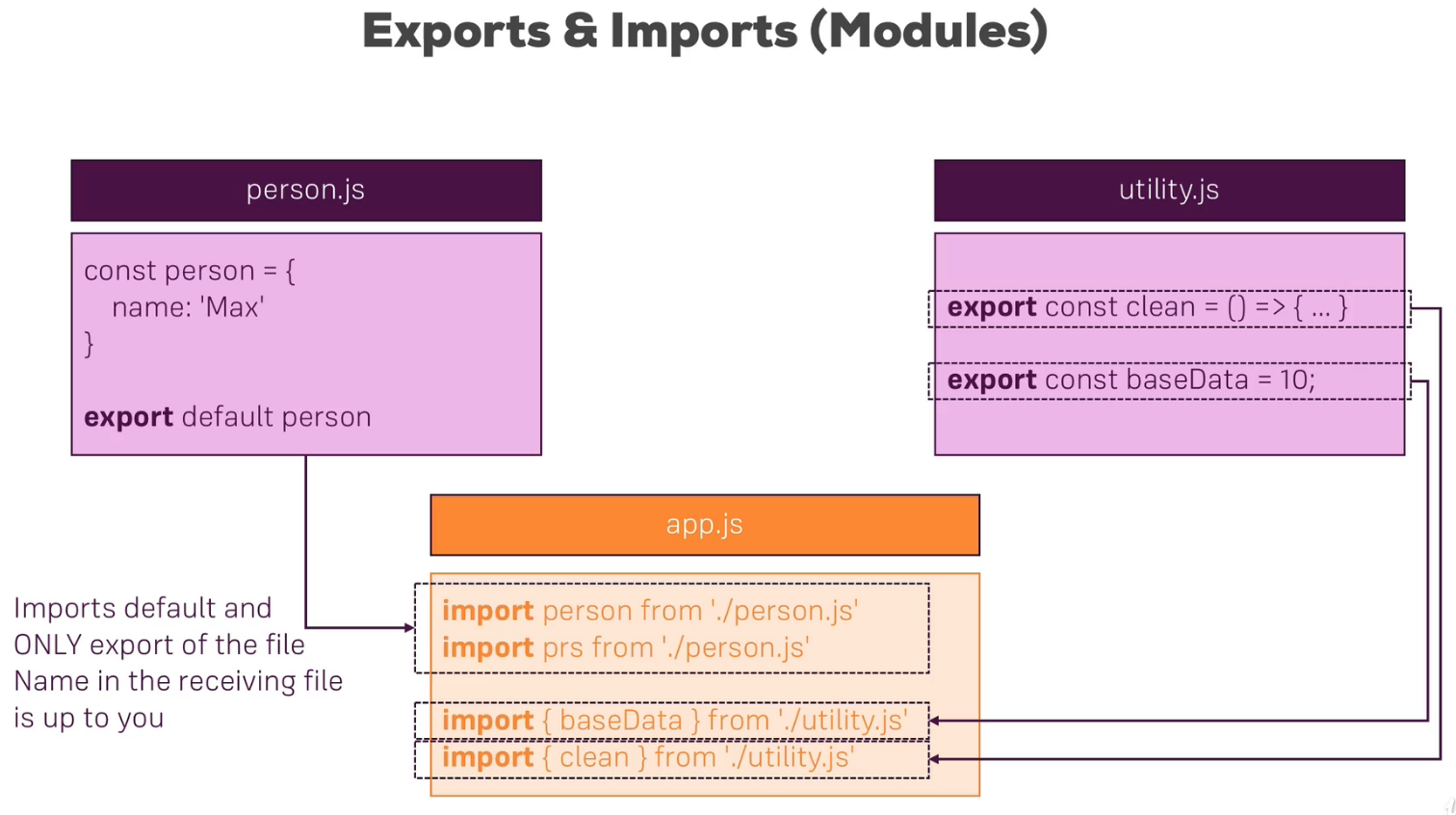
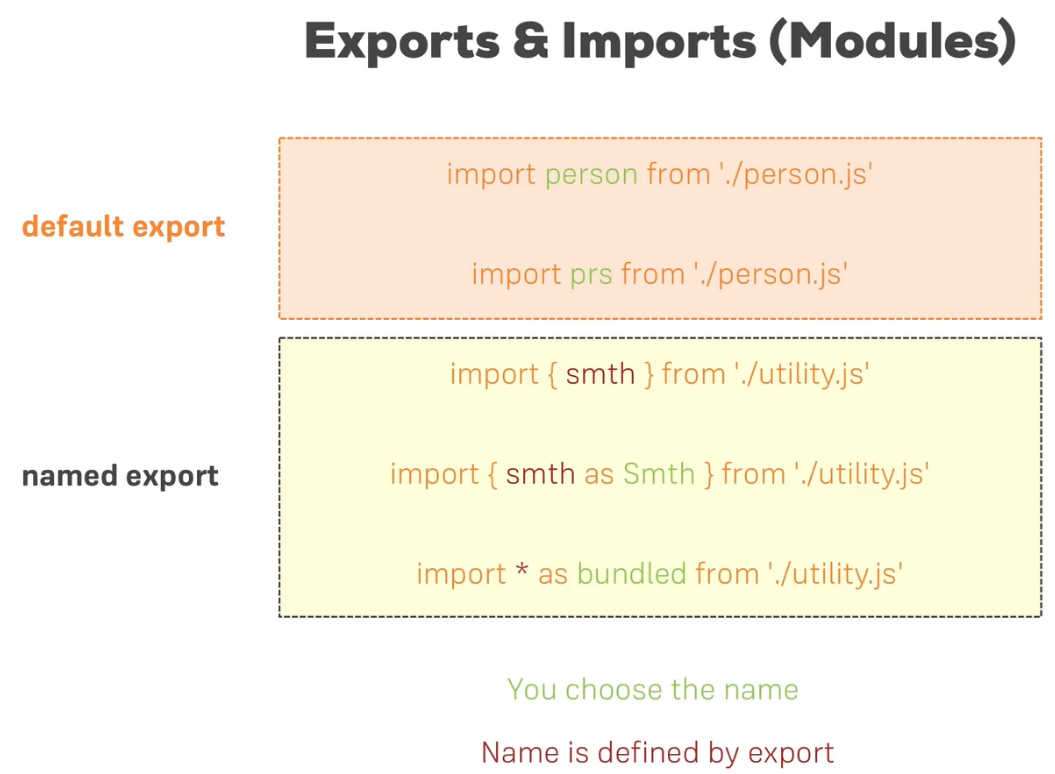
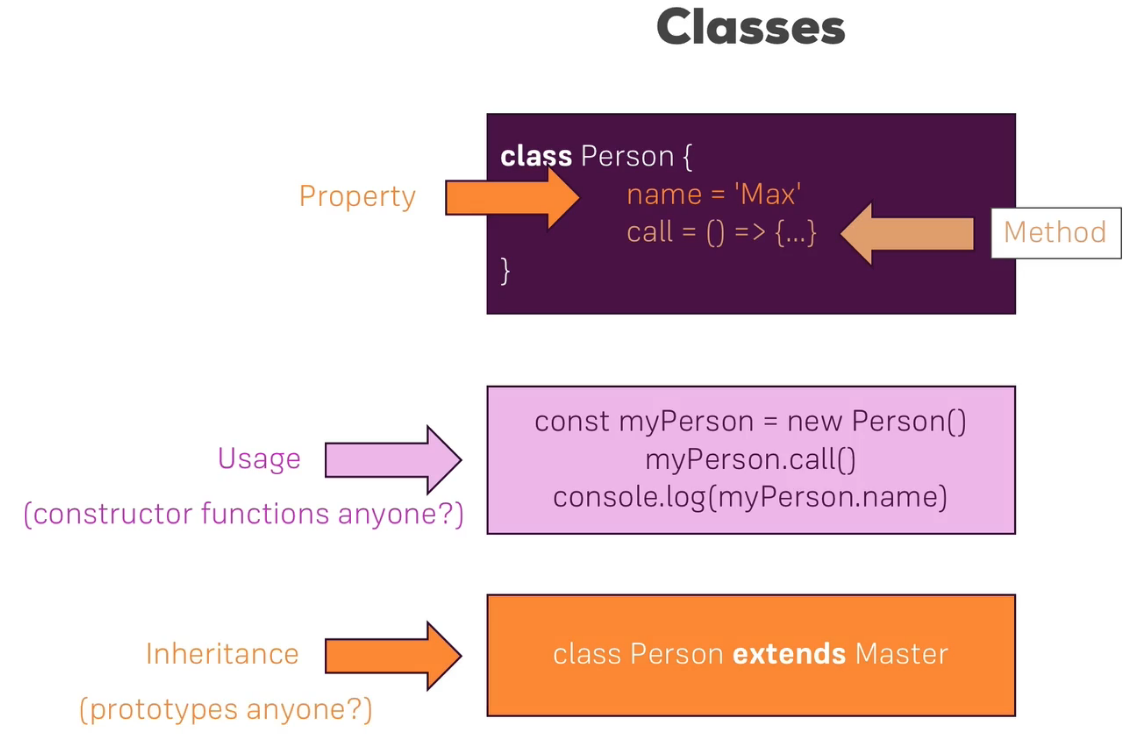
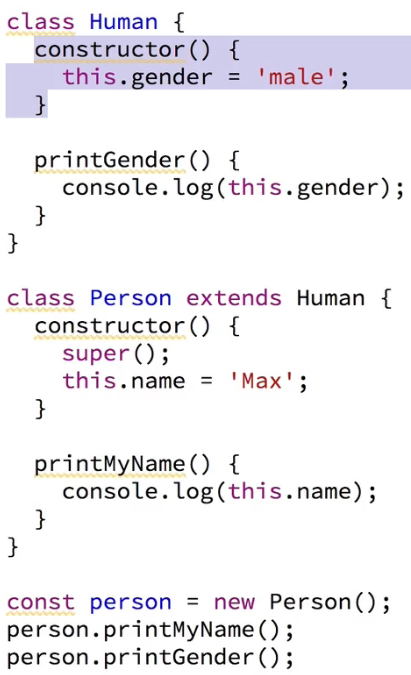
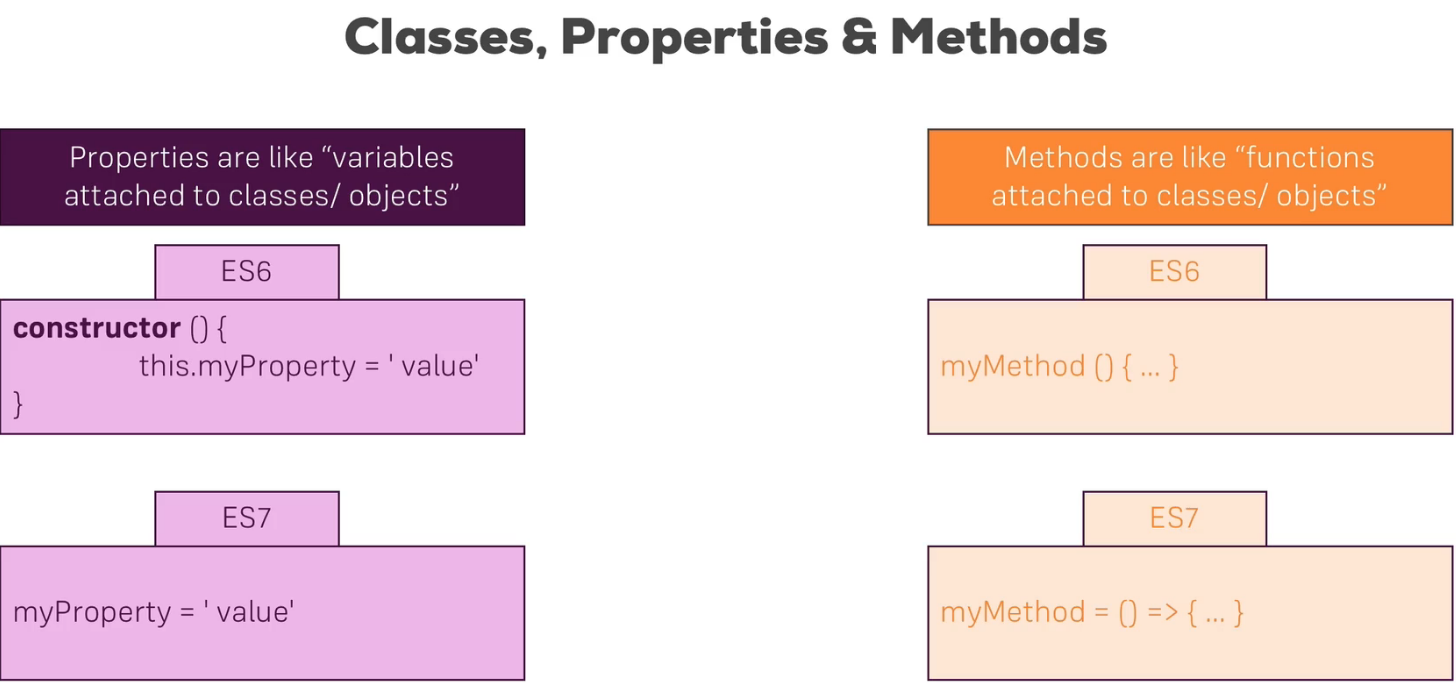
