b JavaScript

During the course, we have a goal and a need to learn a sufficient amount of JavaScript in addition to web development.

JavaScript has advanced rapidly in the last few years and in this course we use features from the newer versions. The official name of the JavaScript standard is ECMAScript. At this moment, the latest version is the one released in June of 2020 with the name ECMAScript®2020, otherwise known as ES11.

Browsers do not yet support all of JavaScript's newest features. Due to this fact, a lot of code run in browsers has been *transpiled* from a newer version of JavaScript to an older, more compatible version.

Today, the most popular way to do the transpiling is by using Babel. Transpilation is automatically configured in React applications created with create-react-app. We will take a closer look at the configuration of the transpilation in part 7 of this course.

Node.js is a JavaScript runtime environment based on Google's Chrome V8 JavaScript engine and works practically anywhere - from servers to mobile phones. Let's practice writing some JavaScript using Node. It is expected that the version of Node.js installed on your machine is at least version *14.8.0*. The latest versions of Node already understand the latest versions of JavaScript, so the code does not need to be transpiled.

The code is written into files ending with *.js* that are run by issuing the command `node name_of_file.js`

It is also possible to write JavaScript code into the Node.js console, which is opened by typing `node` in the command-line, as well as into the browser's developer tool console. The newest revisions of Chrome handle the newer features of JavaScript pretty well without transpiling the code. Alternatively you can use a tool like JS Bin.

JavaScript is sort of reminiscent, both in name and syntax, to Java. But when it comes to the core mechanism of the language they could not be more different. Coming from a Java background, the behavior of JavaScript can seem a bit alien, especially if one does not make the effort to look up its features.

In certain circles it has also been popular to attempt "simulating" Java features and design patterns in JavaScript. We do not recommend doing this as the languages and respective ecosystems are ultimately very different.

Variables

In JavaScript there are a few ways to go about defining variables:

```
const x = 1
let y = 5

console.log(x, y)   // 1, 5 are printed
y += 10
console.log(x, y)   // 1, 15 are printed
y = 'sometext'
console.log(x, y)   // 1, sometext are printed
x = 4               // causes an error
```

const does not actually define a variable but a *constant* for which the value can no longer be changed. On the other hand let defines a normal variable.

In the example above, we also see that the type of the data assigned to the variable can change during execution. At the start y stores an integer and at the end a string.

It is also possible to define variables in JavaScript using the keyword var. var was, for a long time, the only way to define variables. const and let were only recently added in version ES6. In specific situations, var works in a different way compared to variable definitions in most languages. During this course the use of var is ill-advised and you should stick with using const and let! You can find more on this topic on YouTube - e.g. var, let and const - ES6 JavaScript Features

Arrays

An array and a couple of examples of its use:

```
const t = [1, -1, 3]

t.push(5)

console.log(t.length) // 4 is printed
console.log(t[1])     // -1 is printed

t.forEach(value => {
  console.log(value)  // numbers 1, -1, 3, 5 are printed, each to own line
})
```

Notable in this example is the fact that the contents of the array can be modified even though it is defined as a `const`. Because the array is an object, the variable always points to the same object. However, the content of the array changes as new items are added to it.

One way of iterating through the items of the array is using `forEach` as seen in the example. `forEach` receives a *function* defined using the arrow syntax as a parameter.

```
value => {
  console.log(value)
}
```

forEach calls the function *for each of the items in the array*, always passing the individual item as an argument. The function as the argument of forEach may also receive other arguments.

In the previous example, a new item was added to the array using the method push. When using React, techniques from functional programming are often used. One characteristic of the functional programming paradigm is the use of immutable data structures. In React code, it is preferable to use the method concat, which does not add the item to the array, but creates a new array in which the content of the old array and the new item are both included.

```
const t = [1, -1, 3]

const t2 = t.concat(5)

console.log(t)  // [1, -1, 3] is printed
console.log(t2) // [1, -1, 3, 5] is printed
```

The method call `t.concat(5)` does not add a new item to the old array but returns a new array which, besides containing the items of the old array, also contains the new item.

