"></use>
      </svg>
    </div>
  `;
  parentEl.innerHTML = '';
  parentEl.insertAdjacentHTML('afterbegin', markup);
};
```

The solution is to import the icons.svg file

And anywhere in the HTML block in the JavaScript code we now reference the imported icons instead of the file path to the local copy of the icons.svg file.

We create a generic function that loads a spinner over any images (or parent element) while they are loading. Note that the spinner is a static image that uses css transform to rotate it through 260 degrees.

```
ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course/course-exercices (main)
$ cd 18-Foryify-App-Building-A-Modern-Application/

ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course/course-exercices/18-Foryify-App-Building-A-Modern-Application (main)
$ npm install core-js regenerator-runtime

added 1 package, and audited 187 packages in 9s

75 packages are looking for funding
  run `npm fund` for details

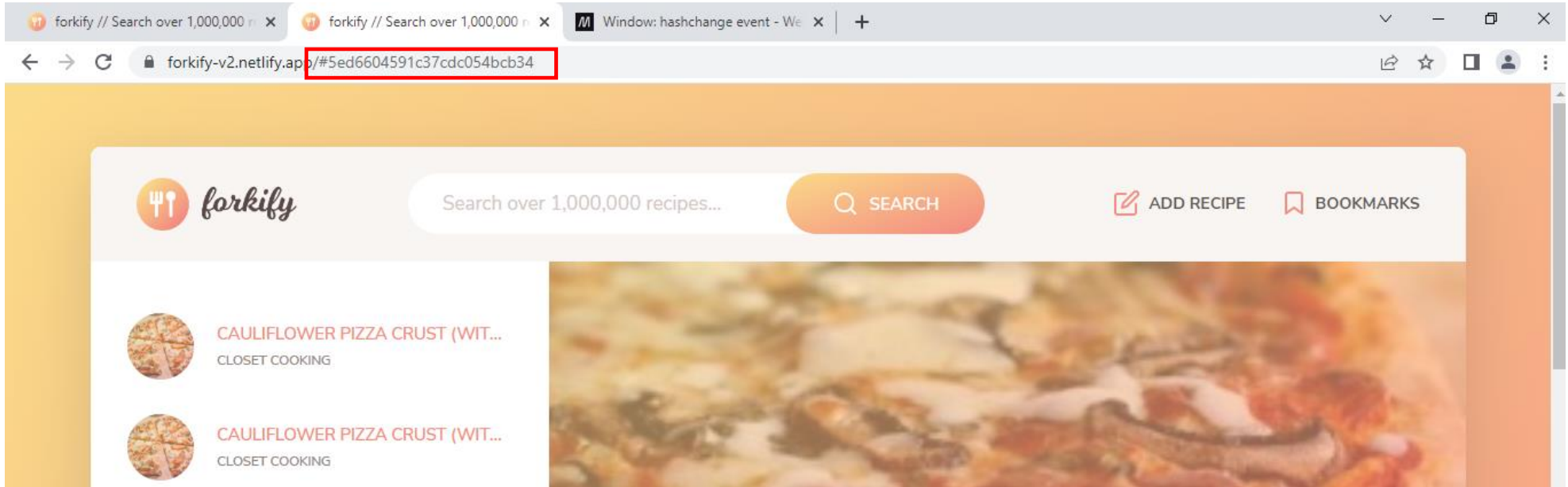
found 0 vulnerabilities

ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course/course-exercices/18-Foryify-App-Building-A-Modern-Application (main)
$ █
```

We need a couple more packages: core-js (for ES6 polyfills) and regenerator-runtime

Listening for Load and Hashtag Events

Window.location is basically the whole URL but we want the part after the hash which is the recipe id. We then insert the recipe id into the API call to load the relevant recipe



```
<div class="search-results">
  <ul class="results">
    <a href="#5ed6604591c37cdc054bcd09">RECIPE</a><br><br>
    <a href="#5ed6604591c37cdc054bcb37">RECIPE</a>
```

For testing purposes we insert some hash links into the html file.

```

const showRecipe = async function () {
  try {
    const id = window.location.hash.slice(1);
    console.log(id);
    // 1. Loading Recipe
    renderSpinner(recipeContainer);
    const res = await fetch(
      `https://forkify-api.herokuapp.com/api/v2/recipes/${id}`
    );
    const data = await res.json();
    if (!res.ok) throw new Error(`${data.message} (${res.status})`);
    let { recipe } = data.data;
    recipe = {
      id: recipe.id,
      title: recipe.title,
      publisher: recipe.publisher,
      sourceUrl: recipe.source_url,
      image: recipe.image_url,
      servings: recipe.servings,
      cookingTime: recipe.cooking_time,
      ingredients: recipe.ingredients,
    };
    // console.log(recipe);
  }
}

```

In the show recipe function we get the hash from the URL and assign it to an id variable minus the leading hash.

We can then put the id value into the API call URL.

Now we can add an event listener that will call the showRecipe function when it detects a hash change (when we click on one of the previously created testing hrefs.)

```

window.addEventListener('hashchange', showRecipe);

```

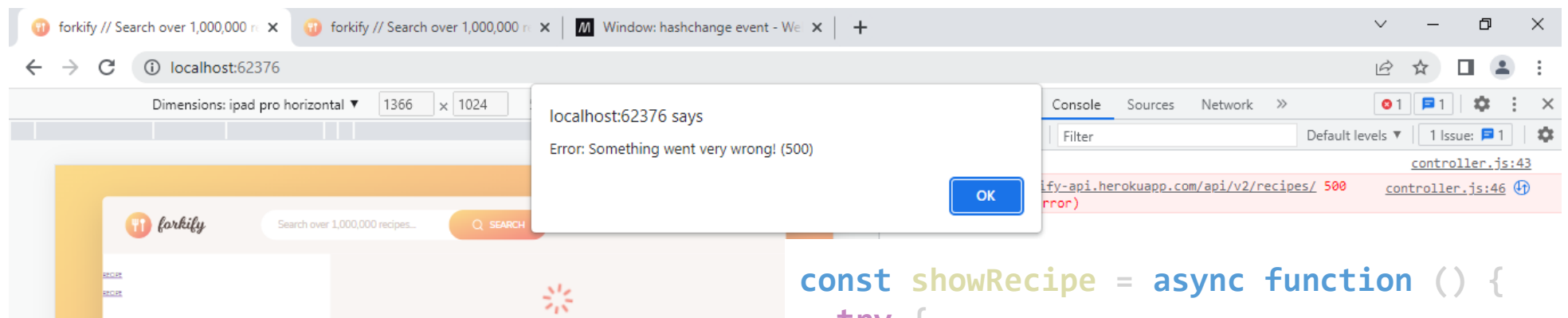
Now we can add an event listener that will call the showRecipe function when it detects a hash change (when we click on one of the previously created testing hrefs.)

```
window.addEventListener('hashchange', showRecipe);  
window.addEventListener('load', showRecipe);
```

But we also want the same showRecipe function to fire on a window load event. This could get tiresome if we have 10 events all calling the same function!

```
['hashchange', 'load'].forEach(ev => window.addEventListener(ev, showRecipe));
```

Fortunately, we can put the events in an array and then loop over them with a forEach.



If there is no recipe id (hash) in the URL then we get the alert and the page gets stuck on the loading spinner.

We can solve this with a guard clause that returns from the function before the loading spinner is called.

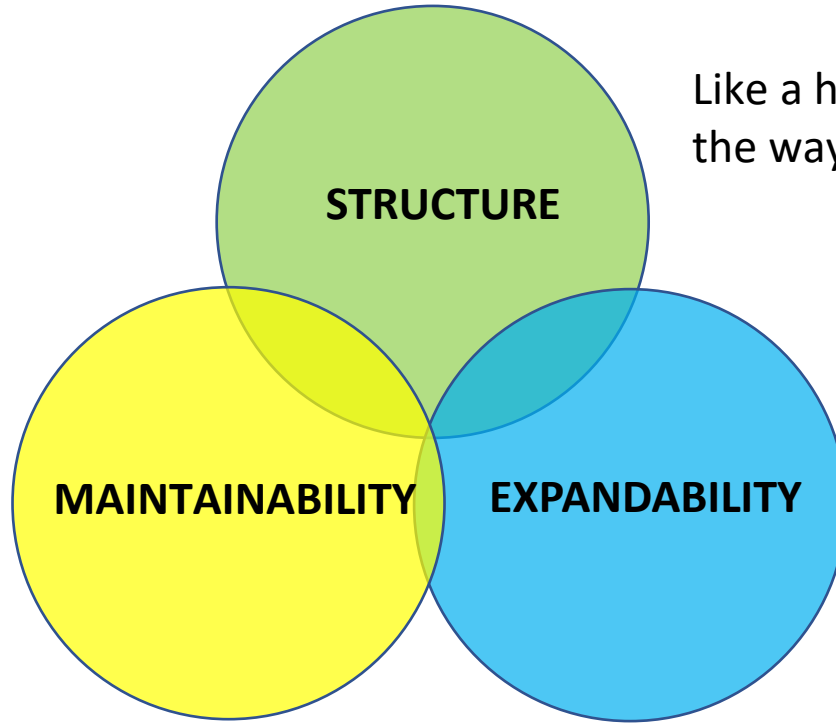
```
const showRecipe = async function () {  
  try {  
    const id = window.location.hash.slice(1);  
    console.log(id);  
  
    if (!id) return;  
  
    // 1. Loading Recipe  
    renderSpinner(recipeContainer);
```

Forkify App: Building a Modern Application

Part 2 – Implementing MVC model

The MVC architecture

A project is never done! We need to be able to easily **change it in the future!**



Like a house, software needs a structure, the way **we organise our code**

We also need to be able to easily **add new features**

- A project is never done! We need to be able to easily change it in the future
- We can create our own architecture (Mapty Project)
- We can use a well established architecture pattern like MVC, Flux, etc (this project)
- We can use a framework like React, Angular, Vue, svelte, etc.



The Components of any Architecture

BUSINESS LOGIC

- Code that solves the actual business problem;
- Directly related to what the business does and what it needs;
- Example: sending messages, storing transactions, calculating taxes...

STATE

- Essentially stores all the data about the application
- should be the single source of truth
- UI should be kept in sync with the state
- state libraries exist

Always
in sync

HTTP LIBRARY

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

APPLICATION LOGIC (ROUTER)

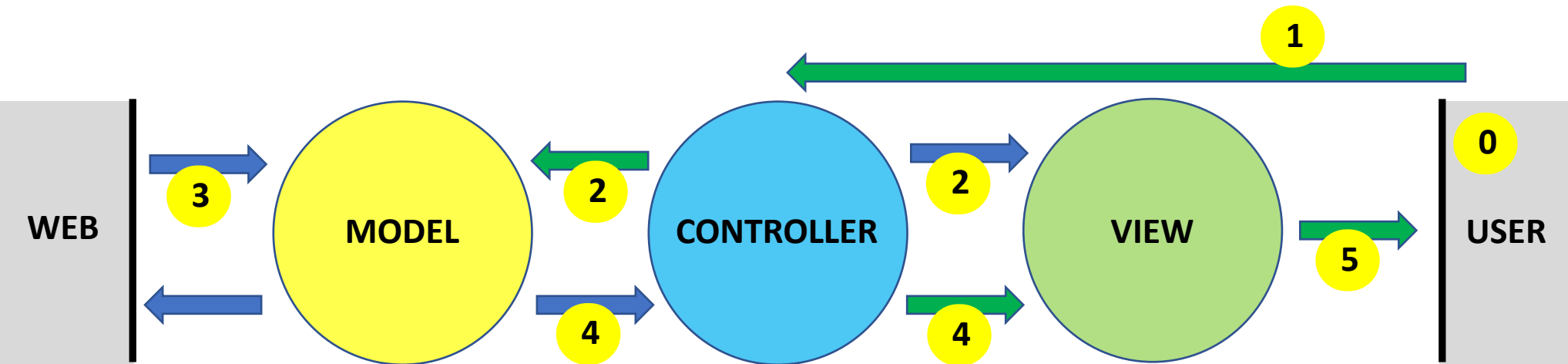
- Code that is only concerned about the implementation of the application itself
- handles navigation and UI events

PRESENTATION LOGIC (UI LAYER)

- Code that is concerned about the visible part of the application
- Essentially displays application state

The Model-View-Controller (MVC) Architecture

 Connected by a function call and import  Data flow



BUSINESS LOGIC

**APPLICATION LOGIC
(ROUTER)**

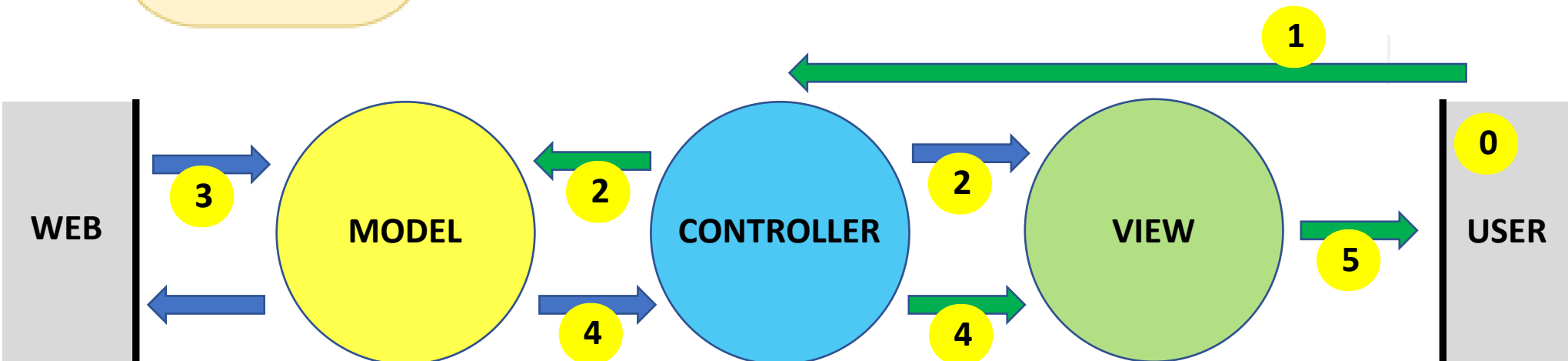
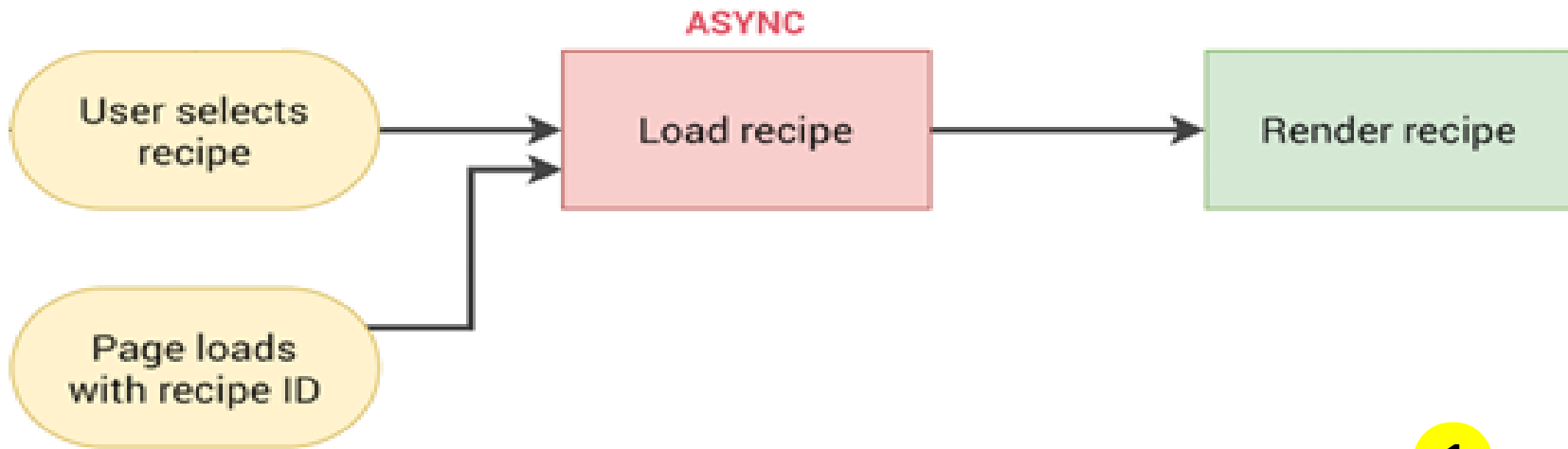
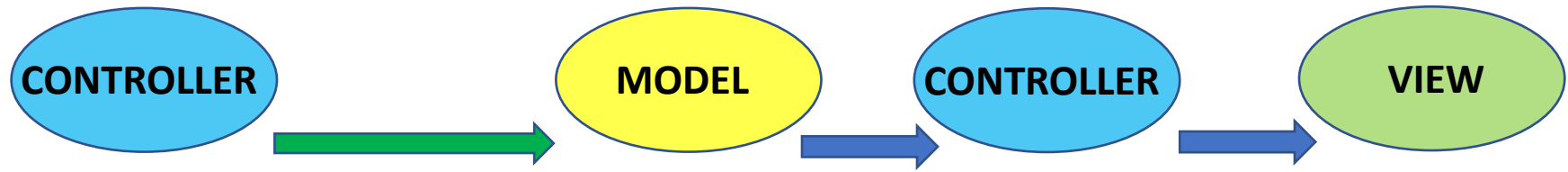
**PRESENTATION LOGIC
(UI LAYER)**

STATE

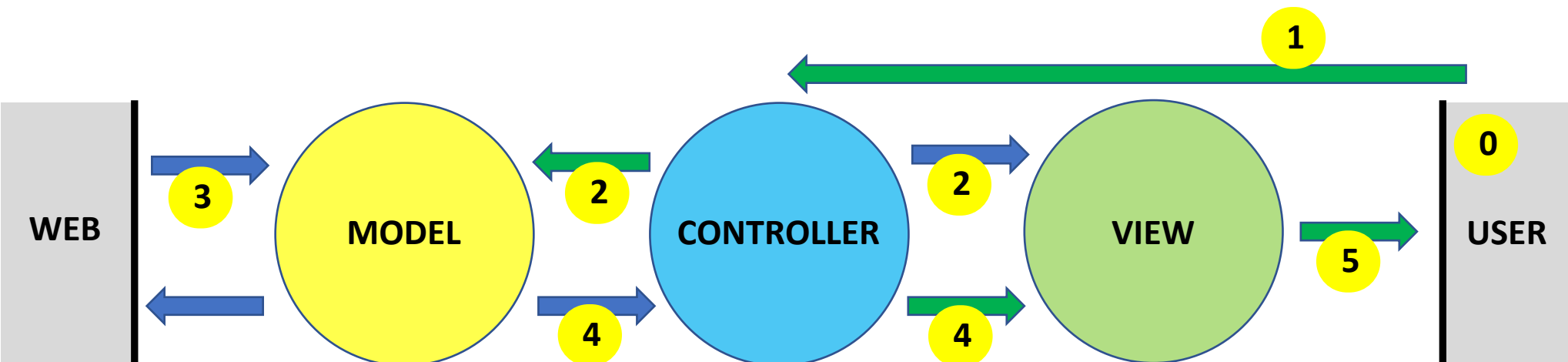
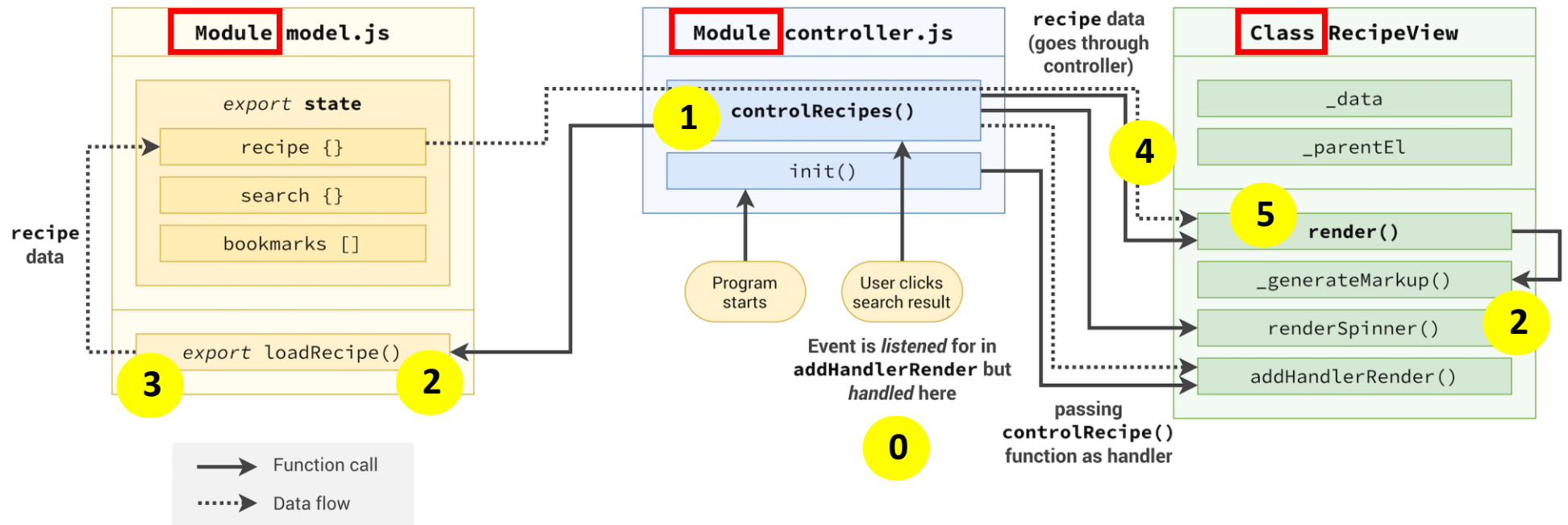
HTTP LIBRARY

- Bridge between model and views (which don't know about one another)
- Handles UI events and dispatches tasks to model and view

Model, View and Controller in Forkify App

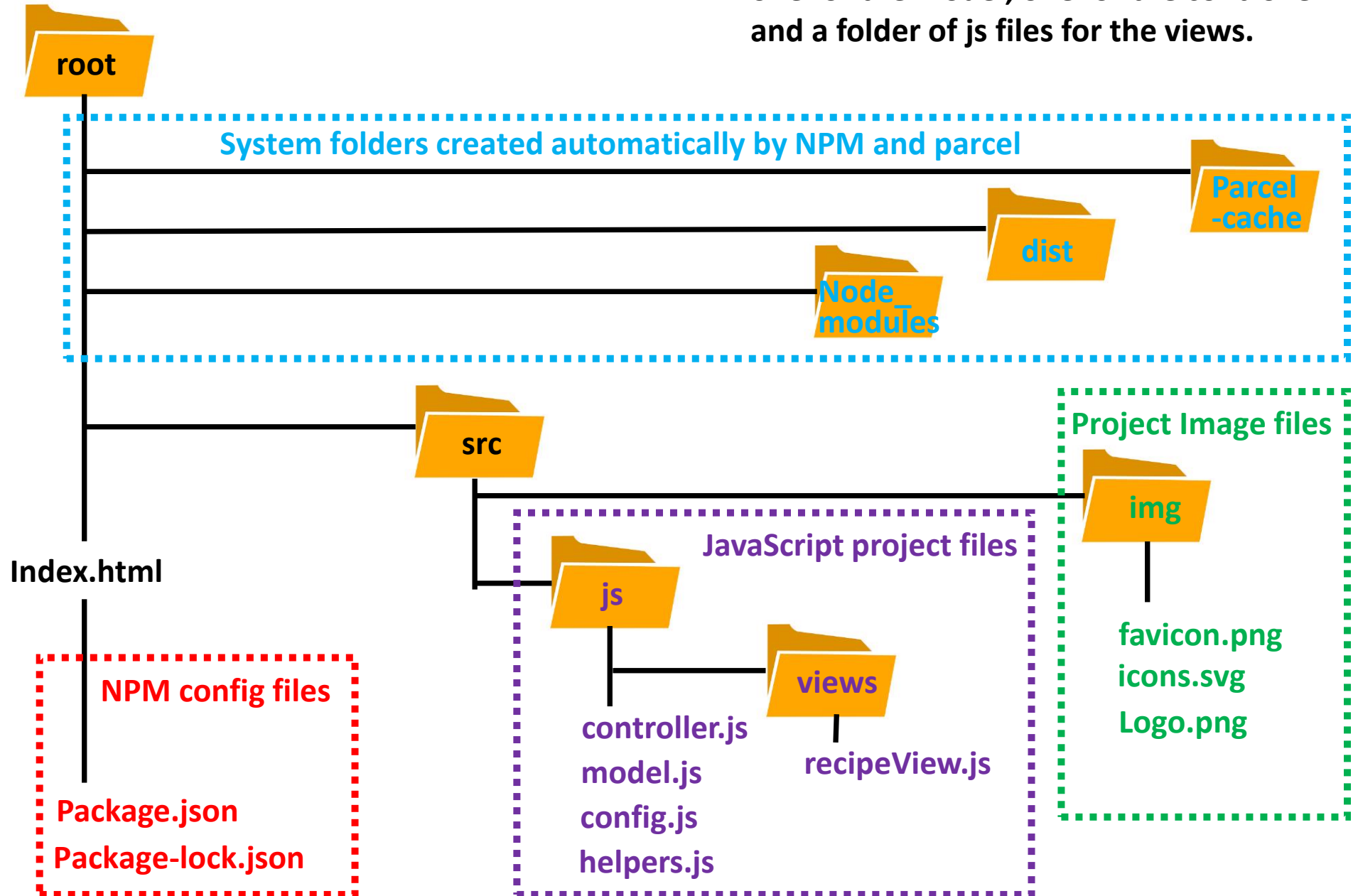


MVC implementation in Forkify App



Refactoring for the MVC model

The file is replaced with three or more files, one for the model, one for the controller and a folder of js files for the views.