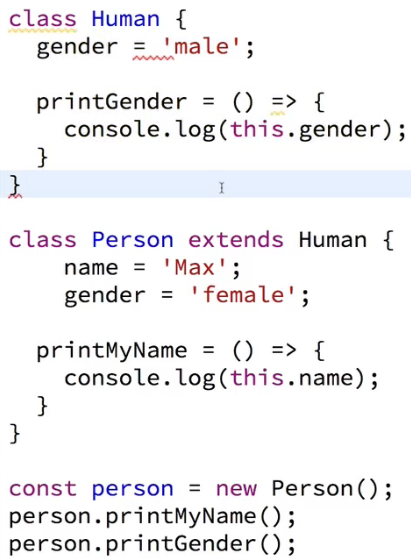
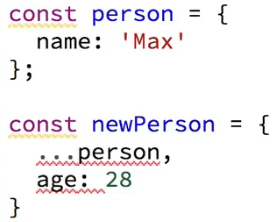
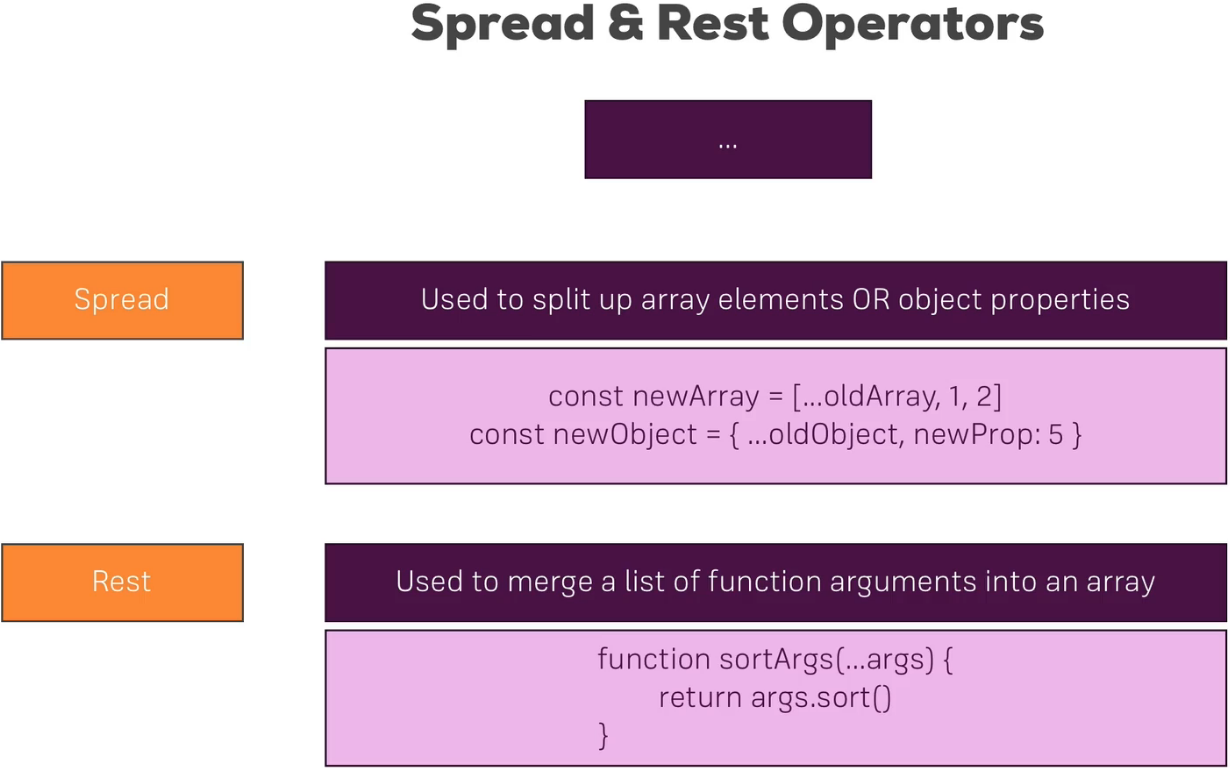
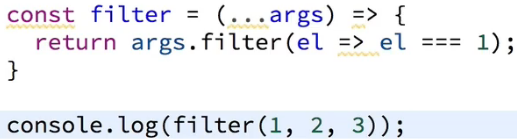
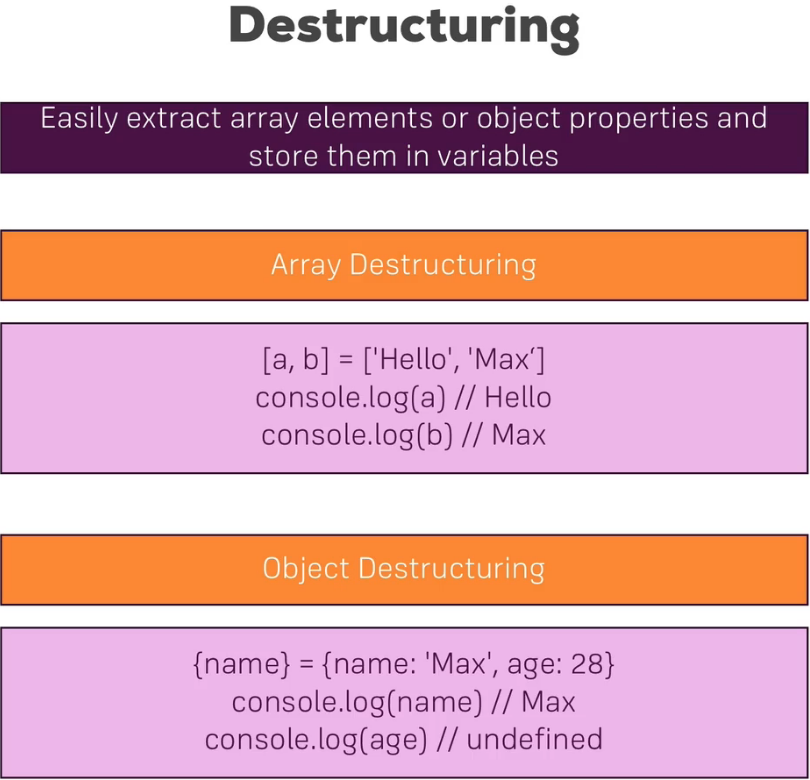
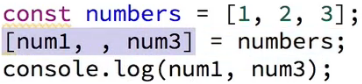
