

```
{() => fs}
```



a Introduction to React

We will now start getting familiar with probably the most important topic of this course, namely the React library. Let's start by making a simple React application as well as getting to know the core concepts of React.

The easiest way to get started by far is by using a tool called Vite.

Let's create an application called *part1*, navigate to its directory and install the libraries:

```
# npm 6.x (outdated, but still used by some):  
npm create vite@latest part1 --template react
```

copy

```
# npm 7+, extra double-dash is needed:  
npm create vite@latest part1 -- --template react
```

```
cd part1  
npm install
```

copy

The application is started as follows

```
npm run dev
```

copy

The console says that the application has started on localhost port 5173, i.e. the address http://localhost:5173/:

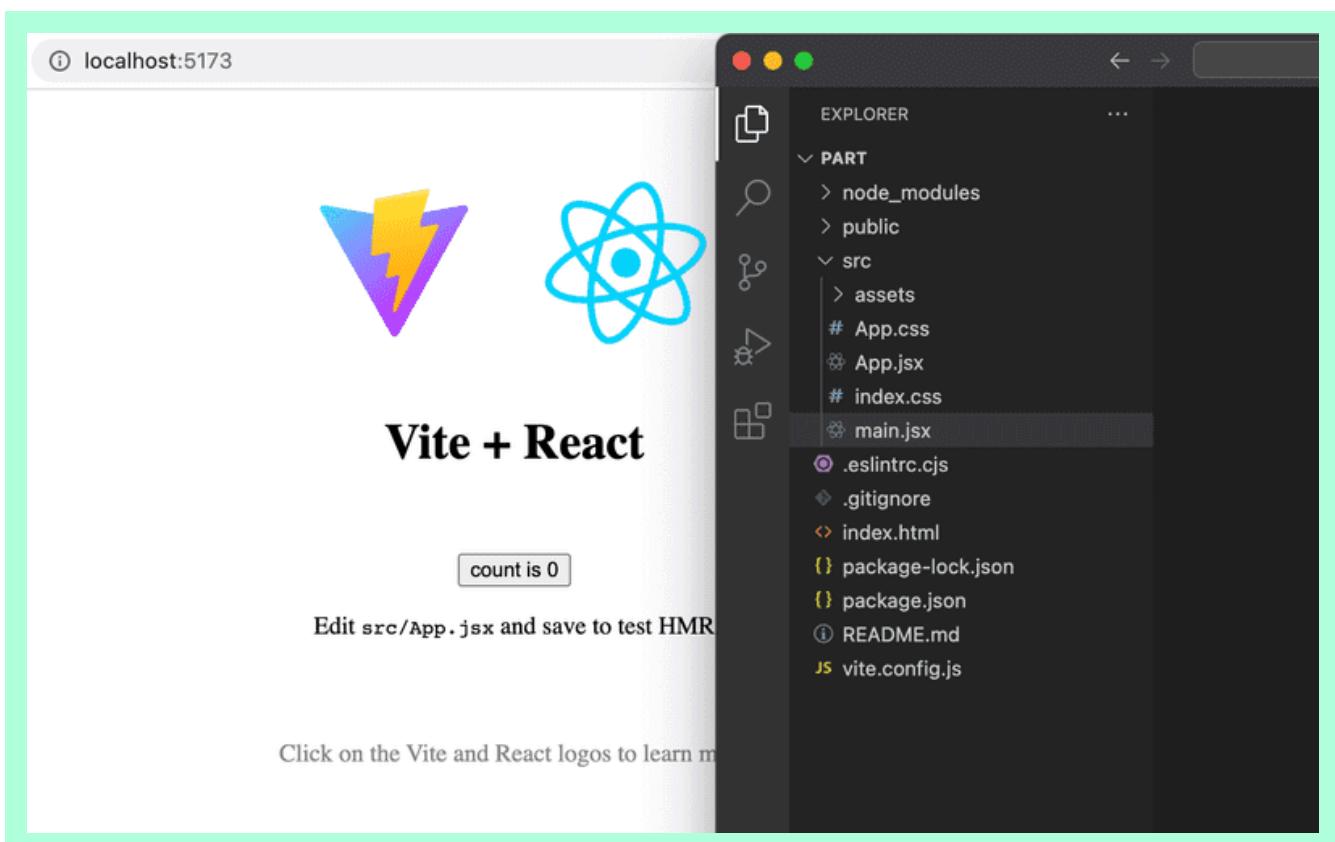
```
npm run dev

VITE v4.4.9 ready in 345 ms

→ Local: http://localhost:5173/
→ Network: use --host to expose
→ press h to show help
```

Vite starts the application by default on port 5173. If it is not free, Vite uses the next free port number.

Open the browser and a text editor so that you can view the code as well as the webpage at the same time on the screen:



The code of the application resides in the `src` folder. Let's simplify the default code such that the contents of the file `main.jsx` looks like this:

```
import ReactDOM from 'react-dom/client'

import App from './App'

ReactDOM.createRoot(document.getElementById('root')).render(<App />)
```

and file *App.jsx* looks like this

```
const App = () => {
  return (
    <div>
      <p>Hello world</p>
    </div>
  )
}

export default App
```

copy

The files *App.css* and *index.css*, and the directory *assets* may be deleted as they are not needed in our application right now.

create-react-app

Instead of Vite you can also use the older generation tool [create-react-app](#) in the course to set up the applications. The most visible difference to Vite is the name of the application startup file, which is *index.js*.

The way to start the application is also different in CRA, it is started with a command

`npm start`

copy

in contrast to Vite's

`npm run dev`

copy

Component

The file *App.jsx* now defines a [React component](#) with the name *App*. The command on the final line of file *main.jsx*

`ReactDOM.createRoot(document.getElementById('root')).render(<App />)`

copy

renders its contents into the *div*-element, defined in the file *index.html*, having the *id* value 'root'.