```
// model.js - API/state handling
import { async } from 'regenerator-runtime';

export const state = { recipe: {} };

export const loadRecipe = async function (id) {
  try {
    const res = await fetch(
      `https://forkify-api.herokuapp.com/api/v2/recipes/${id}`
    );
    const data = await res.json();
    if (!res.ok) throw new Error(`${data.message} (${res.status})`);
    const { recipe } = data.data;
    state.recipe = {
      id: recipe.id,
      title: recipe.title,
      publisher: recipe.publisher,
      sourceUrl: recipe.source_url,
      image: recipe.image_url,
      servings: recipe.servings,
      cookingTime: recipe.cooking_time,
      ingredients: recipe.ingredients,
    };
    // console.log(state.recipe);
  } catch (err) {
    alert(err);
  }
};
```

The parts of the JS code that handle the state are separated into the model JS file and export state and fetch functions.

```
// Controler.js - Handles communication between model and view
```

```
import * as model from './model.js';
```

we can now import everything from the model. This included the exports such as state and loadRecipie so in the controller we will end up with model.state and model.loadRecipie

```
import icons from 'url:../img/icons.svg'; // parcel.v2
```

```
import 'core-js/stable';
```

```
import 'regenerator-runtime/runtime';
```

```
...  
...  
...
```

```
const showRecipe = async function () {  
  try {  
    const id = window.location.hash.slice(1);  
    // console.log(id);
```

```
    if (!id) return;  
    renderSpinner(recipeContainer);
```

```
    // 1. Loading Recipe
```

```
    await model.loadRecipie(id);
```

```
    const { recipe } = model.state;
```

The show recipe function can now be modified to await the loadRecipie from the model and we pass in an id.

For the moment we can destructure the recipe from model.state.

At this point the application should be working as before.

```
// recipeView.js - the class for recipes
import icons from 'url:../../img/icons.svg'; // parcel.v2

class RecipeView {
  #parentElement = document.querySelector('.recipe');
  #data;

  render(data) {
    this.#data = data;
    const markup = this.#generatemarkup();
    this.#clear;
    this.#parentElement.insertAdjacentHTML('afterbegin', markup);
  }

  #clear() {
    this.#parentElement.innerHTML = '';
  }

  renderSpinner = function () {
    const markup = `
      <div class="spinner">
        <svg>
          <use href="${icons}#icon-loader"></use>
        </svg>
      </div>
    `;
    this.#parentElement.innerHTML = '';
    this.#parentElement.insertAdjacentHTML('afterbegin', markup);
  };
};
```

We can take the icons out of the controller and put them in the view adjusting the path to go up two levels from the src/js/views folder.

The recipeView is a class where we first define the parent element and data as private properties

Within this class we have a public method to render the data. We define the data as this. We set a markup variable and dump all the markup html code in it.

We have a separate private method to first clear the html from the parent element then insert the html.

Render spinner is also a view related function so it also goes in the RecipeView class as a public method.

The generateMarkup method is a private method within the class.
Note that now we have changed the template literals to this.#data.

```
#generateMarkup() {  
  return `  
    <figure class="recipe__fig">  
        
      <h1 class="recipe__title">  
        <span>${this.#data.title}</span>  
      </h1>  
    </figure>  
  
    <div class="recipe__details">  
      <div class="recipe__info">  
        <svg class="recipe__info-icon">  
          <use href="${icons}#icon-clock"></use>  
        </svg>  
        <span class="recipe__info-data recipe__info-data--minutes">${ this.#data.cookingTime}</span>  
        <span class="recipe__info-text">minutes</span>  
      </div>  
      <div class="recipe__info">  
        <svg class="recipe__info-icon">  
          <use href="${icons}#icon-users"></use>  
        </svg>  
        <span class="recipe__info-data recipe__info-data--people">${this.#data.servings}</span>  
        <span class="recipe__info-text">servings</span>  
      </div>  
    </div>  
  `;  
}
```

```
<div class="recipe__info-buttons">
  <button class="btn--tiny btn--increase-servings">
    <svg>
      <use href="{icons}#icon-minus-circle"></use>
    </svg>
  </button>
  <button class="btn--tiny btn--increase-servings">
    <svg>
      <use href="{icons}#icon-plus-circle"></use>
    </svg>
  </button>
</div>
</div>
```

```
<div class="recipe__user-generated">
  <svg>
    <use href="{icons}#icon-user"></use>
  </svg>
</div>
<button class="btn--round">
  <svg class="">
    <use href="{icons}#icon-bookmark-fill"></use>
  </svg>
</button>
</div>
```

```
<div class="recipe__ingredients">
  <h2 class="heading--2">Recipe ingredients</h2>
  <ul class="recipe__ingredient-list">
    ${this.#data.ingredients
      .map(ing => {
        return `
          <li class="recipe__ingredient">
            <svg class="recipe__icon">
              <use href="${icons}#icon-check"></use>
            </svg>
            <div class="recipe__quantity">${ing.quantity}</div>
            <div class="recipe__description">
              <span class="recipe__unit">${ing.unit}</span>
              ${ing.description}
            </div>
          </li>
        </ul>
      `;
    }).join('')}
  </div>
```



```

<div class="recipe__directions">
  <h2 class="heading--2">How to cook it</h2>
  <p class="recipe__directions-text">
    This recipe was carefully designed and tested by
    <span class="recipe__publisher">${this.#data.publisher}</span>. Please check out
    directions at their website.
  </p>
  <a
    class="btn--small recipe__btn"
    href="${this.#data.sourceUrl}"
    target="_blank"
  >
    <span>Directions</span>
    <svg class="search__icon">
      <use href="${icons}#icon-arrow-right"></use>
    </svg>
  </a>
</div>
;
}
}

```

export default new RecipeView(); ← We can export the RecipeView class as a default.

```
// Controller.js - Handles communication between model and view
```

```
import * as model from './model.js';
```

```
import recipeView from './views/recipeView.js';
```

← We can now import the RecipeView class

```
import 'core-js/stable';
```

```
import 'regenerator-runtime/runtime';
```

```
const recipeContainer = document.querySelector('.recipe');
```

```
const controlRecipes = async function () {
```

← ShowRecipe function is renamed to controlRecipes

```
  try {
```

```
    const id = window.location.hash.slice(1);  
    // console.log(id);
```

```
    if (!id) return;
```

```
    recipeView.renderSpinner();
```

← Now we render the spinner from the RecipeView class in the recipeView.js file

```
    // 1. Loading Recipe
```

```
    await model.loadRecipe(id);
```

← We load the recipe from the model

```
    // 2. Rendering recipe
```

```
    recipeView.render(model.state.recipe);
```

← We render the recipe from the RecipeView class in the recipeView.js file

```
  } catch (err) {
```

```
    alert(err);
```

```
  }
```

```
};
```

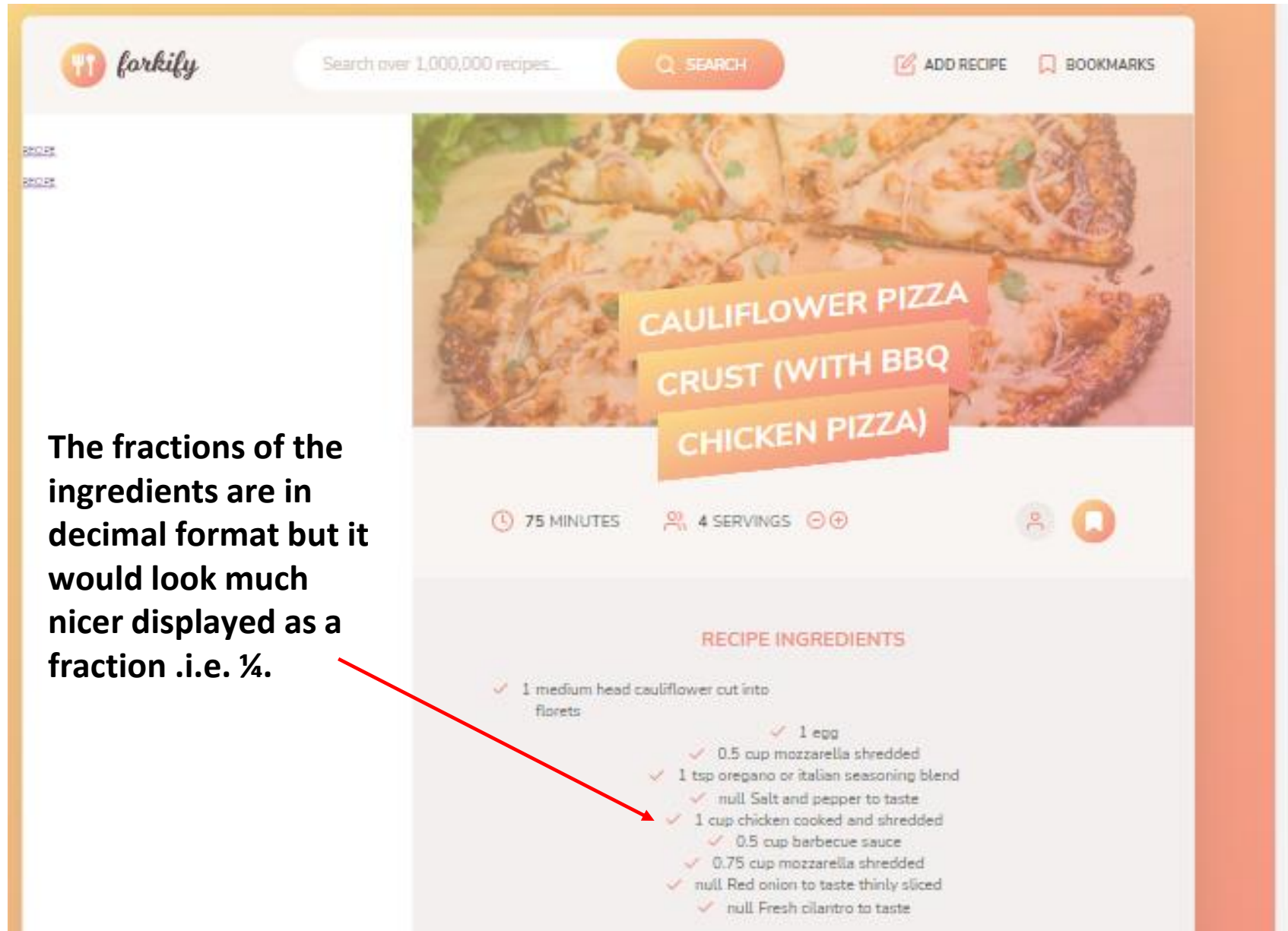
```
['hashchange', 'load'].forEach(ev =>
```

```
  window.addEventListener(ev, controlRecipes)
```

←

```
);
```

Fixing an aesthetic bug using an external library



The screenshot shows the Forkify website interface. At the top, there's a search bar with the text "Search over 1,000,000 recipes...", a "SEARCH" button, and links for "ADD RECIPE" and "BOOKMARKS". The main content area features a large image of a cauliflower pizza with the title "CAULIFLOWER PIZZA CRUST (WITH BBQ CHICKEN PIZZA)". Below the image, it shows "75 MINUTES" and "4 SERVINGS". The "RECIPE INGREDIENTS" section lists the following items:

- ✓ 1 medium head cauliflower cut into florets
- ✓ 1 egg
- ✓ 0.5 cup mozzarella shredded
- ✓ 1 tsp oregano or italian seasoning blend
- ✓ null Salt and pepper to taste
- ✓ 1 cup chicken cooked and shredded
- ✓ 0.5 cup barbecue sauce
- ✓ 0.75 cup mozzarella shredded
- ✓ null Red onion to taste thinly sliced
- ✓ null Fresh cilantro to taste

A red arrow points from the text block on the left to the decimal values "0.5" and "0.75" in the ingredients list.

The fractions of the ingredients are in decimal format but it would look much nicer displayed as a fraction .i.e. $\frac{1}{2}$.

<https://www.npmjs.com/package/fractional> package can be installed with a `npm install fractional` command.

```
ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course/course-exercises/18-Foryify-App-Building-A-Modern-Application (main)
$ npm install fractional

added 1 package, and audited 188 packages in 2s

75 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

We can see that fractional has been added to our dependencies in package.json.

With this package we can convert a decimal to a fraction then to a string, i.e. 0.5 -> ½

```
{
  "name": "forkify",
  "version": "1.0.0",
  "description": "Recipe application",
  "scripts": {
    "start": "parcel index.html",
    "build": "parcel build index.html"
  },
  "author": "Elliott Farmer",
  "license": "ISC",
  "devDependencies": {
    "@parcel/transformer-sass": "^2.6.0",
    "parcel": "^2.6.0"
  },
  "dependencies": {
    "core-js": "^3.22.8",
    "fractional": "^1.0.0",
    "regenerator-runtime": "^0.13.9"
  }
}
```

from the guidance notes on <https://www.npmjs.com/package/fractional> we see that it advises to import Fraction via the old JavaScript require method so we need to import it in our project in the recipeView.js file (it is related to view on our MVC model!). We can destructure it directly in the import. *var Fraction = require('fractional').Fraction*

```
// recipeView.js - the class for recipes
import icons from 'url:../../img/icons.svg';
import { Fraction } from 'fractional';
console.log(Fraction);
```

```
f (numerator, denominator) { recipeView.js:4
  /* double argument invocation */ if (typeof
numerator !== "undefined" && denominator) {
    if (typeof numerator === "number" && typeof
denominator === "number") {
    ...
```

```
<div class="recipe__ingredients">
  <h2 class="heading--2">Recipe ingredients</h2>
  <ul class="recipe__ingredient-list">
    ${this.#data.ingredients
      .map(ing => {
        return `
          <li class="recipe__ingredient">
            <svg class="recipe__icon">
              <use href="${icons}#icon-check"></use>
            </svg>
            <div class="recipe__quantity">${
              ing.quantity ? new Fraction(ing.quantity).toString() : ''
            }</div>
            <div class="recipe__description"><span class="recipe__unit">${ing.unit}</span>
              ${ing.description}
            </div>
          </li>
        </ul>
      `;
    }).join('')}
  </div>
```

On the ingredient quantity we can call a new fraction then convert it to a string as per the guidance notes. However some of the ingredients such as herbs have a null value so we can use a ternary condition to return fraction or empty string.



Refactoring the html markup: Ingredients

To tidy up the code the ingredients part of the recipe is put into its own private method and we pass into it the ingredients array.

```
#generateMarkupIngredient(ing) {  
  return `  
    <li class="recipe__ingredient">  
      <svg class="recipe__icon">  
        <use href="${icons}#icon-check"></use>  
      </svg>  
      <div class="recipe__quantity">${  
        ing.quantity ? new Fraction(ing.quantity).toString() : ''  
      }</div>  
      <div class="recipe__description">  
        <span class="recipe__unit">${ing.unit}</span>  
        ${ing.description}  
      </div>  
    </li>  
  `;  
}
```

In the section of html markup for the recipe ingredients list we can now map that private function then chain the join method without any scruffy html markup code inbetween.

```
<ul class="recipe__ingredient-list">  
  ${this.#data.ingredients.map(this.#generateMarkupIngredient).join('')}  
</ul>
```

Helpers and configuration files

The config file is where we can put common variables that we use across our JavaScript project rather than having them in multiple locations across multiple files. If the variable needs to be changed in the future then we only need to change it in one place. We just simply export the variables.

```
// config.js
export const API_URL = 'https://forkify-api.herokuapp.com/api/v2/recipes';
export const TIMEOUT_SEC = 10;
```

In the model.js file where we need this API_URL variable we can now just import it

```
// model.js - API/state handling
import { API_URL } from './config.js';
```

```
export const loadRecipe = async function (id) {
  try {
    const res = await fetch(`${API_URL}/${id}`);
    const data = await res.json();
    if (!res.ok) throw new Error(`${data.message} (${res.status})`);
    const { recipe } = data.data;
    state.recipe = {
      id: recipe.id,
      title: recipe.title,
      publisher: recipe.publisher,
```

And use the imported variable
in our code where we need it.



```
// helpers.js
import { TIMEOUT_SEC } from './config.js';
```

Helper.js file is a place to store functions that we use over and over again in our project.

```
const timeout = function (s) {
  return new Promise(function (_, reject) {
    setTimeout(function () {
      reject(new Error(`Request took too long! Timeout after ${s} second`));
    }, s * 1000);
  });
};
```

An example would be to put the fetch api in the helper.js file. We can also modify promise to race between a timeout function with that uses a TIMEOUT_SEC variable stored in the config.js file.



```
export const getJSON = async function (url) {
  try {
    const fetchPro = fetch(url);
    const res = await Promise.race([fetchPro, timeout(TIMEOUT_SEC)]);
    const data = await res.json();

    if (!res.ok) throw new Error(`${data.message} (${res.status})`);
    return data;
  } catch (err) {
    throw err;
  }
};
```

Note that we are rethrowing the error so that it propagates in the chain should the fetch api fail or take longer than TIMEOUT_SEC to complete.




```
// model.js - API/state handling
import { async } from 'regenerator-runtime';
import { API_URL } from './config.js';
import { getJSON } from './helpers.js';
```

Now we can import the getJSON function that performs the API fetch.

```
export const state = { recipe: {} };
```

```
export const loadRecipe = async function (id) {
  try {
```

```
    const data = await getJSON(`${API_URL}/${id}`);
```

And use it in our model.js file

```
    const { recipe } = data.data;
    state.recipe = {
      id: recipe.id,
      title: recipe.title,
      publisher: recipe.publisher,
      sourceUrl: recipe.source_url,
      image: recipe.image_url,
      servings: recipe.servings,
      cookingTime: recipe.cooking_time,
      ingredients: recipe.ingredients,
```

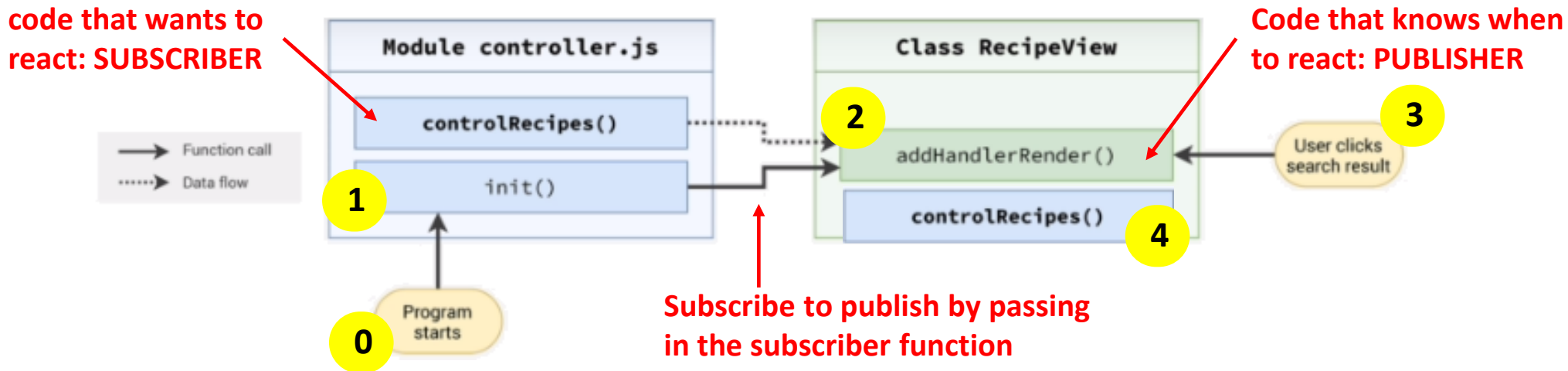
```
    };
    // console.log(state.recipe);
  } catch (err) {
```

```
    console.error(` ❌❌ ${err} ❌❌ `);
  }
}
```

```
};
```

Event handlers in MVC: Publisher-Subscriber Design Pattern

- Events should be **handled** in the **controller** (otherwise we would have application logic in the view)
- Events should be **listened for** in the **view** (otherwise we would have DOM elements in the controller)



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

Event handlers in MVC: Implementation

In the class `RecipeView` of the `recipeView.js` file we add a method called `addHandlerRender` which takes `handler` as a parameter. This is a public method because it needs to talk to the controller which is outside of the class.

```
addHandlerRender(handler) {  
  ['hashchange', 'load'].forEach(ev => window.addEventListener(ev, handler));  
}
```

From the `controller.js` file we have an `init` function that calls the `addHandlerRender` function and passes in the `controlRecipes` function as the handler. We immediately call the `init` function.

```
const init = function () {  
  recipeView.addHandlerRender(controlRecipes);  
};  
  
init();
```

MVC: Implementing Error and Success Messages

```
const data = await getJSON(`${API_URL}/${id}zz`);
```

← I have added zz to the fetch API URL to provoke an error

```
✖ ▶ GET https://forkify-api.herokuapp.com/api/v2/recipes/5ed6604...zz 400 (Bad Request)
✖ ▶ ❌❌ Error: Invalid _id: 5ed6604591c37cdc054bcd09zz. (400) ❌❌
```

But these errors are being logged to the console from the model when in the real world they should output a message to the user from the view.

```
renderError(message) {
  const markup = `
    <div class="error">
      <div>
        <svg>
          <use href="${icons}#icon-alert-triangle"></use>
        </svg>
      </div>
      <p>${message}</p>
    </div>
  `;
  this.#clear;
  this.#parentElement.insertAdjacentHTML('afterbegin', markup);
}
```

In the class RecipeView of the recipeView.js file we add a method called renderError which takes message as a parameter. This is a public method because it needs to talk to the controller which is outside of the class.

```
export const loadRecipe = async function (id) {
  try {
    const data = await getJSON(`${API_URL}/${id}zz`);

    const { recipe } = data.data;
    state.recipe = {
      id: recipe.id,
      title: recipe.title,
      publisher: recipe.publisher,
      sourceUrl: recipe.source_url,
      image: recipe.image_url,
      servings: recipe.servings,
      cookingTime: recipe.cooking_time,
      ingredients: recipe.ingredients,
    };
    // console.log(state.recipe);
  } catch (err) {
    // console.error(`×× ${err} ××`);
    throw err;
  }
};
```

As the error is coming from the
model.js LoadRecipe function we need
to re-throw the error in the catch block

```

const controlRecipes = async function () {
  try {
    const id = window.location.hash.slice(1);
    // console.log(id);

    if (!id) return;
    recipeView.renderSpinner();

    // 1. Loading Recipe
    await model.loadRecipe(id);

    // 2. Rendering recipe
    recipeView.render(model.state.recipe);
  } catch (err) {
    recipeView.renderError(`${err}`); ←
  }
};

```

From the controller.js we can call the `renderError` method in the catch which will now output the error via the view to the user.



Error: Invalid _id:
5ed6604591c37cdc054bcd09zz.
(400)

```

class RecipeView {
  #parentElement = document.querySelector('.recipe');
  #data;
  #errorMessage = 'We could not find that Recipe. Please try another one!';

```

Better still, we can define a default private `#errorMessage` in the `RecipeView` class

Forkify App: Building a Modern Application

Part 3 – Implementing search & pagination

Implementing Search Results

```
export const loadSearchResults = async function (query) {
  try {
    const data = await getJSON(`${API_URL}?search=${query}`);
    console.log(data);
  } catch (err) {
    console.error(`❌❌ ${err} ❌❌`);
    throw err;
  }
};

loadSearchResults('pizza');
```

From the API documentation at <https://forkify-api.herokuapp.com/v2> we can find the correct URL for a search and build an async function in the model.js file to handle it.

From the console log results of data we see that the API call returns an array full of recipe objects.

```
▼ {status: 'success', results: 59, data: {...}} ⓘ
  ▼ data:
    ▼ recipes: Array(59)
      ► 0: {publisher: 'Closet Cooking', image_url: 'http://forkify-api.herokuapp.com/images/BBQChickenPizzawithCauliflowerCrust5004699695624ce.jpg', title: 'Cauliflower Pizza'}
      ► 1: {publisher: 'Closet Cooking', image_url: 'http://forkify-api.herokuapp.com/images/BBQChickenPizzawithCauliflowerCrust5004699695624ce.jpg', title: 'Cauliflower Pizza'}
      ► 2: {publisher: 'Simply Recipes', image_url: 'http://forkify-api.herokuapp.com/images/pizza292x2007a259a79.jpg', title: 'Homemade Pizza', id: '5ed6604591c37cdc054bcb34'}
      ► 3: {publisher: 'Simply Recipes', image_url: 'http://forkify-api.herokuapp.com/images/howtogrillpizzad300x20086a60e1b.jpg', title: 'How to Grill Pizza', id: '5ed6604591c37cdc054bcb34'}
      ► 4: {publisher: 'Closet Cooking', image_url: 'http://forkify-api.herokuapp.com/images/Pizza28Dip2812B500c4c0a26c.jpg', title: 'Pizza Dip', id: '5ed6604591c37cdc054bcb34'}
      ► 5: {publisher: 'All Recipes', image_url: 'http://forkify-api.herokuapp.com/images/391236ba85.jpg', title: 'Veggie Pizza', id: '5ed6604591c37cdc054bca5d'}
      ► 6: {publisher: 'All Recipes', image_url: 'http://forkify-api.herokuapp.com/images/7988559586.jpg', title: 'Valentine Pizza', id: '5ed6604591c37cdc054bca57'}
      ► 7: {publisher: 'A Spicy Perspective', image_url: 'http://forkify-api.herokuapp.com/images/IMG_4351180x1804f4a.jpg', title: 'Greek Pizza', id: '5ed6604591c37cdc054bca3b'}
      ► 8: {publisher: 'My Baking Addiction', image_url: 'http://forkify-api.herokuapp.com/images/PizzaDip21of14f05.jpg', title: 'Pizza Dip', id: '5ed6604591c37cdc054bca10'}
```

```
export const state = {
  recipe: {},
  search: {
    query: '',
    results: [],
  },
};
```

We want to store the results of the search in the state so we can create an object within the state called search then have two nested objects of query which is an empty string and results which is an empty array.

```
export const loadSearchResults = async function (query) {
  try {
    state.search.query = query;

    const data = await getJSON(`${API_URL}?search=${query}`);

    state.search.results = data.data.recipes.map(rec => {
      return {
        id: rec.id,
        title: rec.title,
        publisher: rec.publisher,
        image: rec.image_url,
      };
    });
    console.log(state.search.results);
  } catch (err) {
    console.error(` ❌❌ ${err} ❌❌`);
    throw err;
  }
};

loadSearchResults('pizza');
```

We use map method to fill the search object in the state.

```
(59) [{"id": "5ed6604591c37cdc054bcd09", "title": "Cauliflower Pizza Crust (with BBQ Chicken Pizza)", "publisher": "Closet Cooking", "image": "http://forkify-api.herokuapp.com/images/5ed6604591c37cdc054bcc13", "title": "Cauliflower Pizza Crust (with BBQ Chicken Pizza)", "publisher": "Closet Cooking", "image": "http://forkify-api.herokuapp.com/images/5ed6604591c37cdc054bcb34", "title": "Homemade Pizza", "publisher": "Simply Recipes", "image": "http://forkify-api.herokuapp.com/images/pizza292x2007a259a79.jpg"}, {"id": "5ed6604591c37cdc054bcb37", "title": "How to Grill Pizza", "publisher": "Simply Recipes", "image": "http://forkify-api.herokuapp.com/images/howtogrillpizzad300x20086a60e"}, {"id": "5ed6604591c37cdc054bcac4", "title": "Pizza Dip", "publisher": "Closet Cooking", "image": "http://forkify-api.herokuapp.com/images/Pizza28Dip2812B500c4c0a26c.jpg"}, {"id": "5ed6604591c37cdc054bca5d", "title": "Veggie Pizza", "publisher": "All Recipes", "image": "http://forkify-api.herokuapp.com/images/391236ba85.jpg"}, {"id": "5ed6604591c37cdc054bca57", "title": "Valentine Pizza", "publisher": "All Recipes", "image": "http://forkify-api.herokuapp.com/images/7988559586.jpg"}, {"id": "5ed6604591c37cdc054bca3b", "title": "Greek Pizza", "publisher": "A Spicy Perspective", "image": "http://forkify-api.herokuapp.com/images/IMG_4351180x1804f4a.jpg"}, {"id": "5ed6604591c37cdc054bca10", "title": "Pizza Din", "publisher": "My Baking Addiction", "image": "http://forkify-api.herokuapp.com/images/PizzaDin2016f1405.jpg"}]
```

```
const controlSearchResults = async function () {
  try {
    await model.loadSearchResults('pizza');
    console.log(model.state.search.results);
  } catch (err) {
    console.log(err);
  }
};

controlSearchResults();
```

```
(S9) [{"_id": "5ed6604591c37cdc054bcd09", "title": "Cauliflower Pizza Crust (with BBQ Chicken Pizza)", "publisher": "Closet Cooking", "image": "http://forkify-api.herokuapp.com/images/Pizza28Dip2B12B500c4c0a26c.jpg"}, {"_id": "5ed6604591c37cdc054bcc13", "title": "Cauliflower Pizza Crust (with BBQ Chicken Pizza)", "publisher": "Closet Cooking", "image": "http://forkify-api.herokuapp.com/images/Pizza28Dip2B12B500c4c0a26c.jpg"}, {"_id": "5ed6604591c37cdc054bcb34", "title": "Homemade Pizza", "publisher": "Simply Recipes", "image": "http://forkify-api.herokuapp.com/images/pizza29x2x007a259a79.jpg"}, {"_id": "5ed6604591c37cdc054bcb37", "title": "How to Grill Pizza", "publisher": "Simply Recipes", "image": "http://forkify-api.herokuapp.com/images/howtogrillpizzad300x20086a60e.jpg"}, {"_id": "5ed6604591c37cdc054bcac4", "title": "Pizza Dip", "publisher": "Closet Cooking", "image": "http://forkify-api.herokuapp.com/images/Pizza28Dip2B12B500c4c0a26c.jpg"}, {"_id": "5ed6604591c37cdc054bca5d", "title": "Veggie Pizza", "publisher": "All Recipes", "image": "http://forkify-api.herokuapp.com/images/391236ba85.jpg"}, {"_id": "5ed6604591c37cdc054bca57", "title": "Valentine Pizza", "publisher": "All Recipes", "image": "http://forkify-api.herokuapp.com/images/7988559586.jpg"}, {"_id": "5ed6604591c37cdc054bca3b", "title": "Greek Pizza", "publisher": "A Spicy Perspective", "image": "http://forkify-api.herokuapp.com/images/IMG_4351180x1804f4a.jpg"}, {"_id": "5ed6604591c37cdc054bca10", "title": "Pizza Dip", "publisher": "My Baking Addiction", "image": "http://forkify-api.herokuapp.com/images/Pizza210f14f05.jpg"}]
```

`// SearchView.js` ← In the views folder we create a new file called searchView.js

```
class SearchView {  
  #parentEl = document.querySelector('.search');
```

This is a class to handle everything to do with searching with a private property set to the search form html element.

```
  getQuery() {  
    const query =  
this.#parentEl.querySelector('.search__field').value;  
    this.#clearInput();  
    return query;  
  }
```

The getQuery method returns the search__field value and clears the input.

```
  #clearInput() {  
    this.#parentEl.querySelector('.search__field').value = '';  
  }
```

```
  addHandlerSearch(handler) {  
    this.#parentEl.addEventListener('submit', function (e) {  
      e.preventDefault();  
      handler();  
    });  
  }  
}
```

we use the Publisher-Subscriber pattern to handle a submit event of the search form disabling the default behaviour of page reload on form submit.

```
// Controller.js - Handles communication between model and view
```

```
import * as model from './model.js';  
import recipeView from './views/recipeView.js';  
import searchView from './views/searchView.js';  
import 'core-js/stable';
```

In the controller the
searchView is imported

```
...
```

```
const controlSearchResults = async function () {
```

```
  try {
```

```
    const query = searchView.getQuery();
```

```
    if (!query) return;
```

```
    await model.loadSearchResults(query);
```

```
    console.log(model.state.search.results);
```

```
  } catch (err) {
```

```
    console.log(err);
```

```
  }
```

```
};
```

```
controlSearchResults();
```

```
const init = function () {
```

```
  recipeView.addHandlerRender(controlRecipes);
```

```
  searchView.addHandlerSearch(controlSearchResults);
```

```
};
```

```
init();
```

Now we can define a query variable and set it to the getQuery method of searchView and pass this variable into the loadSearchResults function.

And add the searchhandler
to the init function

```
// view.js
import icons from 'url:../../img/icons.svg';
```

```
export default class view {
  _data;
```

All the common methods from the recipeViews.js and searchResults.js classes can be refactored into a parent class called view. We default export this class.

```
render(data) {
  if (!data || (Array.isArray(data) && data.length === 0)) return this.renderError();

  this._data = data;
  const markup = this._generateMarkup();
  this._clear();
  this._parentElement.insertAdjacentHTML('afterbegin', markup);
}
```

```
_clear() {
  this._parentElement.innerHTML = '';
}
```

Before rendering the data we can put in a guard clause that checks if data is present or not an empty array. If it is then we can renderError message.

```
renderSpinner = function () {
  const markup = `
    <div class="spinner">
      <svg>
        <use href="${icons}#icon-loader"></use>
      </svg>
    </div>
  `;
  this._clear();
  this._parentElement.insertAdjacentHTML('afterbegin', markup);
};
```

```

renderError(message = this._errorMessage) {
  const markup = `
    <div class="error">
      <div>
        <svg>
          <use href="${icons}#icon-alert-triangle"></use>
        </svg>
      </div>
      <p>${message}</p>
    </div>
  `;
  this._clear();
  this._parentElement.insertAdjacentHTML('afterbegin', markup);
}

```

All other common methods are cut from recipeViews.js and put in the parent class of View.

```

renderMessage(message = this._message) {
  const markup = `
    <div class="message">
      <div>
        <svg>
          <use href="${icons}#icon-smile"></use>
        </svg>
      </div>
      <p>${message}</p>
    </div>
  `;
  this._clear();
  this._parentElement.insertAdjacentHTML('afterbegin', markup);
}

```

```
// recipeView.js - the class for recipes
```

```
import View from './view.js';
```

← View is imported into recipeView.js

```
import icons from 'url:../../img/icons.svg';
```

```
import { Fraction } from 'fractional';
```

```
// console.log(Fraction);
```

RecipeView is now a child class of view

```
class RecipeView extends View {
```

← so we need the extends keyword.

```
  _parentElement = document.querySelector('.recipe');
```

```
  _errorMessage = 'We could not find that Recipe. Please try  
another one!';
```

```
  _message = '';
```

```
  addHandlerRender(handler) {
```

```
    ['hashchange', 'load'].forEach(ev =>
```

```
    window.addEventListener(ev, handler));
```

```
  }
```

```
  _generateMarkup() {
```

```
    return `
```

```
      <figure class="recipe__fig">
```

```
        <img src="${this._data.image}" alt="${
```



```
// resultsView.js
import View from './view.js';
import icons from 'url:../../img/icons.svg';
```

View is imported into resultsView.js

```
class ResultsView extends View {
  _parentElement = document.querySelector('.results');
  _errorMessage = 'No recipies found for your query. Please try again!';
  _message = '';
```

ResultsView is a child class of view so we need the extends keyword.

```
  _generateMarkup() {
    return this._data.map(this._generateMarkupPreview).join('');
  }
```

```
  _generateMarkupPreview(result) {
    return `
    <li class="preview">
      <a class="preview__link" href="#${result.id}>
        <figure class="preview__fig">
          
        </figure>
        <div class="preview__data">
          <h4 class="preview__title">${result.title}</h4>
          <p class="preview__publisher">${result.publisher}</p>
        </div>
      </a>
    </li>
    `;
  }
}
```

As data for the resultsView is a narray we need to loop over it using the .map method and join all the iterations together.

```
export default new ResultsView();
```

Implementing Pagination – slice searchResults array

```
// config.js
```

```
export const API_URL = 'https://forkify-  
api.herokuapp.com/api/v2/recipes';
```

```
export const TIMEOUT_SEC = 10;
```

```
export const RES_PER_PAGE = 10;
```

We only want to render 10 results pr page so we can export this variable from the config.js file

```
// model.js - API/state handling
```

```
import { async } from 'regenerator-runtime';
```

```
import { API_URL, RES_PER_PAGE } from './config.js';
```

```
import { getJSON } from './helpers.js';
```

And import the variable in the model.

```
export const state = {
```

```
  recipe: {},
```

```
  search: {
```

```
    query: '',
```

```
    results: [],
```

```
    page: 1,
```

```
    resultsPerPage: RES_PER_PAGE,
```

In the state we want a default page of 1

And the results per page to be set to the imported variable from config.js.

```
  },
```

```
};
```

In the model.js file we need a function that takes page as a parameter or uses the default set in the state.



```
export const getSearchResultsPage = function (page = state.search.page) {  
  state.search.page = page;  
  
  const start = (page - 1) * state.search.resultsPerPage; // 0  
  const end = page * state.search.resultsPerPage; // 9  
  
  return state.search.results.slice(start, end);  
};
```

as an array is zero based numbering if we pass in page 1 then we need to subtract 1 to get zero then multiply it by the number of results per page. this gives us our starting element in the array.