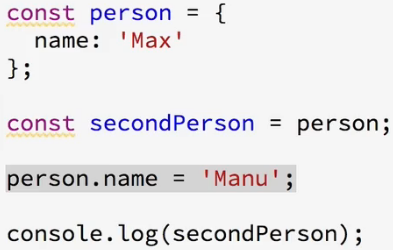
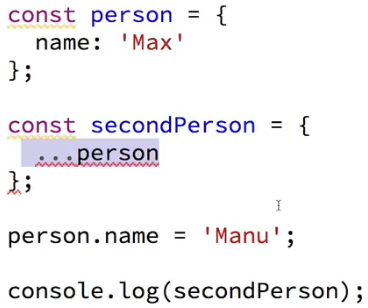
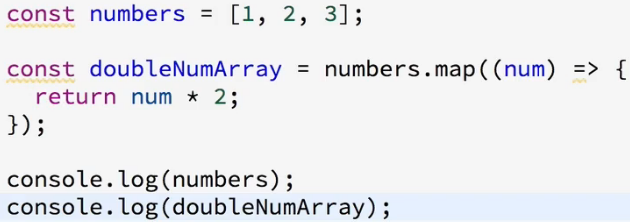
**Section 2 done: 13/13 Refreshing Next Generation JavaScript (Optional)**\* Arrow function => no more issues with `this` keyword (no weird change of context)  
\* export default => if we just import “something” from that file, it’ll always be our default export  
\* named exports => we’re importing stuff by its name  
**Exports & Imports (Modules)**  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
\* bundled in this case would be a JavaScript object which exposes all the constants and whatever you export in the other file, as properties => bundled.baseData, bundled.clean  
\* classes are essentially blueprints for objects, they are very comparable to constructor functions where inheritance is comparable to prototypes  
**Classes**  
  