There are plenty of useful methods defined for arrays. Let's look at a short example of using the map method.

```
const t = [1, 2, 3]

const m1 = t.map(value => value * 2)
console.log(m1)   // [2, 4, 6] is printed
```

Based on the old array, map creates a *new array*, for which the function given as a parameter is used to create the items. In the case of this example the original value is multiplied by two.

Map can also transform the array into something completely different:

```
const m2 = t.map(value => '<li>' + value + '</li>')
console.log(m2)
// [ '<li>1</li>', '<li>2</li>', '<li>3</li>' ] is printed
```

Here an array filled with integer values is transformed into an array containing strings of HTML using the map method. In part 2 of this course, we will see that map is used quite frequently in React.

Individual items of an array are easy to assign to variables with the help of the destructuring assignment.

```
const t = [1, 2, 3, 4, 5]

const [first, second, ...rest] = t

console.log(first, second)  // 1, 2 is printed
console.log(rest)          // [3, 4 ,5] is printed
```

Thanks to the assignment, the variables `first` and `second` will receive the first two integers of the array as their values. The remaining integers are "collected" into an array of their own which is then assigned to the variable `rest`.

Objects

There are a few different ways of defining objects in JavaScript. One very common method is using object literals, which happens by listing its properties within braces:

```
const object1 = {
  name: 'Arto Hellas',
  age: 35,
  education: 'PhD',
}

const object2 = {
  name: 'Full Stack web application development',
  level: 'intermediate studies',
  size: 5,
}

const object3 = {
  name: {
    first: 'Dan',
    last: 'Abramov',
  },
  grades: [2, 3, 5, 3],
  department: 'Stanford University',
}
```

The values of the properties can be of any type, like integers, strings, arrays, objects...

The properties of an object are referenced by using the "dot" notation, or by using brackets:

```
console.log(object1.name)         // Arto Hellas is printed
const fieldName = 'age'
console.log(object1[fieldName])    // 35 is printed
```

You can also add properties to an object on the fly by either using dot notation or brackets:

```
object1.address = 'Helsinki'
object1['secret number'] = 12341
```

The latter of the additions has to be done by using brackets, because when using dot notation, *secret number* is not a valid property name because of the space character.

Naturally, objects in JavaScript can also have methods. However, during this course we do not need to define any objects with methods of their own. This is why they are only discussed briefly during the course.

Objects can also be defined using so-called constructor functions, which results in a mechanism reminiscent of many other programming languages, e.g. Java's classes. Despite this similarity, JavaScript does not have classes in the same sense as object-oriented programming languages. There has been, however, an addition of the *class syntax* starting from version ES6, which in some cases helps structure object-oriented classes.

Functions

We have already become familiar with defining arrow functions. The complete process, without cutting corners, to defining an arrow function is as follows:

```
const sum = (p1, p2) => {
  console.log(p1)
  console.log(p2)
  return p1 + p2
}
```

and the function is called as can be expected:

```
const result = sum(1, 5)
console.log(result)
```

If there is just a single parameter, we can exclude the parentheses from the definition:

```
const square = p => {
  console.log(p)
  return p * p
}
```

If the function only contains a single expression then the braces are not needed. In this case the function only returns the result of its only expression. Now, if we remove console printing, we can further shorten the function definition:

```
const square = p => p * p
```

This form is particularly handy when manipulating arrays - e.g. when using the map method:

```
const t = [1, 2, 3]
const tSquared = t.map(p => p * p)
// tSquared is now [1, 4, 9]
```

The arrow function feature was added to JavaScript only a couple of years ago, with version ES6. Prior to this the only way to define functions was by using the keyword `function`.

There are two ways by which the function can be referenced; one is giving a name in a function declaration.

```
function product(a, b) {
  return a * b
}

const result = product(2, 6)
// result is now 12
```

The other way to define the function is using a function expression. In this case there is no need to give the function a name and the definition may reside among the rest of the code:

```
const average = function(a, b) {
  return (a + b) / 2
}

const result = average(2, 5)
// result is now 3.5
```

During this course we will define all functions using the arrow syntax.

Exercises 1.3.-1.5.

