ANECO ACADEMY

ES6-ARROW-SPREAD-REST-DESTRUCTING

ARROW FUNCTION

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);
}
```

ARROW FUNCTION

which you could also write as:

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

becomes:

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

ARROW FUNCTION

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;
}
```

CLASS

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. 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); // prints 'Max'
```

You can also define methods. Either like this:

CLASS

```
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.

INHERITANCE

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();
```

SPREAD 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
}
```

DESCTRUCTING

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