07/02/2023 11

React

JS NextGen Features: (ES7)

let: new var

const: doesn't change its value

Arrow functions:

- Normal function:

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

- Arrow function: (store it in a let or const name)

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

// for 1 arg func, you can omit () around the arg.

- Arrow function: (One liner return)

```
const multiply = number => number * 2; // omit return
```

// may omit ()

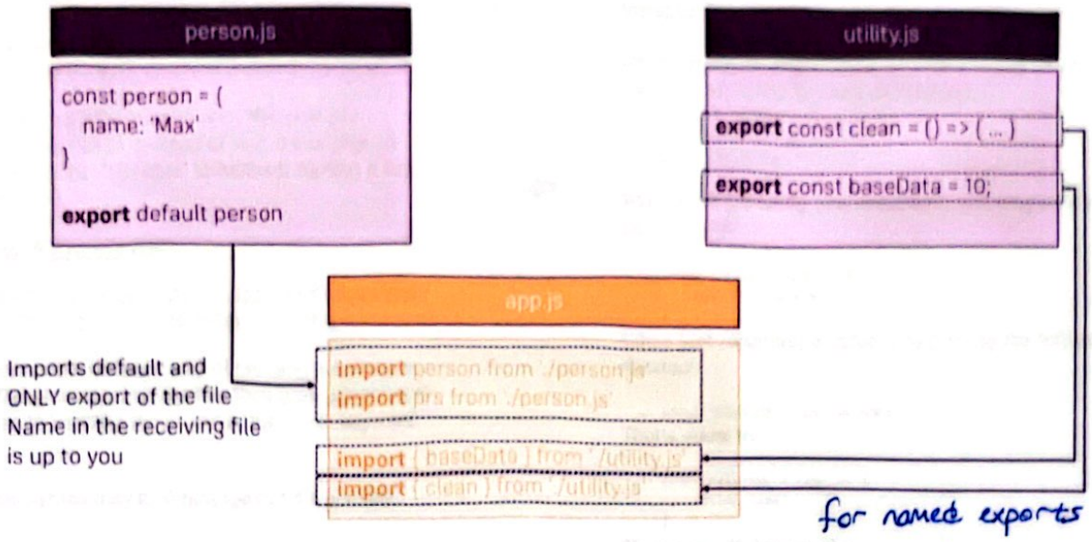
- Arrow function (no args)

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

07/02/2023/2

React

Exports & Imports (Modules)



for named exports (no default) you need to import with exact name.

↳ You can assign an alias:
import { clean as cls } from ...

↳ Or you can import everything in one object:
~~import * as bundled from ...~~

↳ Then use bundled.clean
has imports as properties

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);
```

In the above example, not only the class but also a property of that class (=> `name`) is defined. They 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);
```

You can also define methods. Either like this:

```
class Person {
    name = 'Max';
    printMyName () {
        console.log(this.name);
    }
}

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);
```

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];
```

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
}
```

The spread operator is extremely useful for cloning arrays and objects. Since both are reference types (and not 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);
console.log(b);
console.log(array);
```

And here for an object:

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

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

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

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});
```

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.

07/02/2023/3
React

Reference Type vs. Primitive Type (#19)

→ Objects & arrays are reference types, instead of primitive types.
  ↳ Meaning copies of them refer to the same value in memory.
    If you change one, you change the other.
    ↳ To avoid this, copy array with spread operator:

```
const person = {
  name : 'Max'
};
const secondPerson = {
  ...person
};
```

instead of const secondp=pers

14/02/2023/1

React

Array Methods (#20)

→ They take a function as input,
  ↳ can be an arrow function,
    or a normal function.

then execute it on each element of an array.

↳ E.g. map() is an array function that executes and returns a new array.

```
const numbers = [1, 2, 3];

const doubleNums = numbers.map((num) => {
    return num * 2
});
```

18/02/2023/1

React

REACT

→ React is all about components.

→ Starting a new project:

npx create-react-app appName

→ Running a project you downloaded: First, run in directory (app)

npm install

↳ this will install the dependencies from package.json

these are installed in /node_modules

excluded from git.

→ Then run the development server with:

npm start

↳ This will watch for changes as well.

↳ It also transforms the jsx code.

→ You can import css with: import './index.css'; (same folder as this file)

↳ for js, no need for '.js' part: import App from './App'

component.

a function, returning HTML without quotes ☺ — called → jsx

```
function App() {
  return(
    <p> Text </p>
  );
}
```

19/02/2023

React

→ Declarative Approach

↳ We define the desired target state,
React figures out the IS DOM instructions, to update screen.

→ Best practice to have one file for each component,
named in Capital CamelCase.js format,
with a function in it w/ same name.

→ You build a component tree.

↳ App is special,

↳ Other components

↳ export default FunctionName; // in the component file

↳ import FunctionName from './components/ComponentName';

↳ in App.js

↳ Then you can use imported component
like an HTML element:

<ComponentName> </ComponentName>

↳ MUST start with a capital

or just

<ComponentName />

can also simply do: return HTML
export default MyComp() { ... }

No ".js" here

20/02/2023/1

React

→ Component function must return one root element.