To get the end value of the slice we can multiply page by number of results per page.

i.e. passing page 1::: Start: $(1 - 1) * 10 = 0$, End: $1 * 10 = 10$.

i.e. passing page 2::: Start: $(2 - 1) * 10 = 10$, End: $2 * 10 = 20$.

i.e. passing page 3::: Start: $(3 - 1) * 10 = 20$, End: $3 * 10 = 30$.

Now we can return the slice of the state search results depending on what page we pass in.

```
const controlSearchResults = async function () {  
  try {  
    resultsView.renderSpinner();  
  
    // 1. Get search query  
    const query = searchView.getQuery();  
    if (!query) return;  
  
    // 2. load search results  
    await model.loadSearchResults(query);  
  
    // 3. Render results  
    resultsView.render(model.getSearchResultsPage(1));  
    // console.log(model.state.search.results);  
  } catch (err) {  
    console.log(err);  
  }  
};  
controlSearchResults();
```

In the controller we want to render only the 10 results that are sliced in the `getSearchResultsPage` function. We can pass in a page number, i.e. 1, 2, 3 etc.. Later we will configure buttons for this! 

Implementing Pagination – page navigation buttons

There are a number of scenarios to cater for when implementing search results pagination buttons:

1. we are on page 1 - next to page 2 only button.
2. we are on the last page - previous only button.
3. we are on a page from 2 till penultimate - next and previous page buttons.
4. There are 10 or less results - no pagination buttons.

```
// paginationView.js
import View from './view.js';
import icons from 'url:../img/icons.svg';

class PaginationView extends View {
  _parentElement = document.querySelector('.pagination');
}

export default new PaginationView();
```

← We create a paginationView.js file in the view folder.

← And import the parent class

← And the icons

← The PaginationView class is a child class of View.

← As before we return a default new PaginationView

```
_generateMarkup() { ← We have a generate markup function to make the pagination buttons.  
  const currentPage = this._data.page;  
  const numPages = Math.ceil(this._data.results.length / this._data.resultsPerPage);
```

As this is in a child class it inherits the `_data` from the parent class which includes the state object. Within the state object we have a property called `page` which is set to the current page.

```
// 1. page 1 and there are other pages  
if (currentPage === 1 && numPages > 1)  
  return `  
    <button data-goto="${currentPage + 1}" class="btn--inline pagination__btn--next">  
      <span>Page ${currentPage + 1}</span>  
      <svg class="search__icon">  
        <use href="${icons}#icon-arrow-right"></use>  
      </svg>  
    </button>  
  `;  
// 2. last page  
if (currentPage === numPages && numPages > 1)  
  return `  
    <button data-goto="${currentPage - 1}" class="btn--inline pagination__btn--prev">  
      <svg class="search__icon">  
        <use href="${icons}#icon-arrow-left"></use>  
      </svg>  
      <span>page ${currentPage - 1}</span>  
    </button>  
  `;
```

Number of pages is the `_data.results` length. From this we can calculate the number of pages needed to display the results and round up.

Now we can write an if block for each of the four possible button scenarios.

```

- // 3. other page
if (currentPage < numPages)
    return `
<button data-goto="${currentPage + 1}" class="btn--inline pagination__btn--next">
  <span>Page ${currentPage + 1}</span>
  <svg class="search__icon">
    <use href="${icons}#icon-arrow-right"></use>
  </svg>
</button>
<button data-goto="${currentPage - 1}" class="btn--inline pagination__btn--prev">
  <svg class="search__icon">
    <use href="${icons}#icon-arrow-left"></use>
  </svg>
  <span>page ${currentPage - 1}</span>
</button>
`
;
// 4. page 1, and there are NO other pages
return '';
}

```

Note that the buttons have a dataset parameter set the page we want to go to.

If there is only one page then we do not need pagination buttons so return an empty string.

```
addHandlerClick(handler) {  
  this._parentElement.addEventListener('click', function (e)  
  {  
    const btn = e.target.closest('.btn--inline');  
    if (!btn) return;  
  
    const goToPage = +btn.dataset.goto;  
    // console.log(goToPage);  
    handler(goToPage);  
  });  
}
```

Within the paginationView class we want an event handler attached to the parent element of the buttons.

Using event delegation we can search for the closest element with btn—inline class to the parent. This means we only attach event listener to the buttons and not any other element of the parent such as span or icon.

In the event listener we can store the go to page dataset in a variable and pass it into the handler().

```
// Controller.js - Handles communication between model and view  
import * as model from './model.js';  
import recipeView from './views/recipeView.js';  
import searchView from './views/searchView.js';  
import resultsView from './views/resultsView.js';  
import paginationView from './views/paginationView.js';
```

Import paginationView into the controller


```
const controlSearchResults = async function () {  
  try {  
    resultsView.renderSpinner();  
  
    // 1. Get search query  
    const query = searchView.getQuery();  
    if (!query) return;  
  
    // 2. load search results  
    await model.loadSearchResults(query);  
  
    // 3. Render results  
    resultsView.render(model.getSearchResultsPage());  
    // console.log(model.state.search.results);  
  
    // 4. Render initial pagination buttons  
    paginationView.render(model.state.search);  
  } catch (err) {  
    console.log(err);  
  }  
};  
controlSearchResults();
```

 In the controller we render initial pagination buttons.

```
const controlPagination = function (goToPage) {  
  // 3. Render NEW results  
  resultsView.render(model.getSearchResultsPage(goToPage));  
  // console.log(model.state.search.results);  
  
  // 4. Render NEW pagination buttons  
  paginationView.render(model.state.search);  
};
```

← In the controller the control navigation renders new buttons once the buttons are clicked tsking go to page as a parameter.

```
const init = function () {  
  recipeView.addHandlerRender(controlRecipes);  
  searchView.addHandlerSearch(controlSearchResults);  
  paginationView.addHandlerClick(controlPagination);  
};  
init();
```

The event listener for the pagination buttons is also added to the init function. ←

Forkify App: Building a Modern Application

Part 4 – Change Servings &
Develop an update DOM algorithm

Implementing Update Servings

```
<div class="recipe__info-buttons">
  <button class="btn--tiny btn--update-servings" data-update-to="${ this._data.servings - 1}">
    <svg>
      <use href="${icons}#icon-minus-circle"></use>
    </svg>
  </button>
  <button class="btn--tiny btn--update-servings" data-update-to="${this._data.servings + 1}">
    <svg>
      <use href="${icons}#icon-plus-circle"></use>
    </svg>
  </button>
</div>
</div>
```

In the `recipeView.js` file for the `_generateMarkup()` method, there are two html buttons for plus and minus. We can add a dataset to these to pass a servings +1 or -1 value.

```
addHandlerUpdateServings(handler) {
  this._parentElement.addEventListener('click', function (e) {
    const btn = e.target.closest('.btn--update-servings');
    if (!btn) return;
    const { updateTo } = btn.dataset;
    if (+updateTo > 0) handler(+updateTo);
  });
}
```

Now we attach an event handler to the buttons using `closest`. If one of the buttons is pressed then we pass in the new servings value into the handler in the controller. There is a guard clause to not allow a serving value of less than 1.

```
const controlServings = function (newServings) {  
  // Update the recipe servings in the state  
  model.updateServings(newServings);  
  // Update the recipe view  
  recipeView.render(model.state.recipe);  
};
```

← The controlServings function takes the newServings parameter passed in from the view and updates the state then re-renders the recipe.

```
const init = function () {  
  recipeView.addHandlerRender(controlRecipes);  
  recipeView.addHandlerUpdateServings(controlServings);  
  searchView.addHandlerSearch(controlSearchResults);  
  paginationView.addHandlerClick(controlPagination);  
};  
init();
```

← Upon initialisation we attach the event handler to the update servings buttons which is set to call controlUpdateServings

Develop an update DOM algorithm

Currently when the servings are updated the code renders the whole page again. This means that the image is reloaded as well as all the text elements when all that needs to change are the servings numbers. This is unnecessary load on the browser and means that the image flickers as it reloads each time. We can solve this with a DOM update algorithm.

```
const controlServings = function (newServings) {  
  // Update the recipe servings in the state  
  model.updateServings(newServings);  
  // Update the recipe view  
  // recipeView.render(model.state.recipe);  
  recipeView.update(model.state.recipe);  
};
```

In the controller.js file for the update servings section we can remove the recipeView render and replace it with recipeView update which we will write now.

```
update(data) {  
  if (!data || (Array.isArray(data) && data.length === 0))  
    return this.renderError();  
  
  this._data = data;  
  const newMarkup = this._generateMarkup();  
  
  const newDOM = document.createRange().createContextualFragment(newMarkup);  
  const newElements = newDOM.querySelectorAll('*');  
  console.log(newElements);  
}
```

← We can write a generic DOM update method in the view.js parent class.

← We have a check to ensure that the data passed in is not empty or an empty array.

← we get the new page markup and assign it to variable newMarkup

The Range interface represents a fragment of a document that can contain nodes and parts of text nodes. A range can be created by using the Document.createRange() method.

<https://developer.mozilla.org/en-US/docs/Web/API/Range>

The Range.createContextualFragment() method returns a DocumentFragment by invoking the HTML fragment parsing algorithm or the XML fragment parsing algorithm with the start of the range (the parent of the selected node) as the context node.

<https://developer.mozilla.org/en-US/docs/Web/API/Range/createContextualFragment>

What we are doing is creating a copy of the new DOM in the memory. Then selecting all the html elements with a querySelectAll.

The log of shows a nodeList that is the contents of all the html elements on our page.

[view.js:26](#)

```
NodeList(61) [figure.recipe_fig, img.recipe_img, h1.recipe_title, span,
div.recipe_details, div.recipe_info, svg.recipe_info-icon, use,
span.recipe_info-data.recipe_info-data--minutes, span.recipe_info-text,
div.recipe_info, svg.recipe_info-icon, use,
span.recipe_info-data.recipe_info-data--people, span.recipe_info-text,
div.recipe_info-buttons, button.btn--tiny.btn--update-servings, svg, use,
button.btn--tiny.btn--update-servings, svg, use, div.recipe_user-generated,
button.btn--round, svg, use, div.recipe_ingredients, h2.heading--2,
ul.recipe_ingredient-list, li.recipe_ingredient, svg.recipe_icon, use,
div.recipe_quantity, div.recipe_description, span.recipe_unit, li.recipe_ingredient,
svg.recipe_icon, use, div.recipe_quantity, div.recipe_description, span.recipe_unit,
li.recipe_ingredient, svg.recipe_icon, use, div.recipe_quantity,
div.recipe_description, span.recipe_unit, li.recipe_ingredient, svg.recipe_icon, use,
div.recipe_quantity, div.recipe_description, span.recipe_unit, div.recipe_directions,
h2.heading--2, p.recipe_directions-text, span.recipe_publisher,
a.btn--small.recipe_btn, span, svg.search_icon, use]
```

What we have here is a “virtual DOM” of the html elements on the page after we click a button and values change, but it is not the actual DOM on the page which contains the html elements with the old values.

What we can do is compare this “virtual DOM with the actual DOM on the page to see what has changed.


```
update(data) {
  if (!data || (Array.isArray(data) && data.length === 0))
    return this.renderError();
```

```
this._data = data;
const newMarkup = this._generateMarkup();
```

```
const newDOM = document.createRange().createContextualFragment(newMarkup);
const newElements = Array.from(newDOM.querySelectorAll('*'));
const curElements = Array.from(this._parentElement.querySelectorAll('*'));
```

```
newElements.forEach((newEl, i) => {
  const curEl = curElements[i];
  console.log(curEl, newEl.isEqualNode(curEl));
});
```

We can get the current elements on the page by query selecting all the elements from the parent element.

Note that we want the current elements and the new elements both as an array using the method `Array.from()`

Now the magic starts to happen. We can loop over the new elements and compare them with the current elements to see if it is `isEqualNode`. This will return a true or false if there is a difference.

Note 1. We are now not comparing ALL the DOM elements on the page but comparing only `_parentElement`, i.e. everything in the servings section.

Note 2. if a child element changes then this change will also be reflected in the parent element of that child.

```
<div class="recipe_info"> flex
  <svg class="recipe_info-icon">...</svg>
  <span class="recipe_info-data recipe_info-data--people">4</span>
  <span class="recipe_info-text">servings</span>
  <div class="recipe_info-buttons">...</div> flex
</div>
```

parent

child

false

This recipe only has 4 ingredients so the list of current elements is shorter. We can see the unchanged elements marked as true and the changed elements marked as false.

forkify // Search over 1,000,000 recipes

localhost:1234/#5ed6604591c37cdc054bc971

Dimensions: iPad pro... 1366 x 1024 50%

Elements Console Sources Network Performance Memory Application Security Lighthouse

Filter

Default levels 1 Issue: 1

view.js:30

```
<figure class="recipe_fig">...</figure> true
 true
<h1 class="recipe_title">...</h1> true
<span>Pizza Pinwheels</span> true
<div class="recipe_details">...</div> flex false
<div class="recipe_info">...</div> flex true
<svg class="recipe_info-icon">...</svg> true
<use href="http://localhost:1234/icons.dfd7a6db.svg?1654440770826#icon-clock"></use> true
<span class="recipe_info-data recipe_info-data--minutes">45</span> true
<span class="recipe_info-text">minutes</span> true
<div class="recipe_info">...</div> flex false
<svg class="recipe_info-icon">...</svg> true
<use href="http://localhost:1234/icons.dfd7a6db.svg?1654440770826#icon-users"></use> true
<span class="recipe_info-data recipe_info-data--people">4</span> false
<span class="recipe_info-text">servings</span> true
<div class="recipe_info-buttons">...</div> flex false
<button class="btn--tiny btn--update-servings" data-update-to="3">...</button> false
<svg>...</svg> true
<use href="http://localhost:1234/icons.dfd7a6db.svg?1654440770826#icon-minus-circle"></use> true
<button class="btn--tiny btn--update-servings" data-update-to="5">...</button> false
<svg>...</svg> true
<use href="http://localhost:1234/icons.dfd7a6db.svg?1654440770826#icon-plus-circle"></use> true
<div class="recipe_user-generated"> </div> flex true
<button class="btn--round">...</button> flex true
<svg class="...">...</svg> true
```

```

update(data) {
  if (!data || (Array.isArray(data) && data.length === 0))
    return this.renderError();

  this._data = data;
  const newMarkup = this._generateMarkup();

  const newDOM = document.createRange().createContextualFragment(newMarkup);
  const newElements = Array.from(newDOM.querySelectorAll('*'));
  const curElements = Array.from(this._parentElement.querySelectorAll('*'));

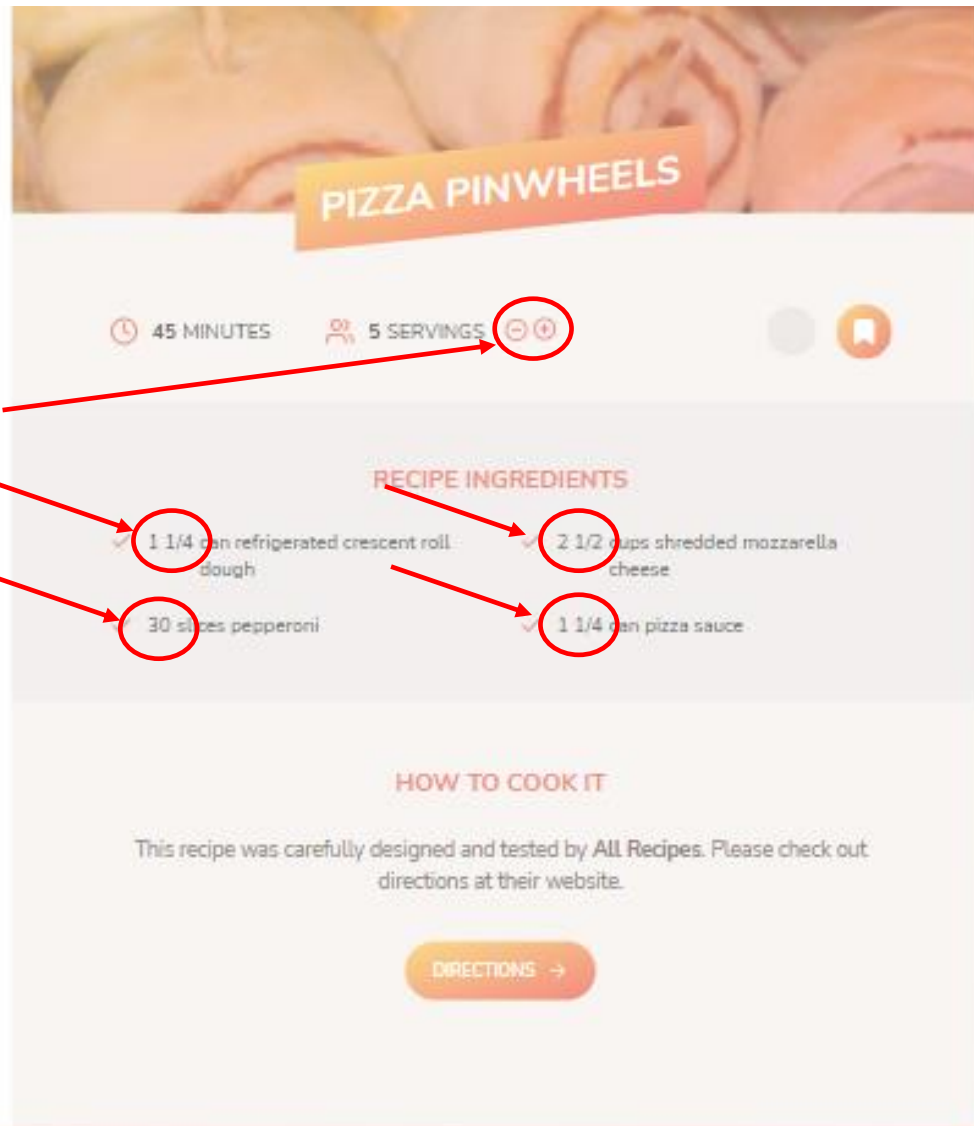
  newElements.forEach((newEl, i) => {
    const curEl = curElements[i];
    // console.log(curEl, newEl.isEqualNode(curEl));
    if (!newEl.isEqualNode(curEl) && newEl.firstChild.nodeValue.trim() !== '') {
      curEl.textContent = newEl.textContent;
    }
  });
}

```

We want to set the current element to the new element if it is not equal. However we need to specify the first child value and trim out any white spaces. This is so hat we can target the text content of child nodes only that have changed otherwise it will change the parent elements as well.

<https://developer.mozilla.org/en-US/docs/Web/API/Node/nodeValue>

Clicking on a increase or decrease servings button will now update the DOM with only the child nodes that have changed



```

update(data) {
  if (!data || (Array.isArray(data) && data.length === 0))
    return this.renderError();

  this._data = data;
  const newMarkup = this._generateMarkup();

  const newDOM = document.createRange().createContextualFragment(newMarkup);
  const newElements = Array.from(newDOM.querySelectorAll('*'));
  const curElements = Array.from(this._parentElement.querySelectorAll('*'));

  newElements.forEach((newEl, i) => {
    const curEl = curElements[i];
    console.log(curEl, newEl.isEqualNode(curEl));
    if (!newEl.isEqualNode(curEl) && newEl.firstChild?.nodeValue.trim() !== '') {
      console.log('✖✖✖✖', newEl.firstChild.nodeValue.trim());
      curEl.textContent = newEl.textContent;
    }
  });
}

```

If we log the nodes that are not equal and we log the first child nodes node values not qual and compare we can see that with `if (!newEl.isEqualNode(curEl) && newEl.firstChild.nodeValue.trim() !== '') {` we are precisely targeting what needs to be changed. If we just set it to newel not equal to curEl then it would change incorrectly the DOM. See following slide.

`if (!newEl.isEqualNode(curEl) && newEl.firstChild?.nodeValue.trim() !== '') {`
we are precisely targeting what needs to be changed. Note the optional chaining on the first child to make fully sure that it exists.