\* class properties are added using a constructor - this is executed whenever we instantiate the class  
\* method is created with just: name() {}  
\* ES6+ classes - classes are used by React to create its Components (1 of 2 ways of creating them)  


**Classes, Properties & Methods**  
  
\* since you use an arrow function as a property value, you have no problems with the `this` keyword  
\* we still use the this.name when we reach out to the property though  
  
**Spread & Rest Operators**  
  
\* …oldArray without the … would just be an array like [1,2,3] inside the newArray, this way it’s 1,2,3  
\* used to copy arrays or add properties to an object while safely copying that old object  
\* Rest is used in a function - merges the arguments into an array  


**Destructuring**  
  
\* spread takes out all elements, all properties and distributes them in a new array/object  
\* destructuring allows you to pull out single elements or properties and store them in variables  
\* in array destructuring the order defines which property we take  
\* in object destructuring it’s the property name   
  
\* you can skip elements in an array  
> The following are things that are not next-gen but might be easy to forget  
**Reference and Primitive Types**  
\* numbers, strings, booleans - those are primitive types - whenever you reassign or you store a variable in another variable, it will copy the value  
  
\* objects and arrays are reference types though - it does not copy the value of person  
  
\* instead person (the object) is stored in memory and in the const person we store a pointer to that place in memory - and if we then assign secondPerson to person, that pointer will be copied  
\* it prints out ‘Manu’ so for the secondPerson the name also changed, it’s the same for arrays  
\* we will learn how to actually copy them though - for that we can use the SPREAD operator, now we can simply create a new person object here and spread the person properties, this will pull out the properties and the values of the properties from the object and add it to this newly created object and we do create a new one with the {} curly braces, now if we run it, we still print ‘Max’  
  
**Array Functions**  
\* we already saw **filter()**, we also got **sort()**, **map()** and so on, conveniently they create a new array, the old one doesn’t change, these functions always have this function that gets executed on each element  
  
**SUMMARY:**   
<https://www.udemy.com/react-the-complete-guide-incl-redux/learn/v4/t/lecture/8199180?start=0>   
**JS Arrow Functions:**  
<https://www.udemy.com/react-the-complete-guide-incl-redux/learn/v4/t/lecture/8199160?start=0>

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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#No_binding_of_this)).

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

1. function callMe(name) {
2. console.log(name);
3. }

which you could also write as:

1. const callMe = function(name) {
2. console.log(name);
3. }

becomes:

1. const callMe = (name) => {
2. console.log(name);
3. }

**Important:**

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

1. const callMe = () => {
2. console.log('Max!');
3. }

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

1. const callMe = name => {
2. console.log(name);
3. }

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

1. const returnMe = name => name

That's equal to:

1. const returnMe = name => {
2. return name;
3. }

#### ****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:

1. class Person {
2. constructor () {
3. this.name = 'Max';
4. }
5. }
7. const person = new Person();
8. 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:

1. class Person {
2. name = 'Max';
3. }
5. const person = new Person();
6. console.log(person.name); // prints 'Max'

You can also define methods. Either like this:

1. class Person {
2. name = 'Max';
3. printMyName () {
4. console.log(this.name); // this is required to refer to the class!
5. }
6. }
8. const person = new Person();
9. person.printMyName();

Or like this:

1. class Person {
2. name = 'Max';
3. printMyName = () => {
4. console.log(this.name);
5. }
6. }
8. const person = new Person();
9. 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:

1. class Human {
2. species = 'human';
3. }
5. class Person extends Human {
6. name = 'Max';
7. printMyName = () => {
8. console.log(this.name);
9. }
10. }
12. const person = new Person();
13. person.printMyName();
14. 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:

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

Here's the spread operator used on an object:

1. const oldObject = {
2. name: 'Max'
3. };
4. const newObject = {
5. ...oldObject,
6. age: 28
7. };

newObject  would then be

1. {
2. name: 'Max',
3. age: 28
4. }

The spread operator is extremely useful for cloning arrays and objects. Since both are [reference types (and not primitives)](https://youtu.be/9ooYYRLdg_g), 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:

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

And here for an object:

1. const myObj = {
2. name: 'Max',
3. age: 28
4. }
5. const {name} = myObj;
6. console.log(name); // prints 'Max'
7. console.log(age); // prints undefined
8. console.log(myObj); // prints {name: 'Max', age: 28}

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

1. const printName = (personObj) => {
2. console.log(personObj.name);
3. }
4. 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:

1. const printName = ({name}) => {
2. console.log(name);
3. }
4. 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.

Not really next-gen JavaScript, but also important: JavaScript array functions like map() , filter() , reduce()  etc.

You'll see me use them quite a bit since a lot of React concepts rely on working with arrays (in immutable ways).

The following page gives a good overview over the various methods you can use on the array prototype - feel free to click through them and refresh your knowledge as required: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array>

Particularly important in this course are:

* map()  => <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map>
* find()  => <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find>
* findIndex()  => <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex>
* filter()  => <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter>
* reduce()  => <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce?v=b>
* concat()  => <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/concat?v=b>
* slice()  => <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice>
* splice()  => <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice>