

In this module, I provided a brief introduction into some core next-gen JavaScript features, of course focusing on the ones you'll see the most in this course. Here's a quick summary!

let & const

Read more about `let` : <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let>

Read more about `const` : <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const>

`let` and `const` basically replace `var` . You use `let` instead of `var` and `const` instead of `var` if you plan on never re-assigning this "variable" (effectively turning it into a constant therefore).

ES6 Arrow Functions

Read more: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Arrow functions are a different way of creating functions in JavaScript. Besides a shorter syntax, they offer advantages when it comes to keeping the scope of the `this` keyword (see [here](#)).

Arrow function syntax may look strange but it's actually simple.

```
.  function callMe(name) {  
.      console.log(name);  
.  }
```

which you could also write as:

```
.  const callMe = function(name) {  
.    console.log(name);  
.  }
```

becomes:

```
.  const callMe = (name) => {  
.    console.log(name);  
.  }
```

Important:

When having **no arguments**, you have to use empty parentheses in the function declaration:

```
.  const callMe = () => {  
.    console.log('Max!');  
.  }
```

When having **exactly one argument**, you may omit the parentheses:

```
.  const callMe = name => {  
.    console.log(name);  
.  }
```

When **just returning a value**, you can use the following shortcut:

```
.  const returnMe = name => name
```

That's equal to:

```
.  const returnMe = name => {  
.    return name;  
.  }
```

Exports & Imports

In React projects (and actually in all modern JavaScript projects), you split your code across multiple JavaScript

files - so-called modules. You do this, to keep each file/module focused and manageable.

To still access functionality in another file, you need **export** (to make it available) and **import** (to get access) statements.

You got two different types of exports: **default** (unnamed) and **named** exports:

default => `export default ...;`

named => `export const someData = ...;`

You can import **default exports** like this:

```
import someNameOfYourChoice from './path/to/file.js';
```

Surprisingly, `someNameOfYourChoice` is totally up to you.

Named exports have to be imported by their name:

```
import { someData } from './path/to/file.js';
```

A file can only contain one default and an unlimited amount of named exports. You can also mix the one default with any amount of named exports in one and the same file.

When importing **named exports**, you can also import all named exports at once with the following syntax:

```
import * as upToYou from './path/to/file.js';
```

`upToYou` is - well - up to you and simply bundles all exported variables/functions in one JavaScript object. For example, if you `export const someData = ... (/path/to/file.js)` you can access it on `upToYou` like this: `upToYou.someData`.

Classes

Classes are a feature which basically replace constructor functions and prototypes. You can define blueprints for JavaScript objects with them.

Like this:

```
. class Person {  
.   constructor () {  
.     this.name = 'Max';  
.   }  
. }  
.   
. const person = new Person();  
. console.log(person.name); // prints 'Max'
```

In the above example, not only the class but also a property of that class (=> `name`) is defined. The syntax you see there, is the "old" syntax for defining properties. In modern JavaScript projects (as the one used in this course), you can use the following, more convenient way of defining class properties:

```
. class Person {  
.   name = 'Max';  
. }  
.   
. const person = new Person();  
. console.log(person.name); // prints 'Max'
```

You can also define methods. Either like this:

```

.   class Person {
.       name = 'Max';
.       printMyName () {
.           console.log(this.name); // this is required to refer
.           to the class!
.       }
.   }
.
.   const person = new Person();
.   person.printMyName();

```

Or like this:

```

.   class Person {
.       name = 'Max';
.       printMyName = () => {
.           console.log(this.name);
.       }
.   }
.
.   const person = new Person();
.   person.printMyName();

```

The second approach has the same advantage as all arrow functions have: The `this` keyword doesn't change its reference.

You can also use **inheritance** when using classes:

```

.   class Human {
.       species = 'human';
.   }
.
.   class Person extends Human {
.       name = 'Max';
.       printMyName = () => {
.           console.log(this.name);
.       }
.   }
.
.   const person = new Person();

```

```
. person.printMyName();  
. console.log(person.species); // prints 'human'
```

Spread & Rest Operator

The spread and rest operators actually use the same syntax: `...`

Yes, that is the operator - just three dots. It's usage determines whether you're using it as the spread or rest operator.