The screenshot displays a web browser window with the URL `localhost:1234/#5ed6604591c37cdc054bc971`. The page shows a recipe for "PIZZA PINWHEELS" with a search bar, a list of ingredients, and a "HOW TO COOK IT" section. The developer console is open, showing the DOM tree. The console highlights the following elements:

- `<li class="recipe_ingredient"> 2 1/2 </li>` (circled in red)
- `<div class="recipe_quantity">30</div>` (circled in red)
- `<div class="recipe_quantity">1 1/4</div>` (circled in red)

The console also shows the following HTML structure for the recipe ingredients:

```
<div class="recipe_description">...</div> true
<span class="recipe_unit"></span> true
<li class="recipe_ingredient">
  <svg class="recipe_icon">...</svg>
  <div class="recipe_quantity">2 1/2</div>
  <div class="recipe_description">...</div>
</li>
<svg class="recipe_icon">...</svg> true
<use href="http://localhost:1234/icons.dfd7a6db.svg?1654443873043#icon-check"></use> true
<div class="recipe_quantity">2 1/2</div> false
  2 1/2
  <div class="recipe_description">...</div> true
  <span class="recipe_unit">cups</span> true
  <li class="recipe_ingredient">...</li>
  <svg class="recipe_icon">...</svg> true
  <use href="http://localhost:1234/icons.dfd7a6db.svg?1654443873043#icon-check"></use> true
  <div class="recipe_quantity">30</div> false
    30
    <div class="recipe_description">...</div> true
    <span class="recipe_unit"></span> true
    <li class="recipe_ingredient">...</li>
    <svg class="recipe_icon">...</svg> true
    <use href="http://localhost:1234/icons.dfd7a6db.svg?1654443873043#icon-check"></use> true
    <div class="recipe_quantity">1 1/4</div> false
      1 1/4
      <div class="recipe_description">...</div> true
```

```
if (!newEl.isEqualNode(curEl) {
```

we are imprecisely targeting what needs to be changed so all the false html elements get overwritten and the recipe section does not render correctly.

forkify // Search over 1,000,000 r x +

localhost:1234/#5ed6604591c37cdc054bc971

Dimensions: ipad pro horizontal 1366 x 1024 50% No throttling

forkify Search over 1,000,000 recipes... SEARCH ADD RECIPE BOOKMARKS

PIZZA PINWHEELS

45 minutes 5 servings

Recipe ingredients 1 1/4 can refrigerated crescent roll dough 2 1/2 cups shredded mozzarella cheese 30 slices pepperoni 1 1/4 can pizza sauce

HOW TO COOK IT

This recipe was carefully designed and tested by All Recipes. Please check out directions at their website.

DIRECTIONS →

view.js:30

```
<figure class="recipe_fig">...</figure> true
view.js:30

true
view.js:30
<h1 class="recipe_title">...</h1> true
view.js:30
<span>Pizza Pinwheels</span> true
view.js:30
<div class="recipe_details">...</div> false
view.js:30
<div class="recipe_info">...</div> true
view.js:30
<svg class="recipe_info-icon">...</svg> true
view.js:30
<use href="http://localhost:1234/icons.dfd7a6db.svg?1654444969087#icon-clock"></use> true
view.js:30
<span class="recipe_info-data recipe_info-data--minutes">45</span> true
view.js:30
<span class="recipe_info-text">minutes</span> true
view.js:30
<div class="recipe_info">
  " 5 servings "
</div> false
view.js:30
<svg class="recipe_info-icon">...</svg> true
view.js:30
<use href="http://localhost:1234/icons.dfd7a6db.svg?1654444969087#icon-users"></use> true
view.js:30
<span class="recipe_info-data recipe_info-data--people">5</span> false
view.js:30
<span class="recipe_info-text">servings</span> true
view.js:30
<div class="recipe_info-buttons">...</div> false
view.js:30
<button class="btn--tiny btn--update-servings" data-update-to="3"> </button> false
view.js:30
<svg>...</svg> true
view.js:30
<use href="http://localhost:1234/icons.dfd7a6db.svg?1654444969087#icon-minus-circle"></use>
true
view.js:30
<button class="btn--tiny btn--update-servings" data-update-to="5"> </button> false
view.js:30
<svg>...</svg> true
view.js:30
```

However the button dataset attributes are not changes with this code. The code precisely targets text content of child nodes. This means we cannot sequentially press the increment or decrement servings because the button dataset will not change.

```
update(data) {
  if (!data || (Array.isArray(data) && data.length === 0))
    return this.renderError();

  this._data = data;
  const newMarkup = this._generateMarkup();

  const newDOM = document.createRange().createContextualFragment(newMarkup);
  const newElements = Array.from(newDOM.querySelectorAll('*'));
  const curElements = Array.from(this._parentElement.querySelectorAll('*'));

  newElements.forEach((newEl, i) => {
    const curEl = curElements[i];

    // Update changed TEXT
    if (!newEl.isEqualNode(curEl) && newEl.firstChild?.nodeValue.trim() !== '') {
      curEl.textContent = newEl.textContent;
    }

    // Update changed ATTRIBUTES
    if (!newEl.isEqualNode(curEl)) console.log(newEl.attributes);
  });
}
```

We need to look at elements
that have changed attributes

We want to specifically target the button object which has an object of attributes {class, data-update-to, length}. The idea is to target the data-update-to attribute with out code.

The screenshot shows a web browser window with the URL `localhost:1234/#5ed6604591c37cdc054bc971`. The page displays a recipe for "PIZZA PINWHEELS" with a loading spinner, a 45-minute timer, 5 servings, and a list of ingredients. The developer console is open, showing the DOM tree and the console log. The console log displays several `NamedNodeMap` objects, with the following structure for the selected object:

```
▶ NamedNodeMap {0: class, 1: data-update-to, class: class, data-update-to: data-update-to, length: 2} 1 view.js:42
  ▶ 0: class
  ▶ 1: data-update-to
  ▶ class: class
  ▶ data-update-to: data-update-to
    length: 2
  ▶ [[Prototype]]: NamedNodeMap
```

The console also shows other `NamedNodeMap` objects for the page's structure, including the header and the recipe ingredients section.

```

update(data) {
  if (!data || (Array.isArray(data) && data.length === 0))
    return this.renderError();

  this._data = data;
  const newMarkup = this._generateMarkup();

  const newDOM = document.createRange().createContextualFragment(newMarkup);
  const newElements = Array.from(newDOM.querySelectorAll('*'));
  const curElements = Array.from(this._parentElement.querySelectorAll('*'));

  newElements.forEach((newEl, i) => {
    const curEl = curElements[i];

    // Update changed TEXT
    if (!newEl.isEqualNode(curEl) && newEl.firstChild?.nodeValue.trim() !== '') {
      curEl.textContent = newEl.textContent;
    }

    // Update changed ATTRIBUTES
    if (!newEl.isEqualNode(curEl))
      Array.from(newEl.attributes).forEach(attr =>
        curEl.setAttribute(attr.name, attr.value)
      );
  });
}

```

Now we can create an array from the attributes and loop over them setting the attribute name and value to the new values.

Bug fix #1

Bug description: if we search for a recipe and use the prev and next buttons, going to a page other than first, i.e. page 3 then run a new search. The new search loads to page 3 and not page 1.

```
export const loadSearchResults = async function (query) {  
  try {  
    state.search.query = query;  
  
    const data = await getJSON(`${API_URL}?search=${query}`);  
  
    state.search.results = data.data.recipes.map(rec => {  
      return {  
        id: rec.id,  
        title: rec.title,  
        publisher: rec.publisher,  
        image: rec.image_url,  
      };  
    });  
    state.search.page = 1;  
  } catch (err) {  
    console.error(`❌❌ ${err} ❌❌`);  
    throw err;  
  }  
};
```

The reason is that in the state when we use prev and next buttons we change the page property in the state. When we load search results we need to set the page in the state back to 1.



Forkify App: Building a Modern Application

Part 5 – Implement bookmarks

Implementing Bookmarks

In the model, the state is modified to contain an array of bookmarks which is initially set to empty.

In the viewRecipie.js file `_generateMarkup()` method we can adjust the classes of the bookmark button and the icon svg.

```
<button class="btn--round button--bookmark">
  <svg class="">
    <use href="{icons}#icon-bookmark"></use>
  </svg>
</button>
```

```
addHandlerAddBookmark(handler) {
  this._parentElement.addEventListener('click', function (e) {
    const btn = e.target.closest('.btn--bookmark');
    if (!btn) return;
    handler();
  });
}
```

In the viewRecipie.js file add the handler for the bookmark.

```
export const state = {
  recipe: {},
  search: {
    query: '',
    results: [],
    page: 1,
    resultsPerPage: RES_PER_PAGE,
  },
  bookmarks: [],
};
```

```
const controlAddBookmark = function () {  
  model.addBookmark(model.state.recipe);  
};
```

In controller we add the control for the add bookmark function

```
const init = function () {  
  recipeView.addHandlerRender(controlRecipes);  
  recipeView.addHandlerUpdateServings(controlServings);  
  recipeView.addHandlerAddBookmark(controlAddBookmark);  
  searchView.addHandlerSearch(controlSearchResults);  
  paginationView.addHandlerClick(controlPagination);  
};  
init();
```

And set the event listener in the init.

```
export const addBookmark = function (recipe) {  
  // Add bookmark  
  state.bookmarks.push(recipe);  
  
  // Mark current recipe as bookmark  
  if (recipe.id === state.recipe.id) state.recipe.bookmarked = true;  
};
```

In the model we create an add bookmark function that takes a recipe as a parameter and pushes the recipe into the bookmarks array of the state.

If the recipe id is the same as the state recipe id then it will set bookmarked to true.

```
export const loadRecipe = async function (id) {  
  try {  
    const data = await getJSON(`${API_URL}/${id}`);
```

```
    const { recipe } = data.data;  
    state.recipe = {  
      id: recipe.id,  
      title: recipe.title,  
      publisher: recipe.publisher,  
      sourceUrl: recipe.source_url,  
      image: recipe.image_url,  
      servings: recipe.servings,  
      cookingTime: recipe.cooking_time,  
      ingredients: recipe.ingredients,  
    };  
  }  
};
```



```
    {  
      id: '5ed6604591c37cdc054bcd45', title: 'Spring Pasta', publisher: '101 Cookbooks', sourceUrl:  
        'http://www.101cookbooks.com/archives/spring-pasta-recipe.html', image: 'http://forkify-a  
        pi.herokuapp.com/images/spring_pasta_recipe8286.jpg', ...}  
    },  
    {  
      bookmarked: true,  
      cookingTime: 60,  
      id: "5ed6604591c37cdc054bcd45",  
      image: "http://forkify-api.herokuapp.com/images/spring_pasta_recipe8286.jpg",  
      ingredients: (8) [{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}],  
      publisher: "101 Cookbooks",  
      servings: 4,  
      sourceUrl: "http://www.101cookbooks.com/archives/spring-pasta-recipe.html",  
      title: "Spring Pasta",  
      [[Prototype]]: Object  
    }  
  ]  
}
```

```
if (state.bookmarks.some(bookmark => bookmark.id === id))  
  state.recipe.bookmarked = true;  
else state.recipe.bookmarked = false;
```

```
console.log(state.recipe);
```

Now in the model, loadRecipe method we can use the some method to Add a bookmarked property accordingly.

Multiple recipes can be set to bookmarked

The screenshot displays a web browser window with the URL `localhost:1234/#5ed6604591c37cdc054bca57`. The browser's developer tools are open, showing the Console tab with two log entries. The first log entry is from `model.js:38` and the second is from `controller.js:85`. Both log entries are JSON objects representing recipe data. In both objects, the `bookmarked` property is set to `true` and is highlighted with a red box. The web application interface shows a sidebar with a list of recipe categories, a main content area displaying the 'Valentine Pizza' recipe with its ingredients and a large image, and a bottom section for 'RECIPE INGREDIENTS'.

Recipe Data (from Console):

```
{
  id: "5ed6604591c37cdc054bca57",
  title: "Valentine Pizza",
  publisher: "All Recipes",
  sourceUrl: "http://allrecipes.com/Recipe/Valentine-Pizza/Detail.aspx",
  image: "http://forkify-api.herokuapp.com/images/7988559586.jpg",
  bookmarked: true,
  cookingTime: 75,
  ingredients: (11) [{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}],
  publisher: "All Recipes",
  servings: 4,
  sourceUrl: "http://allrecipes.com/Recipe/Valentine-Pizza/Detail.aspx",
  title: "Valentine Pizza"
}
```



```
export const deleteBookmark = function (id) {  
  // Delete bookmark  
  const index = state.bookmarks.findIndex(el => el.id === id);  
  state.bookmarks.splice(index, 1);  
  
  // Mark current recipe as NOT bookmarked  
  if (id === state.recipe.id) state.recipe.bookmarked = false;  
};
```

In the model we create a delete bookmark function which uses the `findIndex` method to find the id in the bookmarks of the recipe we want to splice out.

The `bookmarked` property for that recipe is also set to false in the state.

```
const controlAddBookmark = function () {  
  // 1) Add/remove bookmark  
  if (!model.state.recipe.bookmarked) model.addBookmark(model.state.recipe);  
  if (model.deleteBookmark(model.state.recipe.id));  
  
  console.log(model.state.recipe);  
  recipeView.update(model.state.recipe);  
};
```

Now we need to change the logic on our `addBookmark` function in the controller.

If it not bookmarked then we add a bookmark. Otherwise we delete the bookmark. However this code produces a bug because it first adds the bookmarked then in the next line it deletes the bookmarked! It needs to be an `if else`.

```
// 1) Add/remove bookmark  
if (!model.state.recipe.bookmarked) model.addBookmark(model.state.recipe);  
else model.deleteBookmark(model.state.recipe.id);
```

```
// BookmarksView.js - the class for bookmarks
import View from './view.js';
```

Create bookmarks View

```
class BookmarksView extends View {
  _parentElement = document.querySelector('.bookmarks__list');
  _errorMessage = 'No bookmarks yet. Find a nice recipe and bookmark it :)'
  _message = '';

  _generateMarkup() {
    return this._data.map(this._generateMarkupPreview).join('');
  }

  _generateMarkupPreview(result) {
    return `
    <li class="preview">
      <a class="preview__link" href="#${result.id}">
        <figure class="preview__fig">
          
        </figure>
        <div class="preview__data">
          <h4 class="preview__title">${result.title}</h4>
          <p class="preview__publisher">${result.publisher}</p>
        </div>
      </a>
    </li>
    `;
  }
}

export default new BookmarksView();
```

Bookmarksvew is almost a copy of the resultsView. The following has been changed:

- A) Class name
- B) Export default
- C) Error message
- D) Parent Element

```
// Controller.js - Handles communication between model and view
```

```
import * as model from './model.js';
```

```
import recipeView from './views/recipeView.js';
```

```
import searchView from './views/searchView.js';
```

```
import resultsView from './views/resultsView.js';
```

```
import paginationView from './views/paginationView.js';
```

```
import bookmarksView from './views/bookmarksView.js';
```

We import bookmarks view in the controller

```
...
```

```
const controlRecipes = async function () {
```

```
  try {
```

```
    const id = window.location.hash.slice(1);
```

```
    if (!id) return;
```

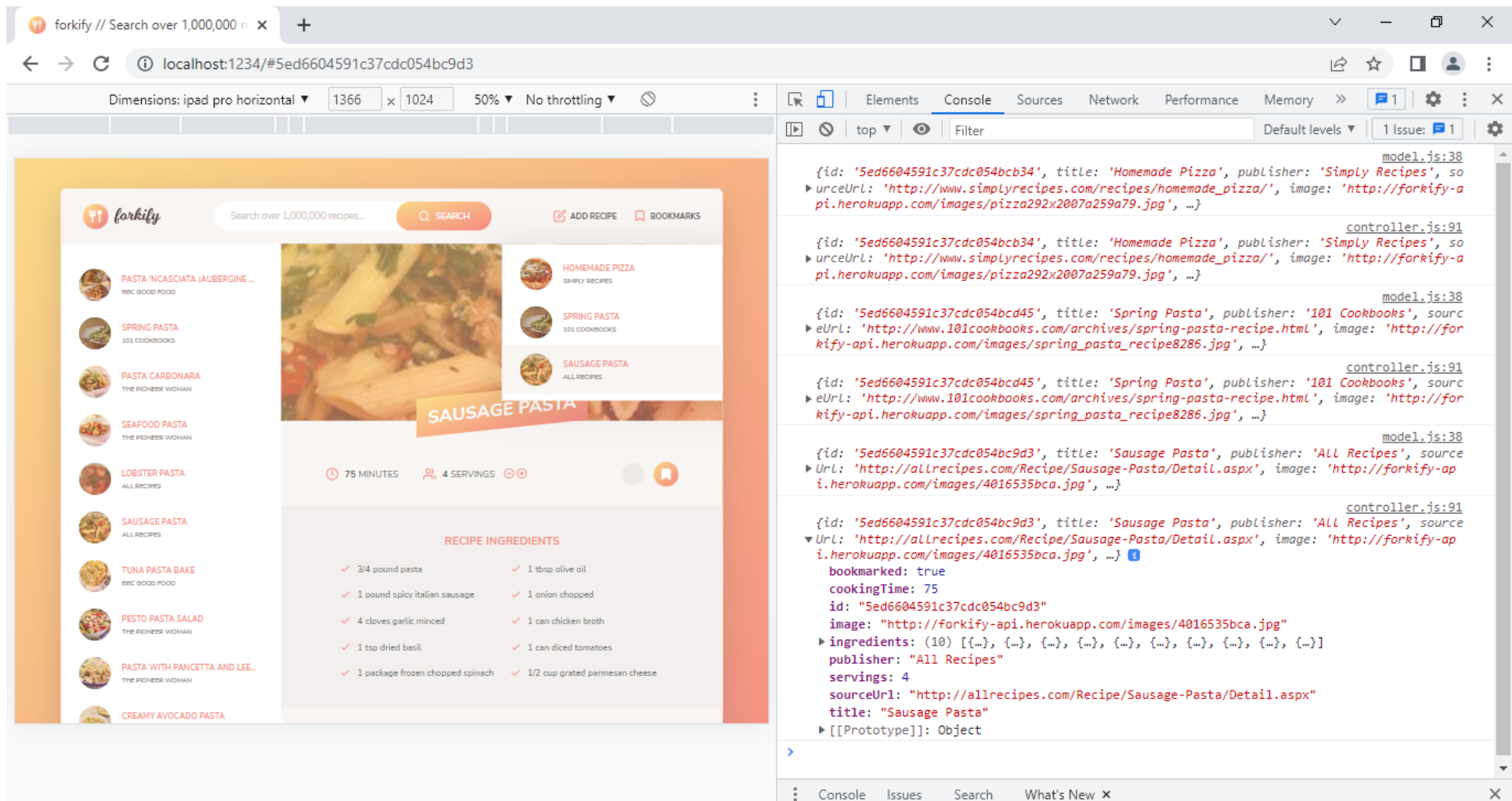
```
    recipeView.renderSpinner();
```

```
// 0) Update results view to mark selected search result
```

```
resultsView.update(model.getSearchResultsPage());
```

```
bookmarksView.update(model.state.bookmarks);
```

We add it here so that if the page is on bookmarked recipe and we open the bookmarks dropdown then the recipe is highlighted



The beauty of having code structure is that it becomes very quick to deploy new features.

But we can improve this more by refactoring bookmarksView and ResultsView into a new class called PreviewView because the code is almost identical.

refactoring Bookmarks and Results

```
import View from './View.js';  
  
class PreviewView extends View {  
  _parentElement = '';  
  
  _generateMarkup() {  
    const id = window.location.hash.slice(1);  
  
    return `  
      <li class="preview">  
        <a class="preview__link ${  
          this._data.id === id ? 'preview__link--active' : ''  
        }" href="#${this._data.id}">  
          <figure class="preview__fig">  
              
          </figure>  
          <div class="preview__data">  
            <h4 class="preview__title">${this._data.title}</h4>  
            <p class="preview__publisher">${this._data.publisher}</p>  
          </div>  
        </a>  
      </li>  
    `;  
  }  
}  
  
export default new PreviewView();
```

← We import view.js and build a new class called PreviewView which is a child of view.

← We only need the recipe id

← The renderMarkup will render the html for each li item within the ul parent

```
import View from './View.js';
import previewView from './previewView.js';
import icons from 'url:../../img/icons.svg'; // Parcel 2

class BookmarksView extends View {
  _parentElement = document.querySelector('.bookmarks__list');
  _errorMessage = 'No bookmarks yet. Find a nice recipe and bookmark it ;)';
  _message = '';

  _generateMarkup() {
    return this._data
      .map(bookmark => previewView.render(bookmark, false))
      .join('');
  }
}

export default new BookmarksView();
```



For the generateMarkup of the bookmarks we are calling a render on the previewView class and passing a false value then joining the results together into a single string.

```
import View from './View.js';
import previewView from './previewView.js';
import icons from 'url:../../img/icons.svg'; // Parcel 2

class ResultsView extends View {
  _parentElement = document.querySelector('.results');
  _errorMessage = 'No recipes found for your query! Please try again ;)';
  _message = '';

  _generateMarkup() {
    return this._data.map(result => previewView.render(result,
false)).join('');
  }
}

export default new ResultsView();
```



For the generateMarkup of the results we are calling a render on the previewView class and passing a false value then joining the results together into a single string.

```
render(data, render = true) {  
  if (!data || (Array.isArray(data) && data.length === 0))  
    return this.renderError();  
  
  this._data = data;  
  const markup = this._generateMarkup();  
  
  if (!render) return markup;  
  
  this._clear();  
  this._parentElement.insertAdjacentHTML("afterbegin", markup);  
}
```

In the render method we have the second parameter default set to true

When we pass in bookmarks or results with an additional parameter of false we return the markup rather as a string so all the results or bookmarks are joined into one single string of html code.

Storing bookmarks with local storage

```
const persistBookmarks = function () {  
  localStorage.setItem('bookmarks', JSON.stringify(state.bookmarks));  
};
```

```
export const addBookmark = function (recipe) {  
  // Add bookmark  
  state.bookmarks.push(recipe);  
  
  // Mark current recipe as bookmark  
  if (recipe.id === state.recipe.id) state.recipe.bookmarked = true;  
  
  persistBookmarks();  
};
```

```
export const deleteBookmark = function (id) {  
  // Delete bookmark  
  const index = state.bookmarks.findIndex(el => el.id === id);  
  state.bookmarks.splice(index, 1);  
  
  // Mark current recipe as NOT bookmarked  
  if (id === state.recipe.id) state.recipe.bookmarked = false;  
  
  persistBookmarks();  
};
```

In the model.js file we create a method that stringifies the contents of the bookmarks array in the state and adds it to local storage.

The screenshot shows a web browser displaying a recipe application. The application has a sidebar with a list of recipe cards, including "CAULIFLOWER PIZZA CRUST (WITH BBQ CHICKEN PIZZA)", "HOMEMADE PIZZA", "HOW TO GRILL PIZZA", "PIZZA DIP", "VEGGIE PIZZA", "VALENTINE PIZZA", "GREEK PIZZA", "PIZZA DIP", and "PITTA PIZZAS". The main content area shows a detailed view of the "CAULIFLOWER PIZZA CRUST (WITH BBQ CHICKEN PIZZA)" recipe, including a large image of the pizza, a "75 MINUTES" timer, "4 SERVINGS", and a list of "RECIPE INGREDIENTS".

The browser's developer tools are open, showing the Application tab. The Local Storage section is expanded, showing a list of bookmarked recipes:

Key	Value
bookmarks	[{"id": "5ed6604591c37cdc054bcd3", "title": "Sausage Pasta", "publisher": "All Recipes", ...}]

Now we see the bookmarked recipes in local storage.

But if we reload the page then the bookmarks remain in local storage but disappear from the bookmarks dropdown. We need a method that retrieves bookmarks from local storage and renders them in the dropdown.

```
const init = function () {  
  const storage = localStorage.getItem('bookmarks');  
  if (storage) state.bookmarks = JSON.parse(storage);  
};  
init();
```

```
const clearBookmarks = function () {  
  localStorage.clear('bookmarks');  
};  
// clearBookmarks();
```

In the model.js we have an init method that gets the bookmarks from local storage then parses them and adds them into the state.bookmarks property. This function is invoked after.

We add a clearBookmarks method for housekeeping.

Bug fix #2

However, this init function causes an error and needs to be debugged.

```
✖ ▶ TypeError: Cannot set properties of undefined (setting  
  'textContent')  
    at View.js:37:26  
    at Array.forEach (<anonymous>)  
    at BookmarksView.update (View.js:27:17)  
    at controlRecipes (controller.js:25:19)
```

The stack trace shows the error to be coming from line 25 of the controller.

```
bookmarksView.update(model.state.bookmarks);
```

```
// 1) Updating bookmarks view  
debugger;  
bookmarksView.update(model.state.bookmarks);
```

In the controller before the line that originates the error, we use a debugger;

The issue is coming from our DOM update algorithm. We have an array of current elements on the page and an array of new elements which we are comparing. The goal of this method is to replace the text of elements that have changed however the bookmark data is empty so it has nothing to replace. Note how current elements is zero

The screenshot shows a web browser at `localhost:1234/#5ed6604591c37cdc054bcd09`. The browser's developer tools are open, showing the `Sources` panel with the `View.js` file selected. The code is at line 27, where a red arrow points to the line `i = 0` inside a `newElements.forEach` loop. The code is as follows:

```
23 const newDOM = document.createRange().createContextualFragment(newMarkup);
24 const newElements = Array.from(newDOM.querySelectorAll("*"));
25 const curElements = Array.from(this._parentElement.querySelectorAll("*"));
26
27 newElements.forEach((newEl, i) => {
28   const curEl = curElements[i]; i = 0
29   // console.log(curEl, newEl.isEqualNode(curEl));
30
31   // Updates changed TEXT
32   if (
33     !newEl.isEqualNode(curEl) && newEl = li.preview {value: 0, type: '', title: '', lang: '', translate: tr
34     newEl.firstChild?.nodeValue.trim() !== ""
35   ) {
36     // console.log('X', newEl.firstChild.nodeValue.trim());
37     curEl.textContent = newEl.textContent; curEl = div.preview {align: '', title: '', lang: '', translate:
38   }
39
40   // Updates changed ATTRIBUTES
41   if (!newEl.isEqualNode(curEl)) newEl = li.preview {value: 0, type: '', title: '', lang: '', translate: tr
42     Array.from(newEl.attributes).forEach((attr) =>
43       curEl.setAttribute(attr.name, attr.value) curEl = div.preview {align: '', title: '', lang: '', transl
44     });
45   });
46 }
47
48 _clear() {
49   this._parentElement.innerHTML = "";
50 }
51
52 renderSpinner() {
53   const markup = `
54     <div class="spinner">
55       <svg>
56         <use href="${icons}#icon-loader"></use>
57       </svg>
58     </div>
```

The right-hand side of the developer tools shows the `Debugger` panel, which is paused. The `Scope` panel shows the `Local` scope with the following variables:

- `Return value`: undefined
- `this`: undefined
- `curEl`: `div.preview`
- `i`: 0
- `newEl`: `li.preview`
- `Closure (update)`
- `Closure (5cUXS.url1:../img/icons.svg)`
- `Global`: `Window`

The `Call Stack` panel shows the following stack:

- `(anonymous)` at `View.js:44`
- `update` at `View.js:27`
- `controlRecipes` at `controller.js:26`
- `load (async)`
- `(anonymous)` at `recipeView.js:13`
- `addHandlerRender` at `recipeView.js:13`
- `init` at `controller.js:125`
- `aenu9.core-js/modules/es.array.includes.js` at `controller.js:132`
- `newRequire` at `index.e37f48ea.js:71`

```
import View from './View.js';
import previewView from './previewView.js';
import icons from 'url:../../img/icons.svg'; // Parcel 2
```

```
class BookmarksView extends View {
  _parentElement = document.querySelector('.bookmarks__list');
  _errorMessage = 'No bookmarks yet. Find a nice recipe and bookmark it ;)';
  _message = '';
```

```
  addHandlerRender(handler) {
    window.addEventListener('load', handler);
  }
```

In the bookmarks.js we
add a render handler that
is listening for a load event



```
  _generateMarkup() {
    return this._data
      .map(bookmark => previewView.render(bookmark, false))
      .join('');
  }
}

export default new BookmarksView();
```

```
const init = function () {  
  bookmarksView.addHandlerRender(controlBookmarks);  
  recipeView.addHandlerRender(controlRecipes);  
  recipeView.addHandlerUpdateServings(controlServings);  
  recipeView.addHandlerAddBookmark(controlAddBookmark);  
  searchView.addHandlerSearch(controlSearchResults);  
  paginationView.addHandlerClick(controlPagination);  
};  
init();
```

In the controller.js we point the render handler to controlBookmarks as the first thing to be executed.

```
const controlBookmarks = function () {  
  bookmarksView.render(model.state.bookmarks);  
};
```

In the controller.js the controlBookmarks renders with model state bookmarks.

```
const controlRecipes = async function () {
  try {
    const id = window.location.hash.slice(1);

    if (!id) return;
    recipeView.renderSpinner();

    // 0) Update results view to mark selected search result
    resultsView.update(model.getSearchResultsPage());

    // 1) Updating bookmarks view
    bookmarksView.update(model.state.bookmarks);

    // 2) Loading recipe
    await model.loadRecipe(id);

    // 3) Rendering recipe
    recipeView.render(model.state.recipe);
  } catch (err) {
    recipeView.renderError();
    console.error(err);
  }
};
```

In the controller.js the update bookmarks method is moved to the top so that this is in a more logical order.



Now the bookmarks load before the recipe and all works well.

Forkify App: Building a Modern Application

Part 6 – Implement recipe upload

Implementing Recipe Upload

```
<div class="overlay hidden"></div>
  <div class="add-recipe-window hidden">
    <button class="btn--close-modal">&times;</button>
    <form class="upload">
      <div class="upload__column">
        <h3 class="upload__heading">Recipe data</h3>
        <label>Title</label>
        <input value="TEST23" required name="title" type="text" />
        <label>URL</label>
        <input value="TEST23" required name="sourceUrl" type="text" />
        <label>Image URL</label>
        <input value="TEST23" required name="image" type="text" />
        <label>Publisher</label>
        <input value="TEST23" required name="publisher" type="text" />
        <label>Prep time</label>
        <input value="23" required name="cookingTime" type="number" />
        <label>Servings</label>
        <input value="23" required name="servings" type="number" />
      </div>
```

In the index.html file the modal window to add a recipe is already there but with a hidden class.

```
import View from './View.js';
import icons from 'url:../../img/icons.svg'; // Parcel 2
```

```
class AddRecipeView extends View {
  _parentElement = document.querySelector('.upload');
  _message = 'Recipe was successfully uploaded :)'

  _window = document.querySelector('.add-recipe-window');
  _overlay = document.querySelector('.overlay');
  _btnOpen = document.querySelector('.nav__btn--add-recipe');
  _btnClose = document.querySelector('.btn--close-modal');
```

1) In the addRecipeView.js file we need to assign variables to all the buttons and DOM elements that we want to manipulate

```
constructor() {
  super();
  this._addHandlerShowWindow();
}
```

3) But because this window is independent of other features in the project and the open/close is only going to be used inside this class we can add a constructor. The super() allows us to attach the this keyword to a child class.

```
  _addHandlerShowWindow() {
    this._btnOpen.addEventListener('click', function () {
      this._overlay.classList.toggle('hidden');
      this._window.classList.toggle('hidden');
    });
  }
```

2) we want a handler to listen for the event of opening or closing modal window of add recipe. As usual we will add a handler to do this. When the window is opened we want to toggle the overlay and the window. Toggle will add or remove hidden class from the window and overlay.

```
export default new AddRecipeView();
```

```
import * as model from './model.js';
import recipeView from './views/recipeView.js';
import searchView from './views/searchView.js';
import resultsView from './views/resultsView.js';
import paginationView from './views/paginationView.js';
import bookmarksView from './views/bookmarksView.js';
import addRecipeView from './views/addRecipeView.js';
```

4) We need to
import this class in
the controller.

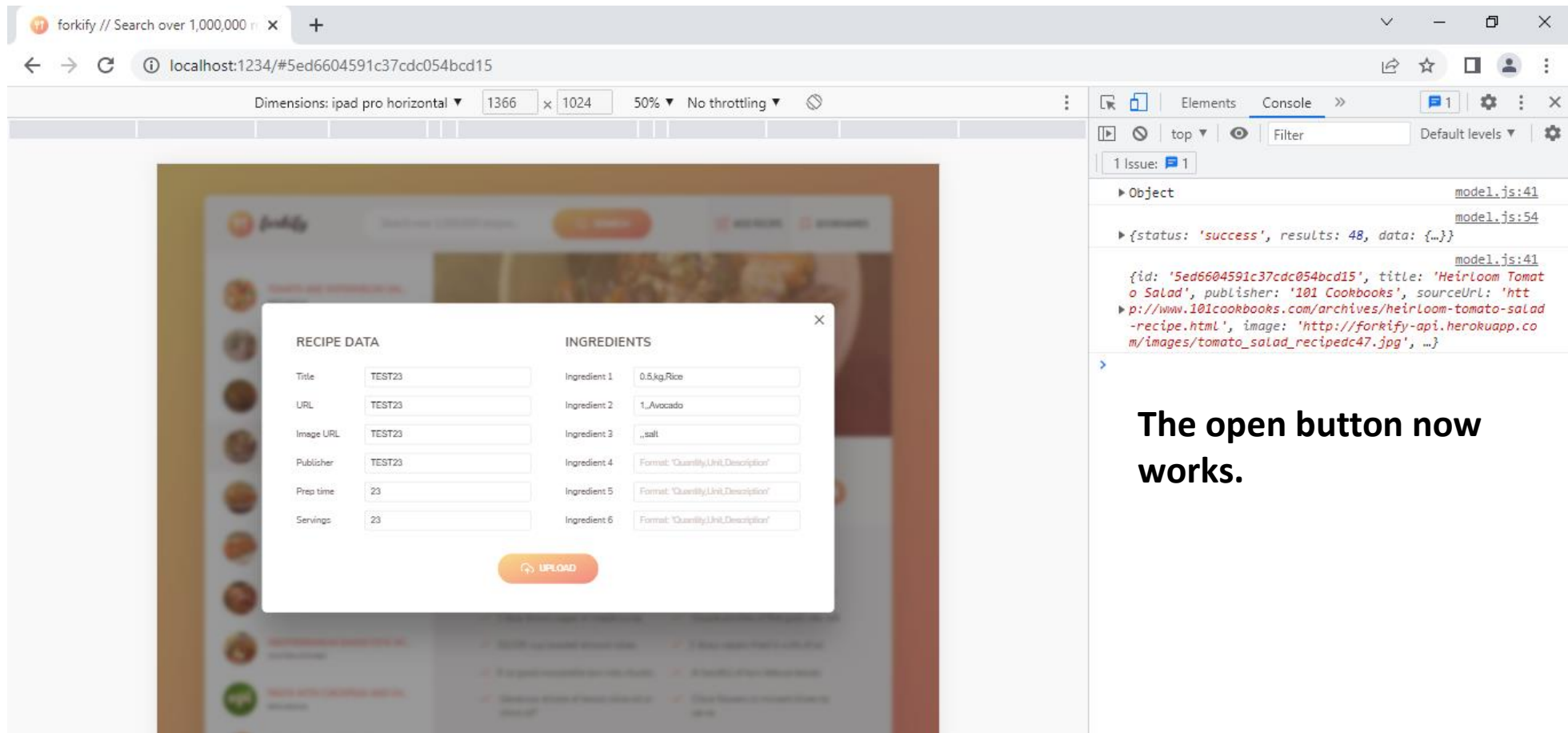
But this code produces a bug: we are using the `this` keyword inside of a handler and so the `this` keyword points to the element to which it is attached, in this case the open button when we need the `this` keyword bound to the overlay and window.

```
toggleWindow() {
  this._overlay.classList.toggle('hidden');
  this._window.classList.toggle('hidden');
}
```

So in a separate toggle window function
we will set the `this` keyword to the overlay
and window.

```
_addHandlerShowWindow() {
  this._btnOpen.addEventListener('click', this.toggleWindow.bind(this));
}
```

Then bind the `this` keyword



The open button now works.

```
_addHandlerHideWindow() {
  this._btnClose.addEventListener('click', this.toggleWindow.bind(this));
  this._overlay.addEventListener('click', this.toggleWindow.bind(this));
}

constructor() {
  super();
  this._addHandlerShowWindow();
  this._addHandlerHideWindow();
}
```

We can add a handler for the hide modal window.

And call the hide in the constructor

```
addHandlerUpload(handler) {  
  this._parentElement.addEventListener('submit', function (e) {  
    e.preventDefault();  
    const dataArr = [...new FormData(this)];  
    handler(dataArr);  
  });  
}
```

Now we need to get the data from the form when the upload button is clicked. to do this we can attach another event listener to the parent element which is the form. We could select the data one by one but it is better to use the new API method of FormData which will create a weird object. We can spread this Object into an array. Note the this keyword is the data because this is set to the parent element which is the form.

```
const controlAddRecipe = function (newRecipe) {  
  console.log(newRecipe);  
};
```

```
const init = function () {  
  bookmarksView.addHandlerRender(controlBookmarks);  
  recipeView.addHandlerRender(controlRecipes);  
  recipeView.addHandlerUpdateServings(controlServings);  
  recipeView.addHandlerAddBookmark(controlAddBookmark);  
  searchView.addHandlerSearch(controlSearchResults);  
  paginationView.addHandlerClick(controlPagination);  
  addRecipeView.addHandlerUpload(controlAddRecipe);  
};  
init();
```

The action of uploading the data is just another API call so we need a controller for the handler event.

When we click the upload button this is the data array that gets logged to the console. Within the array are a series of sub arrays each containing the name of the field and the value but we would rather this is an object than an array because that is how recipes are stored in the state and how they are in the API.

forkify // Search over 1,000,000 recipes

localhost:1234/#5ed6604591c37cdc054bcd15

Dimensions: ipad pro horizontal 1366 x 1024 50% No throttling

Recipe Data

Title

TEST23

URL

TEST23

Image URL

TEST23

Publisher

TEST23

Prep time

23

Servings

23

Ingredients

Ingredient 1

0.5,kg,Rice

Ingredient 2

1,,Avocado

Ingredient 3

,,salt

Ingredient 4

Format: "Quantity,Unit,Description"

Ingredient 5

Format: "Quantity,Unit,Description"

Ingredient 6

Format: "Quantity,Unit,Description"

Upload

Console

model.js:41

{id: '5ed6604591c37cdc054bcd15', title: 'Heirloom Tomato Salad', publisher: '101 Cookbooks', sourceUrl: 'http://www.101cookbooks.com/archives/heirloom-tomato-salad-recipe.html', image: 'http://forkify-api.herokuapp.com/images/tomato_salad_recipecdc47.jpg', ...}

controller.js:92

(12) [Array(2), Array(2), Array(2), Array(2), Array(2), Array(2), Array(2), Array(2), Array(2), Array(2), Array(2), Array(2)]

▶ 0: (2) ['title', 'TEST23']

▶ 1: (2) ['sourceUrl', 'TEST23']

▶ 2: (2) ['image', 'TEST23']

▶ 3: (2) ['publisher', 'TEST23']

▶ 4: (2) ['cookingTime', '23']

▶ 5: (2) ['servings', '23']

▶ 6: (2) ['ingredient-1', '0.5,kg,Rice']

▶ 7: (2) ['ingredient-2', '1,,Avocado']

▶ 8: (2) ['ingredient-3', ',,salt']

▶ 9: (2) ['ingredient-4', '']

▶ 10: (2) ['ingredient-5', '']

▶ 11: (2) ['ingredient-6', '']

length: 12

▶ [[Prototype]]: Array(0)

```

addHandlerUpload(handler) {
  this._parentElement.addEventListener('submit', function (e) {
    e.preventDefault();
    const dataArr = [...new FormData(this)];
    const data = Object.fromEntries(dataArr);
    handler(data);
  });
}

```

The ES2019 method of `Object.FromEntries` will convert an array into an object.

```

{title: 'TEST23', sourceUrl: 'TEST23', image: 'TEST23', publisher: 'TEST2
3', cookingTime: '23', ...}
  cookingTime: "23"
  image: "TEST23"
  ingredient-1: "0.5,kg,Rice"
  ingredient-2: "1,,Avocado"
  ingredient-3: ",,salt"
  ingredient-4: ""
  ingredient-5: ""
  ingredient-6: ""
  publisher: "TEST23"
  servings: "23"
  sourceUrl: "TEST23"
  title: "TEST23"
  [[Prototype]]: Object

```

```

  ingredients: Array(10)
    0: {quantity: 2, unit: 'pounds', description: '/ 1 kg tomatoes halved'}
    1: {quantity: 0.25, unit: 'cup', description: '/ 60 ml extra virgin olive oil'}
    2: {quantity: 1, unit: 'tbsp', description: 'brown sugar or maple syrup'}
    3: {quantity: null, unit: '', description: 'Couple pinches of fine grain sea salt'}
    4: {quantity: 0.33, unit: 'cup', description: 'toasted almond slices'}
    5: {quantity: 2, unit: 'tbsps', description: 'capers fried in a bit of oil'}
    6: {quantity: 6, unit: 'oz', description: 'good mozzarella torn into chunks'}
    7: {quantity: null, unit: '', description: 'A handful of torn lettuce leaves'}
    8: {quantity: null, unit: '', description: 'Generous drizzle of lemon olive oil or }
    9: {quantity: null, unit: '', description: 'Chive flowers or minced chives to serve
length: 10

```

Now that we have the recipe as an object (left) we need to format the ingredients to resemble those of our recipe object show on the right.