We continue building the application that we started working on in the previous exercises. You can write the code into the same project, since we are only interested in the final state of the submitted application.

Pro-tip: you may run into issues when it comes to the structure of the *props* that components receive. A good way to make things more clear is by printing the props to the console, e.g. as follows:

```
const Header = (props) => {
  console.log(props)
  return <h1>{props.course}</h1>
}
```

1.3: course information step3

Let's move forward to using objects in our application. Modify the variable definitions of the *App* component as follows and also refactor the application so that it still works:

```
const App = () => {
  const course = 'Half Stack application development'
  const part1 = {
    name: 'Fundamentals of React',
    exercises: 10
  }
  const part2 = {
    name: 'Using props to pass data',
    exercises: 7
  }
  const part3 = {
    name: 'State of a component',
    exercises: 14
  }
  return (
    <div>
      ...
    </div>
  )
}
```

1.4: course information step4

And then place the objects into an array. Modify the variable definitions of *App* into the following form and modify the other parts of the application accordingly:

```
const App = () => {
  const course = 'Half Stack application development'
  const parts = [
    {
      name: 'Fundamentals of React',
      exercises: 10
    },
    {
      name: 'Using props to pass data',
      exercises: 7
    },
    {
      name: 'State of a component',
      exercises: 14
    }
  ]

  return (
    <div>
      ...
    </div>
  )
}
```

NB at this point *you can assume that there are always three items*, so there is no need to go through the arrays using loops. We will come back to the topic of rendering components based on items in arrays with a more thorough exploration in the next part of the course.

However, do not pass different objects as separate props from the *App* component to the components *Content* and *Total*. Instead, pass them directly as an array:

```
const App = () => {
  // const definitions

  return (
    <div>
      <Header course={course} />
      <Content parts={parts} />
      <Total parts={parts} />
    </div>
  )
}
```

1.5: course information step5

Let's take the changes one step further. Change the course and its parts into a single JavaScript object. Fix everything that breaks.

```
const App = () => {
  const course = {
    name: 'Half Stack application development',
    parts: [
      {
        name: 'Fundamentals of React',
        exercises: 10
      },
      {
        name: 'Using props to pass data',
        exercises: 7
      },
      {
        name: 'State of a component',
        exercises: 14
      }
    ]
  }

  return (
    <div>
      ...
    </div>
  )
}
```

Object methods and "this"

Due to the fact that during this course we are using a version of React containing React Hooks we have no need for defining objects with methods. The contents of this chapter are not relevant to the course but are certainly in many ways good to know. In particular when using older versions of React one must understand the topics of this chapter.

Arrow functions and functions defined using the `function` keyword vary substantially when it comes to how they behave with respect to the keyword `this`, which refers to the object itself.

We can assign methods to an object by defining properties that are functions:

```
const arto = {
  name: 'Arto Hellas',
  age: 35,
  education: 'PhD',
  greet: function() {
    console.log('hello, my name is ' + this.name)
  },
}

arto.greet()  // "hello, my name is Arto Hellas" gets printed
```

Methods can be assigned to objects even after the creation of the object:

```
const arto = {
  name: 'Arto Hellas',
  age: 35,
  education: 'PhD',
  greet: function() {
    console.log('hello, my name is ' + this.name)
  },
}

arto.growOlder = function() {
  this.age += 1
}

console.log(arto.age)   // 35 is printed
arto.growOlder()
console.log(arto.age)   // 36 is printed
```

Let's slightly modify the object:

```
const arto = {
  name: 'Arto Hellas',
  age: 35,
  education: 'PhD',
  greet: function() {
    console.log('hello, my name is ' + this.name)
  },
  doAddition: function(a, b) {
    console.log(a + b)
  },
}
```

```
arto.doAddition(1, 4)        // 5 is printed

const referenceToAddition = arto.doAddition
referenceToAddition(10, 15)   // 25 is printed
```

Now the object has the method `doAddition` which calculates the sum of numbers given to it as parameters. The method is called in the usual way, using the object `arto.doAddition(1, 4)` or by storing a *method reference* in a variable and calling the method through the variable: `referenceToAddition(10, 15)`.