↳ You ~~can~~ must wrap everything into a div,

↳ To span across multiple lines, you should also wrap jsx with parenthesis.

→ To style the components, create a css file in the same dir as the component, with the same name " " " .

↳ Then import css into ComponentName.js :

```
import './ComponentName.css';
```

↳ For applying classes, add them with

```
<div className="...">
```

class also works but...

20/02/2023/2

React (

→ Dynamic data in components:

↳ You can have constants defined in the component func
variables → can be const since it's only const for that instance
const/let

```
const item_description = 'Car insurance';
```

then you can put those into HTML
with {} around them.

```
<p>{item_description}</p>
```

↳ You can run any JS between {} in HTML.

→ Props: properties (attributes) of custom components.

↳ We can move the variables inside the component to App.js:

```
const expenses = [
    {title: 'Car', amount: 25000},
    {...}
];
```

↳ The custom components inside the HTML can take these

```
<MyComponent title={expenses[0].title}>
```

no need for " " around.

↳ These attributes are passed bundled in one parameter. (conventionally named props)
to the component
↳ Its props are these attributes.
↳ These are key-value pairs.

20/02/2023/3

React

→ If you need to transform the data given to component with props, do it before the HTML in the component function, instead of inline between {}.

```
const year = props.date.getFullYear();

return (
  :
  <p>{year}</p>
```

22/02/2023/1

React

→ children is a special prop, always exist,
  ↳ contains the contents between the tags openly and closing the component HTML

→ className for custom components does not automatically add the class, it's just a prop, component needs to take that prop and add it to the HTML it returns:

```
function Card(props) {
  const classes = 'card ' + props.className

  return <div className={classes}> props.children </div>
```

Using useState
0. Import
1. Add inside the component function
use State (props.title);
↳ This returns an array of 2:
1) The state value,
2) Function to update this value
2. Store these 2 functions in 2 constants with array destructuring
↳ First const is a pointer at the value
↳ Second is a function called to set a new value.
3. Whatever event will update the value
will call this function
↳ each time it is called the component function will execute again

Set Title triggers the component function again
↳ Doesn't change the value right away,
instead reschedules it, so it won't be available next line.

23/02/2023/1

React

- Listening to Events
  - Default html event listeners are available in jsx
    like onClick
    - All event handler-props expect a function as value.
      - It can be an inline, anonymous arrow function:
        `<button onClick={() => {...}}> ...</...>`
      - You just give the functions name,
        you don't execute it, with ().
- States:
  - Just because you used a variable in jsx, then changed its value doesn't mean it will update the DOM.
  - If you need to reflect changes in the data in the interface, then you need state.
  - useState: A React function that allows us to define values as state, and changes in it trigger updates
    - This is a react hook; as other hooks, it start w/ use
    - Hooks must be called inside React component functions
      - Not outside, not in nested functions...

23/02/2023/2

React

↳ Using the useState() function:

1. Import

```
import {useState} from "react";
```

2. Add it in the component function:

↳ It will return an array of 2:

1) The state value,
2) A function to update the state value.

3. Store what it returns in 2 constants, with array destructuring

```
const [title, setTitle] = useState(props.title)
```

↳ Use this value in jsx, wherever you want to print it.

initial, default value

4. When changing state value, use the function you named, (call) instead of title = 'New title'

```
setTitle('new title');
```

↳ Each time this function is called, the entire component function will be executed with the new value.

state

→ useState registers a value as the state for the particular instance of the component from which it is called.

i.e: States are on a per-component-instance basis.

↳ When state changes, only that component instance is re-run.

23/02/2023/3

React

→ When the component function is re-executed on a state change, React does not assign the initial, default value (set in the useState function) to the state again; instead, it uses the arg. we gave to the setTitle function.

09/03/2023/1

React

→ Listening to User Input:

↳ Define the event handler function inside the component function,

↳ Add listener to the event. onChange is better than onInput (works on all input types)

↳ Point to the handler (no execution with ())

```
<input onChange={titleChangeHandler} />
```

This function, by default, gets an event object passed to it

↳ `const component = () => {`

`const titleChangeHandler = (event) => {`

↳ event.target.value holds this input's value.

↳ Always a string!

09/03/2023/2

React

→ You can store the input value (event.target.value) in a state.

↳ import {useState} from 'react';

↳ Call it at the beginning of the component func:

```
const ExpenseForm = () => {

    // destructuring
    const [enteredTitle, setEnteredTitle] = useState('');
```

You can use multiple state slices like this;

↳ or you can have 1 state, pass it an object with values {...} instead of a string.

↳ If you chose this, each time you update the object, you have to update all keys of the object passed to useState.

↳ Spread operator can do this.

It ~~brings~~ returns all key-value pairs of an object.

```
...userInput,
// you can overwrite the keys you
   wrote after.
```

09/03/2023/3

React

- Wherever your state update depends on a previous state,
  - do not just pass a new state object by building it

like this:

```
setInput({
        ...userInput,
        enteredDate: event.target.value
    });
```

- Instead, build in on the previous state by passing an anonymous function:

```
setInput((prevState) => {
    return { ...prevState, enteredDate = event.target.value
    };
});
```