I.E

ingredient-1: "0.5,kg, Rice" → 0: {Quantity: 0.5, unit: Kg, description: Rice}
 ingredient-2: "1,, Avocado" → 1: {Quantity: 1, unit: '', description: Avocado}
 ingredient-3: ",, salt" → 1: {Quantity: null, unit: '', description: salt}

```

export const uploadRecipe = async function (newRecipe) {
  try {
    const ingredients = Object.entries(newRecipe)
      .filter(entry => entry[0].startsWith('ingredient') && entry[1] !== '')
      .map(ing => {
        const ingArr = ing[1].split(',').map(el => el.trim());
        // const ingArr = ing[1].replaceAll(' ', '').split(',');
        if (ingArr.length !== 3)
          throw new Error('Wrong ingredient format! Please use the correct format :')
        );

        const [quantity, unit, description] = ingArr;

        return { quantity: quantity ? +quantity : null, unit, description };
      });
  }
};

```

We need to convert the new recipe object back into an array using the `Object.entries` method then filter the array entries for elements that start with `ingredient` and are not blank.

[ingredient-1: "0.5,Kg,Rice"] where `ingredient` is [0] and "0.5,Kg,Rice" is [1].

Now we have filtered the array to remove empty ingredients.

Next we can map over the array with each entry taking the data out of the string ("0.5,Kg,Rice") and putting it in an object. We use `split`(by coma), `map` and `trim` (to remove whitespaces) to do this. If the object has less than 3 items we throw an error.

Next we can destructure the array into `[quantity, unit, description]` `ingArr` and return an object with the format {Quantity: 0.5, unit: Kg, description: Rice}.


```
const recipe = {
  title: newRecipe.title,
  source_url: newRecipe.sourceUrl,
  image_url: newRecipe.image,
  publisher: newRecipe.publisher,
  cooking_time: +newRecipe.cookingTime,
  servings: +newRecipe.servings,
  ingredients,
};

const data = await snedJSON(`${API_URL}`);

addBookmark(state.recipe);
} catch (err) {
  throw err;
}
};
```

Now we can build a new recipe object taking all but the ingredients from the new recipe passed into this function. Then we add the ingredients array.

Now we can perform an AJAX call to the API_URL with the key (see following slide) and recipe.

```

export const sendJSON = async function (url, uploadData) {
  try {
    const fetchPro = fetch(url, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(uploadData),
    });

    const res = await Promise.race([fetchPro, timeout(TIMEOUT_SEC)]);
    const data = await res.json();

    console.log(recipie);
    sendJSON(`${API_URL}`);
  } catch (err) {
    throw err;
  }
};

```

to send data we need to POST data. We need to specify Content-Type is application/json so the API knows what format we are sending the data in to upload it to the database.

We need to specify the body passing in the uploadData and JSON stringifying it.

We also need to pass in a URL for the fetch request.

As per the GET recipes data we will use a race against a time out function, to stop it running forever.

From the Forkify documentation <https://forkify-api.herokuapp.com/v2> we can see a button to generate an API key (one key per hour permitted) and we can store the key in the config.js file.

```
export const API_URL = "https://forkify-api.herokuapp.com/api/v2/recipes/";
export const TIMEOUT_SEC = 10;
export const RES_PER_PAGE = 10;
export const KEY = ' ';
```

```
https://forkify-api.herokuapp.com/api/v2/recipes&Key=<your API key>
```

From the Forkify documentation we can see that the URL to upload must have an API key.

```
sendJSON(` ${API_URL}$key=${key}, recipe` );
```

Now we can modify the SendJSON call to include the key in the URL and then add the recipe.

```
const createRecipeObject = function (data) {  
  const { recipe } = data.data;  
  return {  
    id: recipe.id,  
    title: recipe.title,  
    publisher: recipe.publisher,  
    sourceUrl: recipe.source_url,  
    image: recipe.image_url,  
    servings: recipe.servings,  
    cookingTime: recipe.cooking_time,  
    ingredients: recipe.ingredients,  
  };  
};
```

Now in model.js, when we get back a response from the upload new recipe, it also contains the recipe object so we need to convert it into the format we use to load recipe. Rather than write this code twice we can copy the code and put it in its own method, called CreateRecipeObject.

```
export const loadRecipe = async function (id) {  
  try {  
    const data = await getJSON(` ${API_URL}${id}`);  
    state.recipe = createRecipeObject(data);  
  
    console.log(recipe);  
    const data = await sendJSON(` ${API_URL}$key=${key}, recipe`);  
    state.recipe = createRecipeObject(data);  
  } catch (err) {  
    throw err;  
  }  
};
```

Then in the recipeLoad method we just call the createRecipeObject passing in the data.

And we can reuse the createRecipeObject method again in our UploadRecipe method.

```

console.log(recipe);
    const data = await sendJSON(` ${API_URL}$key=${key}, recipe`);
    state.recipe = createRecipeObject(data);
    state.addBookmark (recipe);
} catch (err) {
    throw err;
}
};

```

And we also need to add the uploaded recipe to bookmarks using the previously created addBookmark method.

```

const createRecipeObject = function (data) {
    const { recipe } = data.data;
    return {
        id: recipe.id,
        title: recipe.title,
        publisher: recipe.publisher,
        sourceUrl: recipe.source_url,
        image: recipe.image_url,
        servings: recipe.servings,
        cookingTime: recipe.cooking_time,
        ingredients: recipe.ingredients,
        ...(recipe.key && { key: recipe.key }),
    };
};

```

In the case of the uploaded recipes the recipe object should also contain the key. But only the uploaded recipes have a key, not the ones fetched from the API.

The and operator short circuits. i.e. if the recipe key value is falsy then nothing happens so the destructuring does nothing. if the key is a truthy value then the recipe key value is spread and put in the object. This is a trick to conditionally add to an object.

```

const controlAddRecipe = async function (newRecipe) {
  try {
    // 1. Upload the new recipe data
    await model.uploadRecipe(newRecipe);
    console.log(model.state.recipe);

    // 2. Render recipe
    recipeView.render(model.state.recipe);

    // 3. Success message
    addRecipeView.renderMessage();

    // 4. Close form window
    setTimeout(function () {
      addRecipeView.toggleWindow();
    }, MODAL_CLOSE_SEC * 1000);
  } catch (err) {
    console.error('❌', err);
    addRecipeView.renderError(err.message);
  }
};

```

Now in the controlAddRecipe method we can render the uploaded recipe and close modal window. But rather than closing the modal window straight away we can add a timeout function to delay this by 2.5 seconds. Note that the MODAL_CLOSE_SEC is a parameter defined in the config.js file.

```

class AddRecipeView extends View {
  _parentElement = document.querySelector('.upload');
  _message = 'Recipe was successfully uploaded :)'

```

And in the AddRecipeView class we can add a success message and call this message from the controlAddRecipe method.



Search over 1,000,000 recipes...

SEARCH

ADD RECIPE

BOOKMARKS

When a new recipe is uploaded, The recipe renders and is added to the bookmarks.



CAULIFLOWER PIZZA CRUST (WIT...
CLOSET COOKING



HEIRLOOM TOMATO SALAD
101 COOKBOOKS



TEST23
TEST23

TEST23

23 MINUTES

23 SERVINGS



RECIPE INGREDIENTS

- ✓ 1/2 kg Rice
- ✓ 1 Avocado
- ✓ salt

HOW TO COOK IT

This recipe was carefully designed and tested by TEST23. Please check out directions at their website.

```
const controlAddRecipe = async function (newRecipe) {  
  try {  
    // 1. Show loading spinner  
    addRecipeView.renderSpinner();  
  
    // 2. Upload the new recipe data  
    await model.uploadRecipe(newRecipe);  
    console.log(model.state.recipe);  
  
    // 3. Render recipe  
    recipeView.render(model.state.recipe);  
  
    // 4. Success message  
    addRecipeView.renderMessage();  
  
    // 5. Close form window  
    setTimeout(function () {  
      addRecipeView.toggleWindow();  
    }, MODAL_CLOSE_SEC * 1000);  
  } catch (err) {  
    console.error('💣', err);  
    addRecipeView.renderError(err.message);  
  }  
};
```

When the recipe is being uploaded we want a spinner so that the user knows something is happening.


```

const controlAddRecipe = async function (newRecipe) {
  try {
    // 1. Show loading spinner
    addRecipeView.renderSpinner();

    // 2. Upload the new recipe data
    await model.uploadRecipe(newRecipe);
    console.log(model.state.recipe);

    // 3. Render recipe
    recipeView.render(model.state.recipe);

    // 4. Success message
    addRecipeView.renderMessage();

    // 5. Render bookmark view
    bookmarksView.render(model.state.bookmarks);

    // 6. Change ID in URL
    window.history.pushState(null, '', `#${model.state.recipe.id}`);

    // 7. Close form window
    setTimeout(function () {
      addRecipeView.toggleWindow();
    }, MODAL_CLOSE_SEC * 1000);
  } catch (err) {
    console.error('❌', err);
    addRecipeView.renderError(err.message);
  }
};

```

To finish the upload recipe functionality we need to fix a few small details.

- 1) The bookmarks are not re-rendered.
- 2) The URL of the new recipe when it renders does not change.

We use render here and not update because we want to insert a new element, not just change the text of existing elements.

we can use the history API of the browser pushState method. This takes parameters of state, unused, and url where state is null, unused is a depreciated thing that we omit, and the URL we add in.

<https://developer.mozilla.org/en-US/docs/Web/API/History/pushState>

Finally, getJSON and SendSON are basically the same and can be refactored into AJAX in the helper.js file.

```
export const AJAX = async function (url, uploadData = undefined) {
```

```
  try {
```

```
    const fetchPro = uploadData
```

```
      ? fetch(url, {
```

```
        method: 'POST',
```

```
        headers: {
```

```
          'Content-Type': 'application/json',
```

```
        },
```

```
        body: JSON.stringify(uploadData),
```

```
      })
```

```
      : fetch(url);
```

```
    const res = await Promise.race([fetchPro,
```

```
      timeout(TIMEOUT_SEC)]);
```

```
    const data = await res.json();
```

```
    if (!res.ok) throw new Error(`${data.message} (${res.status})`);
```

```
    return data;
```

```
  } catch (err) {
```

```
    throw err;
```

```
  }
```

```
};
```

The difference between get data and post data is that the post receives uploadData as an argument. We can use this by setting the default of upload data to undefined for the get.

Then we define a variable called fetch pro and in a ternary conditional we define the post parameters for the fetch url or if it is a get then this will be the false part of the ternary which just has fetch url without additional parameters.

The rest of the function with the promise race and error handling is the same as getJSON or sendJSON so can be copied.

```
import { async } from 'regenerator-runtime';
import { API_URL, RES_PER_PAGE, KEY } from './config.js';
// import { getJSON, sendJSON } from './helpers.js';
import { AJAX } from './helpers.js';
```

Now in the model.js file we can import this new AJAX named method and comment out the old getJson and sendJson

```
export const loadRecipe = async function (id) {
  try {
    const data = await AJAX(` ${API_URL}${id}`);
    state.recipe = createRecipeObject(data);
  }

  export const loadSearchResults = async function (query) {
    try {
      state.search.query = query;

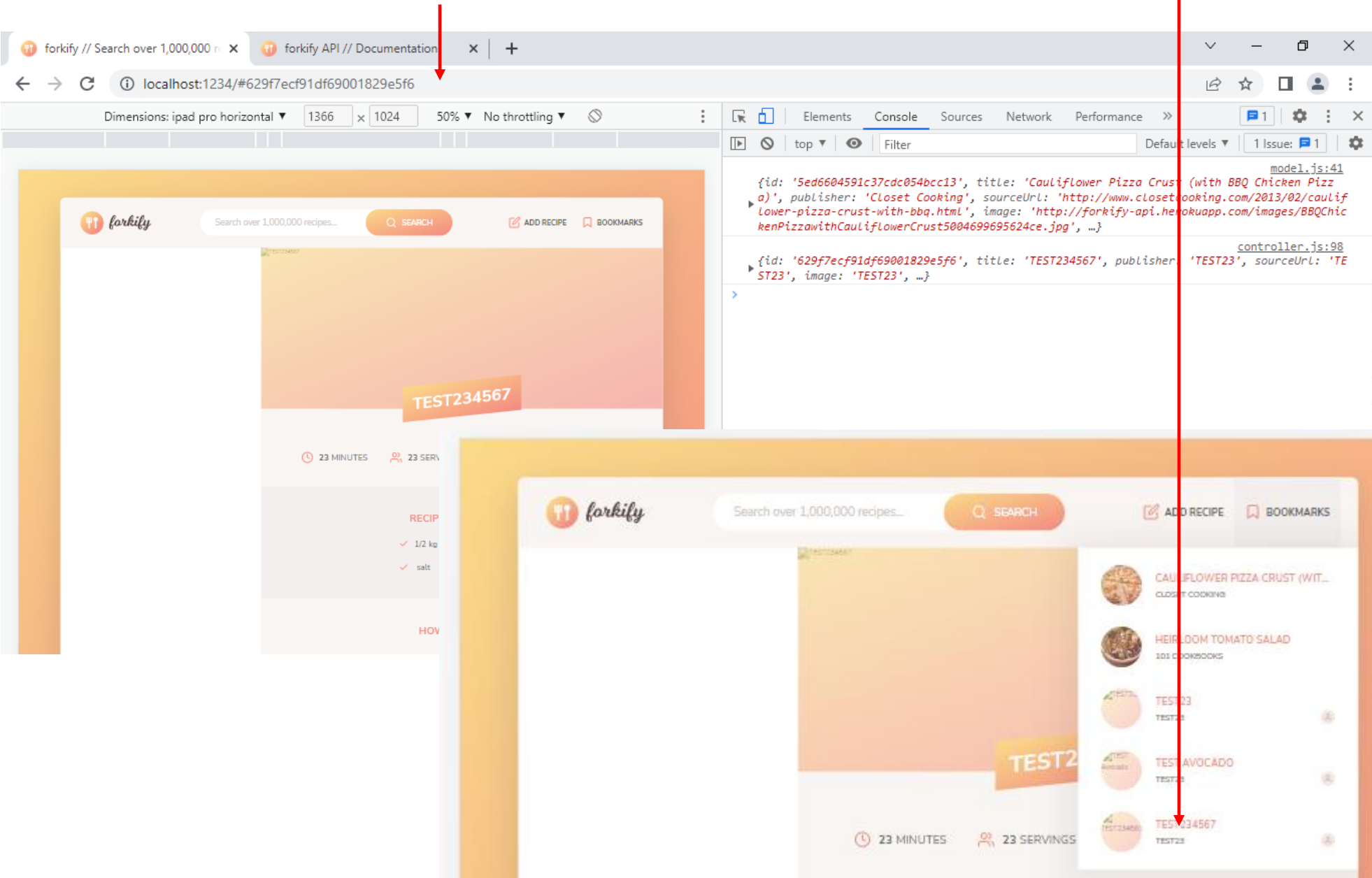
      const data = await AJAX(` ${API_URL}?search=${query}`);

    export const uploadRecipe = async function (newRecipe) {

    ...
      const data = await AJAX(` ${API_URL}?key=${KEY}`, recipe);
      state.recipe = createRecipeObject(data);
      addBookmark(state.recipe);
    } catch (err) {
      throw err;
    }
  };
};
```

We also need to change getJSON or sendJSON to AJAX wherever we have used it in the model.js code.

Checking it again, the upload recipe function still works. Also note that the URL now changes when the new recipe is loaded. Also the new recipe is in the bookmarks.



The last feature to implement is to mark user submitted recipes when they are loaded on the page and in the search results.

```
export const loadRecipe = async function (id) {  
  try {  
    const data = await AJAX(`${API_URL}${id}?key=${KEY}`);  
    state.recipe = createRecipeObject(data);
```

To do this we also pass the key when we load a recipe or search for a recipe.

```
export const loadSearchResults = async function (query) {  
  try {  
    state.search.query = query;  
  
    const data = await AJAX(`${API_URL}?search=${query}&key=${KEY}`);
```

```
<div class="recipe__user-generated ${this._data.key ? '' : 'hidden'}">  
  <svg>  
    <use href="${icons}#icon-user"></use>  
  </svg>  
</div>  
  <button class="btn--round btn--bookmark">  
    <svg class="">  
      <use href="${icons}#icon-bookmark${this._data.bookmarked ? '-fill' : ''}"></use>  
    </svg>  
  </button>  
</div>
```

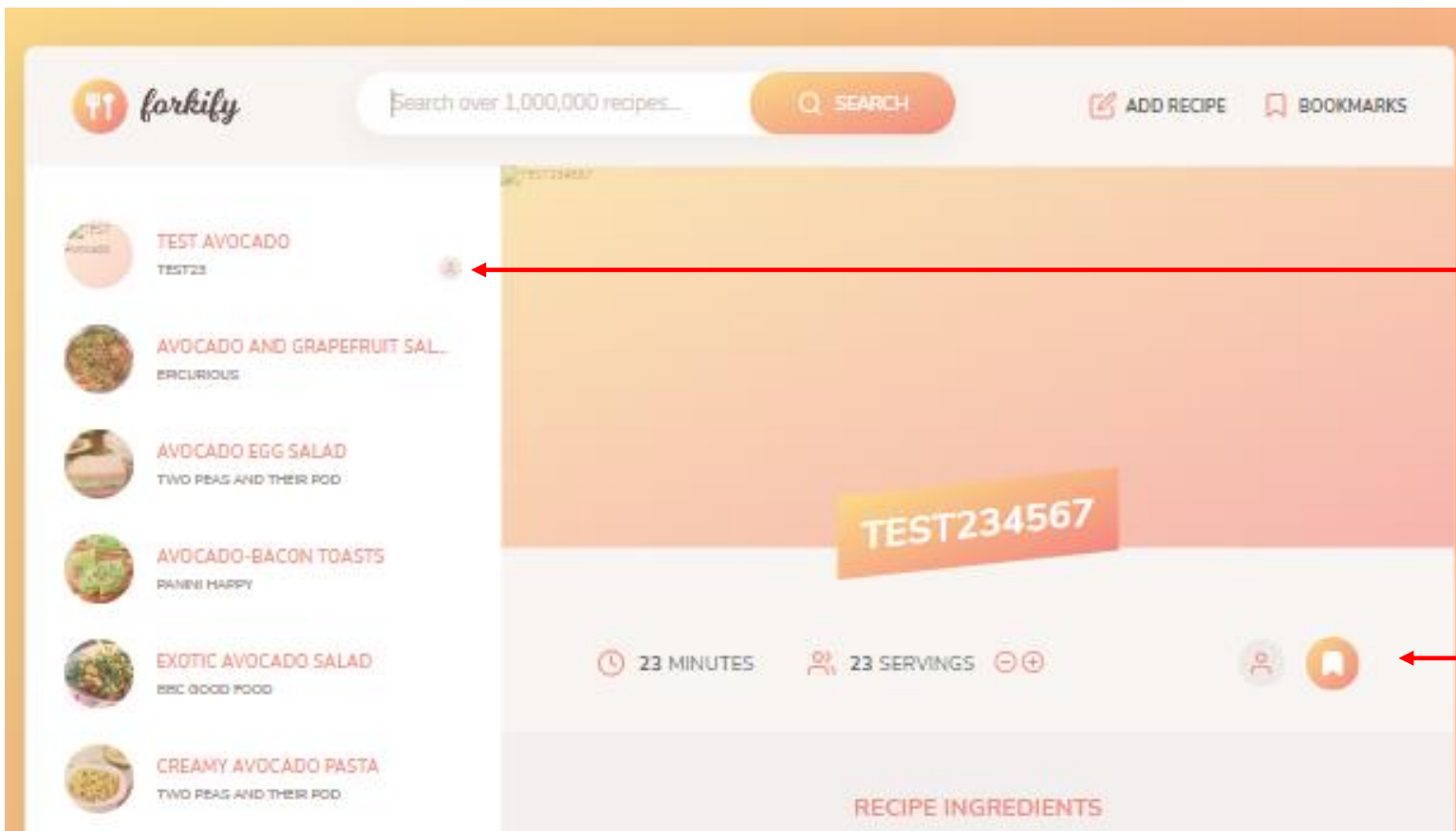
To get the user icon to show, in the recipeView.js file under the renderMarkup function we add a "hidden" class on the user-generated div if there is no key.

```
export const loadSearchResults = async function (query) {
  try {
    state.search.query = query;

    const data = await AJAX(` ${API_URL}?search=${query}&key=${KEY} `);
    console.log(data);

    state.search.results = data.data.recipes.map(rec => {
      return {
        id: rec.id,
        title: rec.title,
        publisher: rec.publisher,
        image: rec.image_url,
        ...(rec.key && { key: rec.key }),
      };
    });
    state.search.page = 1;
  } catch (err) {
    console.error(` ${err} ❌❌❌❌ `);
    throw err;
  }
};
```

To get user submitted recipes to show
in the search results we have to add the
key.



Now the user submitted recipe shows up in the search results with the user icon.

And in recipe view with the user icon not hidden.

The project is now complete and all features are implemented.