If we try to do the same with the method `greet` we run into an issue:

```
arto.greet()       // "hello, my name is Arto Hellas" gets printed

const referenceToGreet = arto.greet
referenceToGreet() // prints "hello, my name is undefined"
```

When calling the method through a reference, the method loses knowledge of what was the original `this`. Contrary to other languages, in JavaScript the value of this is defined based on *how the method is called*. When calling the method through a reference the value of `this` becomes the so-called global object and the end result is often not what the software developer had originally intended.

Losing track of `this` when writing JavaScript code brings forth a few potential issues. Situations often arise where React or Node (or more specifically the JavaScript engine of the web browser) needs to call some method in an object that the developer has defined. However, in this course we avoid these issues by using the "this-less" JavaScript.

One situation leading to the "disappearance" of `this` arises when we set a timeout to call the `greet` function on the `arto` object, using the setTimeout function.

```
const arto = {
  name: 'Arto Hellas',
  greet: function() {
    console.log('hello, my name is ' + this.name)
  },
}

setTimeout(arto.greet, 1000)
```

As mentioned, the value of `this` in JavaScript is defined based on how the method is being called. When `setTimeout` is calling the method, it is the JavaScript engine that actually calls the method and, at that point, `this` refers to the global object.

There are several mechanisms by which the original `this` can be preserved. One of these is using a method called bind:

```
setTimeout(arto.greet.bind(arto), 1000)
```

Calling `arto.greet.bind(arto)` creates a new function where `this` is bound to point to Arto, independent of where and how the method is being called.

Using arrow functions it is possible to solve some of the problems related to `this` . They should not, however, be used as methods for objects because then `this` does not work at all. We will come back later to the behavior of `this` in relation to arrow functions.

If you want to gain a better understanding of how `this` works in JavaScript, the Internet is full of material about the topic, e.g. the screencast series Understand JavaScript's this Keyword in Depth by egghead.io is highly recommended!

Classes

As mentioned previously, there is no class mechanism like the ones in object-oriented programming languages. There are, however, features in JavaScript which make "simulating" object-oriented classes possible.

Let's take a quick look at the *class syntax* that was introduced into JavaScript with ES6, which substantially simplifies the definition of classes (or class-like things) in JavaScript.

In the following example we define a "class" called Person and two Person objects:

```
class Person {
  constructor(name, age) {
    this.name = name
    this.age = age
  }
  greet() {
    console.log('hello, my name is ' + this.name)
  }
}

const adam = new Person('Adam Ondra', 35)
adam.greet()

const janja = new Person('Janja Garnbret', 22)
janja.greet()
```

When it comes to syntax, the classes and the objects created from them are very reminiscent of Java classes and objects. Their behavior is also quite similar to Java objects. At the core they are still objects based on JavaScript's prototypal inheritance. The type of both objects is actually `Object`, since JavaScript essentially only defines the types Boolean, Null, Undefined, Number, String, Symbol, BigInt, and Object.

The introduction of the class syntax was a controversial addition. Check out Not Awesome: ES6 Classes or Is “Class” In ES6 The New “Bad” Part? for more details.

The ES6 class syntax is used a lot in "old" React and also in Node.js, hence an understanding of it is beneficial even in this course. However, since we are using the new Hooks feature of React throughout this course, we have no concrete use for JavaScript's class syntax.

JavaScript materials

There exist both good and poor guides for JavaScript on the Internet. Most of the links on this page relating to JavaScript feature reference Mozilla's JavaScript Guide.

It is highly recommended to immediately read A re-introduction to JavaScript (JS tutorial) on Mozilla's website.

If you wish to get to know JavaScript deeply there is a great free book series on the Internet called You-Dont-Know-JS.

Another great resource for learning JavaScript is javascript.info.

egghead.io has plenty of quality screencasts on JavaScript, React, and other interesting topics. Unfortunately, some of the material is behind a paywall.

Propose changes to material

Part 1a
Previous part

Part 1c
Next part

About course

Course contents

FAQ

Partners

Challenge

UNIVERSITY OF HELSINKI

HOUSTON