Using the Spread Operator:

The spread operator allows you to pull elements out of an array (=> split the array into a list of its elements) or pull the properties out of an object. Here are two examples:

```
. const oldArray = [1, 2, 3];  
. const newArray = [...oldArray, 4, 5]; // This now is [1, 2,  
    3, 4, 5];
```

Here's the spread operator used on an object:

```
. const oldObject = {  
.   name: 'Max'  
. };  
. const newObject = {  
.   ...oldObject,  
.   age: 28  
. };
```

`newObject` would then be

```
. {  
.   name: 'Max',  
.   age: 28  
. }
```

reference type: objects
and array...not actually
copy, the pointer is
copied, and the copied
pointer will also point to
the same object.

The spread operator is extremely useful for cloning arrays and objects. Since both are reference types (and not primitive type: num...it actually copied the number

Reassign the object or array value is not really copying them, but spread operator creates a real copy that prevent mutation of the copied ones.

primitives), copying them safely (i.e. preventing future mutation of the copied original) can be tricky. With the spread operator you have an easy way of creating a (shallow!) clone of the object or array.

Destructuring

Destructuring allows you to easily access the values of arrays or objects and assign them to variables.

Here's an example for an array:

```
.  const array = [1, 2, 3];  
.  const [a, b] = array;  
.  console.log(a); // prints 1  
.  console.log(b); // prints 2  
.  console.log(array); // prints [1, 2, 3]
```

And here for an object:

```
.  const myObj = {  
.    name: 'Max',  
.    age: 28  
.  }  
.  const {name} = myObj;  
.  console.log(name); // prints 'Max'  
.  console.log(age); // prints undefined  
.  console.log(myObj); // prints {name: 'Max', age: 28}
```

Destructuring is very useful when working with function arguments. Consider this example:

```
.  const printName = (personObj) => {  
.    console.log(personObj.name);  
.  }  
.  printName({name: 'Max', age: 28}); // prints 'Max'
```

Here, we only want to print the name in the function but we pass a complete person object to the function. Of course this is no issue but it forces us to call personObj.name

inside of our function. We can condense this code with destructuring:

```
.  const printName = ({name}) => {  
.    console.log(name);  
.  }  
.  printName({name: 'Max', age: 28}); // prints 'Max'
```

We get the same result as above but we save some code.

By destructuring, we simply pull out the `name` property and store it in a variable/ argument named `name` which we then can use in the function body.

map is a built-in array method: execute on each element in original array and return a new array;

filter
find
findindex
reduce
concat
slice
splice

WHICH ARRAY METHOD TO USE? 🤔

"I WANT...:"

To mutate original array

- ➡ Add to original:

.push (end)

.unshift (start)

- ➡ Remove from original:

.pop (end)

.shift (start)

.splice (any)

The splice() method changes the contents of an array by removing or replacing existing elements and/or adding new elements in place. To access part of an array without modifying it, see slice(). splice(start, deleteCount, item1, item2, itemN)

- ➡ Others:

.reverse

.sort

.fill

The sort() method sorts the elements of an array in place and returns the sorted array. The default sort order is ascending, built upon converting the elements into strings, then comparing their sequences of UTF-16 code units values.

A new array not mutate the original

- ➡ Computed from original:

.map (loop)

- ➡ Filtered using condition:

.filter

The filter() method creates a new array with all elements that pass the test implemented by the provided function.

- ➡ Portion of original:

.slice

The slice() method returns a shallow copy of a portion of an array into a new array object selected from start to end (end not included) where start and end represent the index of items in that array. The original array will not be modified.

- ➡ Adding original to other:

.concat

- ➡ Flattening the original:

.flat

.flatMap

An array index

- ➡ Based on value:

.indexOf

return number (if true)
-1 (if false)

- ➡ Based on test condition:

.findIndex

return number

return boolean

An array element

- ➡ Based on test condition:

.find

return first element

The find() method returns the value of the first element in the provided array that satisfies the provided testing function. If no values satisfy the testing function, undefined is returned.

.at

get last element
array.at(-1)

Know if array includes

- ➡ Based on value:

.includes

- ➡ Based on test condition:

.some

.every

Some is gonna return, if at least one of the elements in the array, satisfies the condition, and every only returns true, if all of the elements, satisfy the condition.

A new string

- ➡ Based on separator string:

.join

to loop over the array, and in each iteration it will execute the callback function here.

To transform to value

- ➡ Based on accumulator:

.reduce

(Boil down array to 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)

.map create a new array

cannot break out of the function

for of can use continue and break.