Forkify App: Building a Modern Application

Part 7 – Adding Annotation to Code using JS.docs Standard

Adding Annotations to Code

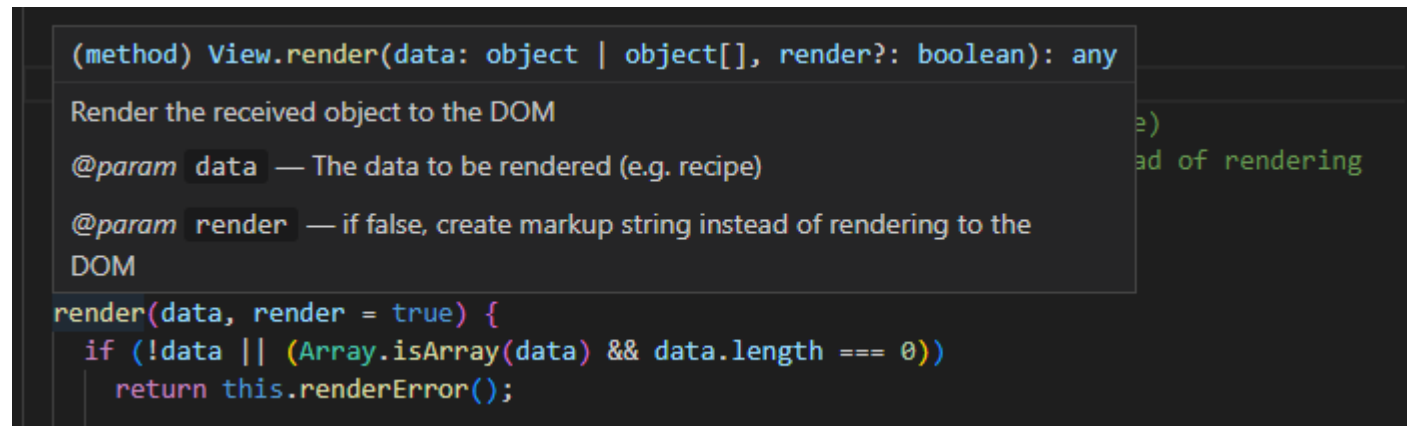
There is a standard to annotate JavaScript projects so that other developers can find their way around. The standard is defined at <https://jsdoc.app>

```
export default class View {  
  _data;
```

Below is an example of an annotation that we could add to the function of render. Inside the curly braces are details about what parameters this function accepts and text descriptions.

```
  /**  
   * Render the received object to the DOM  
   * @param {object | object[]} data The data to be rendered (e.g. recipe)  
   * @param {boolean} [render=true] if false, create markup string instead of  
   rendering to the DOM  
   */  
  
  render(data, render = true) {  
    if (!data || (Array.isArray(data) && data.length === 0))  
      return this.renderError();  
  }
```

Now when we hover over the render method this annotation information is shown in a pop up. Anywhere else in our code where we call the render method this pop up also shows.



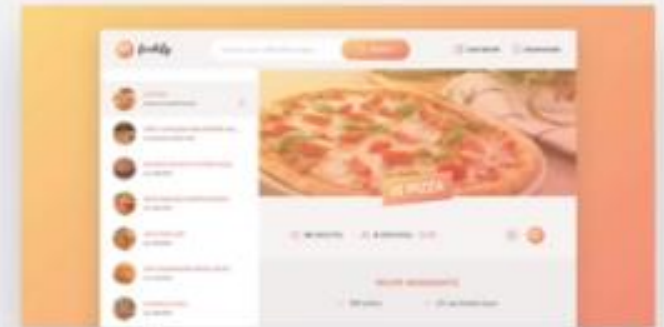
```
/**
 * Render the received object to the DOM
 * @param {object | object[]} data The data to be rendered (e.g. recipe)
 * @param {boolean} [render=true] if false, create markup string instead of rendering to the DOM
 * @returns {undefined | string} a markup string is returned if render is false
 * @this {object} view instance
 * @author Elliott Farmer
 * @todo There are no items pending
 */
```

We can add lots more information to the annotation. More but not all of the examples could include what the method returns, what the this keyword is set to, the author, todo list etc...

Suggested Student Upgrades to the Forkify App.

The knowledge gained from this course will be deployed in improvements to the UX/UI of www.barcelonaTravelHacks.com, my website.

- 👉 Display **number of pages** between the pagination buttons;
- 👉 Ability to **sort** search results by duration or number of ingredients;
- 👉 Perform **ingredient validation** in view, before submitting the form;
- 👉 **Improve recipe ingredient input**: separate in multiple fields and allow more than 6 ingredients;
- 👉 **Shopping list feature**: button on recipe to add ingredients to a list;
- 👉 **Weekly meal planning feature**: assign recipes to the next 7 days and show on a weekly calendar;
- 👉 **Get nutrition data** on each ingredient from spoonacular API (<https://spoonacular.com/food-api>) and calculate total calories of recipe.



Setting up GIT and Deployment

Simple Deployment with Netlify.

Now the project is finished we can prepare it for deployment. Step one is to delete the dist and parcel-cache folder, then modify the package.json file.

“main” is changed to “default”

We also set the build folder output to --dist-dir and then specify the name and path to the distribution directory.

Make sure the package.json file is saved.

Then run the `npm run build` command in terminal.

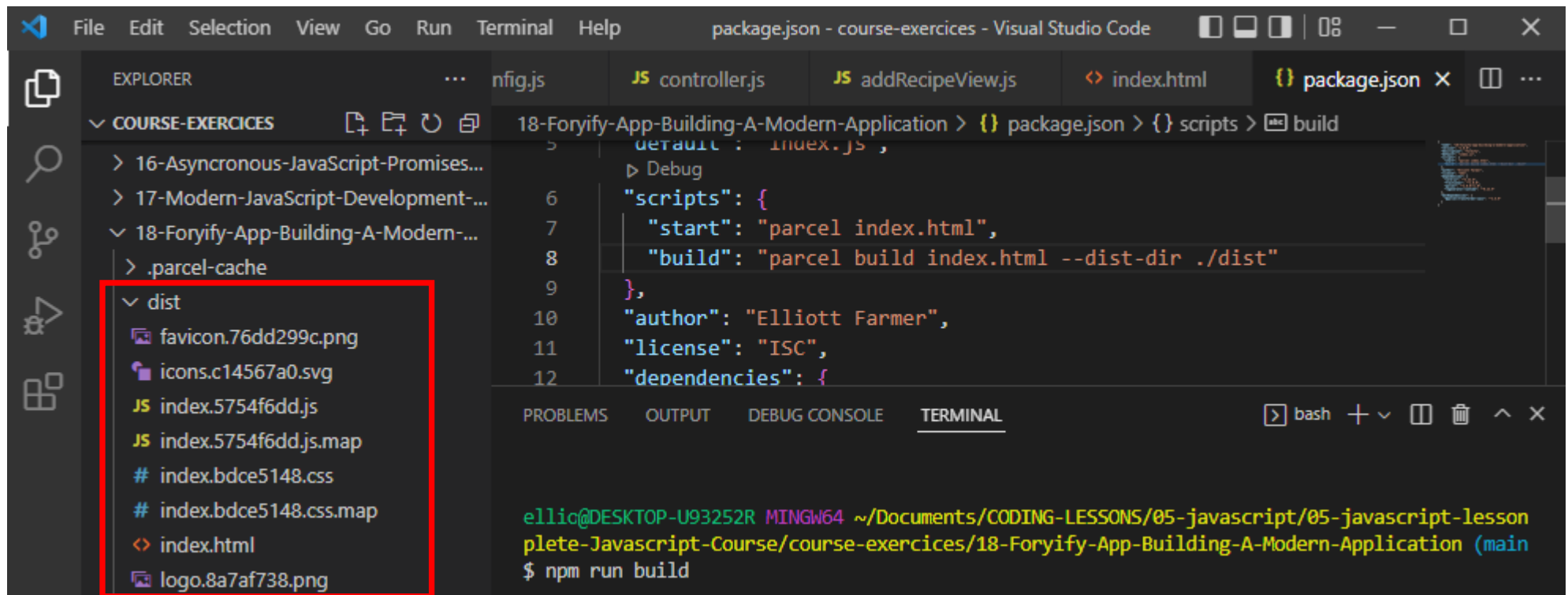
```
{
  "name": "18-foryify-app-building-a-modern-application",
  "version": "1.0.0",
  "description": "forkify",
  "default": "index.js",
  "scripts": {
    "start": "parcel index.html",
    "build": "parcel build index.html --dist-dir ./dist"
  },
  "author": "Elliott Farmer",
  "license": "ISC",
  "dependencies": {
    "core-js": "^3.22.8",
    "fractional": "^1.0.0",
    "parcel": "^2.0.0-rc.0",
    "regenerator-runtime": "^0.13.9"
  },
  "devDependencies": {
    "@parcel/transformer-sass": "^2.6.0"
  }
}
```

```
ellic@DESKTOP-U93252R MINGW64 ~/Documents/CODING-LESSONS/05-javascript/05-javascript-lessons/Complete-Javascript-Course/course-exercices/18-Foryify-App-Building-A-Modern-Application (main)
$ npm run build

> 18-foryify-app-building-a-modern-application@1.0.0 build
> parcel build index.html --dist-dir ./dist

✨ Built in 3.67s

dist\index.html          3.9 KB    1.22s
dist\favicon.76dd299c.png 8.25 KB   132ms
dist\logo.8a7af738.png   57.71 KB   1.13s
dist\icons.c14567a0.svg   8.81 KB   1.57s
dist\index.bdce5148.css   9.89 KB   1.12s
dist\index.5754f6dd.js    35.39 KB   1.49s
dist\icons.c14567a0.svg   8.81 KB   1.21s
```



This will produce our final code ready for deployment

<https://www.netlify.com> is a place where we can host a static site for free if we open an account. A static site is one that contains html, css, JavaScript code and any image files. We cannot deploy any database or server side code. The site is deployed to a CDN so a user will be directed to the nearest CDN server when going to the URL speeding up page loading.

<https://surge.sh> is a similar place to host a static site (web publishing) but without the need to open an account.

With netlify we can create a new project by dropping the dist folder onto the new project section of the website.

In github I have a .gitignore file to exclude uploading of node_modules, dist folder and parcel_cache.

Netlify allows us to configure continuous deployment which means that when the code changes in the repository the live website is automatically updated

Continuous Deployment

Settings for Continuous Deployment from a Git repository

Repository

Your site is linked to a Git repository, which means you can take advantage of Continuous Deployment [build settings](#).

Current repository:  github.com/EL-EL-BCN/forkify

[Learn more about Continuous Deployment in the docs](#) ↗

Manage repository ▾

We need to define the remote repository

Build settings

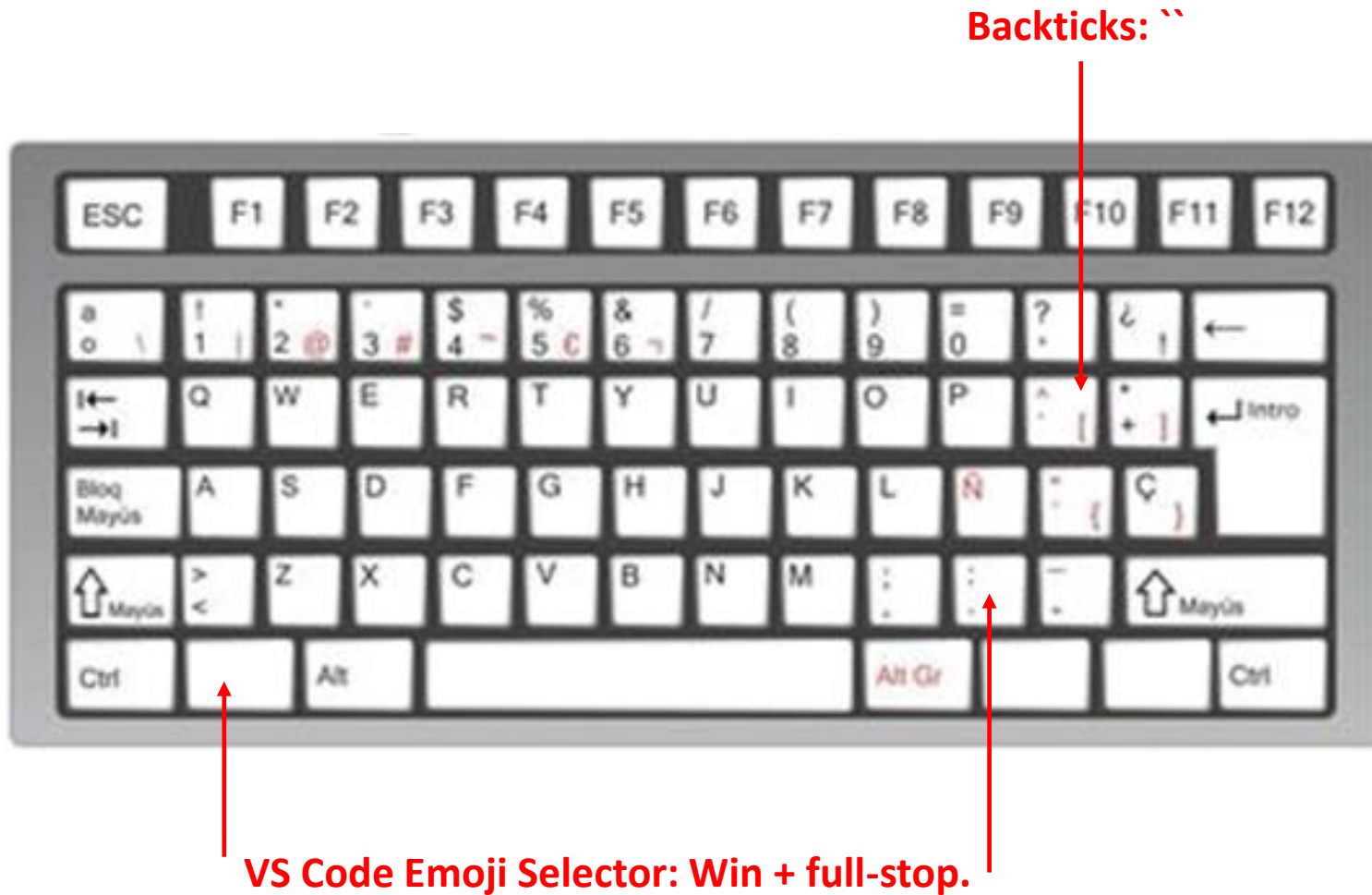
Base directory:	Not set
Build command:	parcel build index.html --dist-dir ./dist
Publish directory:	dist
Builds:	Active

[Learn more about common configuration directives in the docs](#) ↗

Edit settings

We need to define the build command and the publish directory.

Spanish Keyboard New Keys Introduced in this Course



Course Content

Inline vs External | JS releases | dynamic typing | let, const & var | Arithmetic operators | comparison operators | assignment operators | Operator precedence | Strings & Template Literals | If Else statements | Type Conversion & Coercion | Truthy & Falsy Values | Equality Operators == or === | Boolean Logic | Logical Operators And Or Not | Switch Statement | Statements & Expressions | Conditional Ternary Operator | JavaScript Releases: ES5, ES6+ and ESNext | activating Strict Mode | JavaScript Functions | function Declarations vs Expressions | Arrow Function | Functions calling other functions | Arrays littoral syntax, Inbuilt function | multi-dimensional array | populate array with function values | Array methods | Introduction to objects | dot or bracket object notation | Object methods | this object | Iteration – for loop | Looping arrays - breaking and Continuing | Looping backwards | loops within loops | while loops | config of vs code – prettier, live server | debugging in console | debugging with breakpoints | HTML & CSS basics | DOM theory | DOM manipulation | DOM events | JavaScript Engine | call stack & Heap | Just-in-Time Optimisation | hoisting | DMZ | scope | Primitive types | Reference types | this keyword | self keyword | object.assign | de-structuring and array | de-structuring and object | spread operator | Rest Pattern and Parameters | Short Circuiting (&& and ||) | Nullish coalescing Operator (??) | Logical Assignment Operators | The for of loop | ES6 Enhanced object literals | Optional Chaining (?) | Object keys, Values & Entries | sets | Maps | What data structure to use? Array, Object, Set, Map | Working with strings | Es6 function default parameters | passing arguments: Value vs Reference | First class & High-Order functions | Functions Accepting Callback Functions | Functions Returning other Functions | The call and Apply Methods | The bind Method | Immediately Invoked Function Expressions (IIFE) | The closure | arr.splice | arr.splice | arr.join | arr.reverse | arr.at | forEach loop

Course Content

innerHTML | insertAdjacentHTML | Data Transformations: Map, Filter, Reduce | Chaining Methods | Find Method | FindIndex Method | Some and Every methods | flat, map and flatMap methods | array.fill and Array.from | Number.parseInt | Number.parseFloat | Number.isNaN | Number.isFinite | Number.isInteger | Math.sqrt | Math.max | Math.min | Math.PI | Math.trunc | Math.random | Math.round | Math.ceil | Math.floor | Math.ToFixed | Remainder Operator | Numeric Separators | BigInt | new Date | DateObjects | toISOString | unix datetime | timestamp | Intl.DateTimeFormat | Intl.NumberFormat | Timers: set timeout and Set Interval | countdownTimer | DOM API | DOM node, EventTarget, window, Element node, Text node, Comment node & document node | DOM Inheritance | DOM node methods & properties | selecting elements – document.querySelector/all(), getElementsById(), getElementsByClassName(), getElementsByTagName() | Inserting elements - insertAdjacentHTML(), createElement() | classList.add(), prepend/append(), cloneNode(), before/after() | Deleting Elements - remove(), parentElement.removeChild() | styles: inline style, getComputedStyle | setProperty() | getAttribute() | setAttribute() | dataset | classList.add, remove, toggle & contains | smooth scrolling: getBoundingClientRect(), clientHeight/Width, scrollTo(), scrollIntoView() | addEventListener(), removeEventListener(), eventTimeout | Event Propagation: Capture target & bubbling phases | e.stopPropagation() | listening in capturing phase | Event Delegation: Implementing Page Navigation | DOM Traversing: childNodes, children, firstElementChild/last, parentNode, parentElement, closest(), previousElementSibling/next | Building a Tabbed Component | Guard clause | sticky Navigation: The scroll Event vs Intersection Observer API | Revealing Elements on scroll: Intersection Observer API | Lazy Loading Images: Intersection Observer API | Building a dynamic slider component | Lifecycle DOM events: DOM content loaded, load event, before unload event | Script loading: head, end of body, Async & Defer |

Course Content

Object Orientated Javascript: Object & prototype | Constructor Functions | New Operator | Prototypal Inheritance | Prototype Chain | ES6 classes | Static Methods | Inheritance between Classes: Constructor Functions, ES6 classes, Object.create() | prototype chain `__proto__` | parent/child classes: Constructor Functions, ES6 classes, Object.create() | Encapsulation: Protected properties, Private/public fields, private/public methods | extends, super | chaining public methods | How to plan a web project | Project Architecture | Geolocation API | Leaflet Library | load external library | Implement workout app with leaflet map using OOP classes | Asynchronous/Synchronous JavaScript | new XMLHttpRequest | Request & Response theory | Callback Hell | Promises and the fetch API | The Promise Lifecycle: Pending, Settled, Fulfilled, Rejected | Build Promise | Consume Promise: fetch(), then(), json() | Chaining Promises, common mistakes | Handling Rejected Promises: catch(), finally() | manually throw Error: throw new Error() | Asynchronous Behind the Scenes: The Event Loop, Web API environment, Callback queue, microTasks queue, call stack | Practical demonstration of the event loop and microtask priority | Building a Promise | Promisify Callback Function | Async/Await promises | try...catch error handling | Running Promises in parallel: Promise.all | Other Promise Combinators: race(), allSettled(), any() | Modern JavaScript Development: Modules, third-party | Tools: babel, Webpack, parcel | How ES6 Modules are Imported | Exporting & Importing ES6: Named Export, Default Export, import, as, from | Top Level Await (ES2022) | CommonJS Modules overview | command line NPM install | NPM installing packages: leaflet, lodash-es | restore NPM packages from cloned git repo | Bundling with parcel | Configuring Babel and Pollyfilling | Writing Clean and Modern JavaScript | Declarative and functional JavaScript Principles | Building forkify app in a NPM parcel/babel/pollyfilled core-js environment | Event listener .forEach() event in array | Project architecture: MVC model |

Course Content

Components of any Architecture | Model-View-Controller (MVC) Architecture | MVC implementation in Forkify App | Refactoring for the MVC model | Fractional NPM package | Refactoring the html markup | Helpers and configuration files | Event handlers in MVC: Publisher-Subscriber Design Pattern | Event handlers in MVC: Implementation | MVC: Implementing Error and Success Messages | Refactoring classes | implementing search results pagination | Develop an Update DOM algorithm | Develop an update DOM algorithm | Implement bookmarks in local storage | small bug fixes | stack trace debugger | Implement upload recipe: FormData(), class constructor(), super(), refactoring GET and POST ajax calls, window.history.pushState (), API_KEY | annotating according to JsDoc Standard | publishing with netlify | npm build | continuous deployment from git repo