By default, the file *index.html* doesn't contain any HTML markup that is visible to us in the browser:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

[copy](#)

You can try adding there some HTML to the file. However, when using React, all content that needs to be rendered is usually defined as React components.

Let's take a closer look at the code defining the component:

```
const App = () => (
  <div>
    <p>Hello world</p>
  </div>
)
```

[copy](#)

As you probably guessed, the component will be rendered as a *div*-tag, which wraps a *p*-tag containing the text *Hello world*.

Technically the component is defined as a JavaScript function. The following is a function (which does not receive any parameters):

```
() => (
  <div>
    <p>Hello world</p>
  </div>
)
```

[copy](#)

The function is then assigned to a constant variable *App*:

```
const App = ...
```

[copy](#)

There are a few ways to define functions in JavaScript. Here we will use arrow functions, which are described in a newer version of JavaScript known as ECMAScript 6, also called ES6.

Because the function consists of only a single expression we have used a shorthand, which represents this piece of code:

```
const App = () => {
  return (
    <div>
      <p>Hello world</p>
    </div>
  )
}
```

copy

In other words, the function returns the value of the expression.

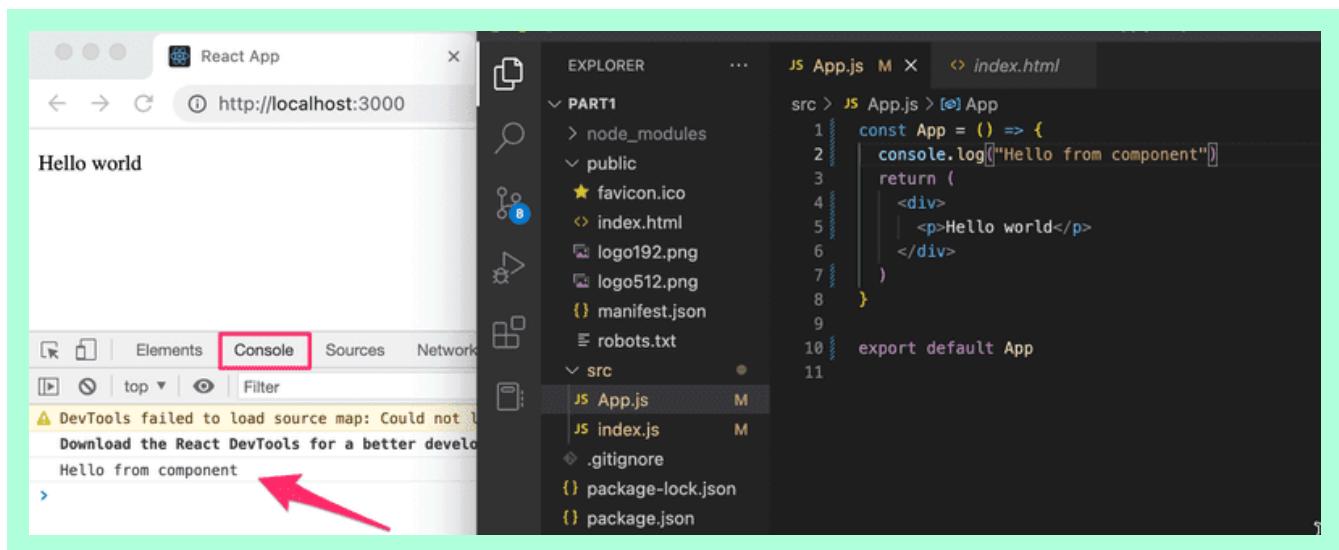
The function defining the component may contain any kind of JavaScript code. Modify your component to be as follows:

```
const App = () => {
  console.log('Hello from component')
  return (
    <div>
      <p>Hello world</p>
    </div>
  )
}

export default App
```

copy

and observe what happens in the browser console



The first rule of frontend web development:

keep the console open all the time

Let us repeat this together: *I promise to keep the console open all the time* during this course, and for the rest of my life when I'm doing web development.

It is also possible to render dynamic content inside of a component.

Modify the component as follows:

```
const App = () => {
  const now = new Date()
  const a = 10
  const b = 20
  console.log(now, a+b)

  return (
    <div>
      <p>Hello world, it is {now.toString()}</p>
      <p>
        {a} plus {b} is {a + b}
      </p>
    </div>
  )
}
```

copy

Any JavaScript code within the curly braces is evaluated and the result of this evaluation is embedded into the defined place in the HTML produced by the component.

Note that you should not remove the line at the bottom of the component

```
export default App
```

copy

The export is not shown in most of the examples of the course material. Without the export the component and the whole app breaks down.

Did you remember your promise to keep the console open? What was printed out there?

JSX

It seems like React components are returning HTML markup. However, this is not the case. The layout of React components is mostly written using JSX. Although JSX looks like HTML, we are dealing with a way to write JavaScript. Under the hood, JSX returned by React components is compiled into JavaScript.

After compiling, our application looks like this:

```
const App = () => {
  const now = new Date()
  const a = 10
  const b = 20
  return React.createElement(
    'div',
    null,
    React.createElement(
      'p', null, 'Hello world, it is ', now.toString()
    ),
    React.createElement(
      'p', null, a, ' plus ', b, ' is ', a + b
    )
  )
}
```

[copy](#)

The compilation is handled by Babel. Projects created with `create-react-app` or `vite` are configured to compile automatically. We will learn more about this topic in part 7 of this course.

It is also possible to write React as "pure JavaScript" without using JSX. Although, nobody with a sound mind would do so.

In practice, JSX is much like HTML with the distinction that with JSX you can easily embed dynamic content by writing appropriate JavaScript within curly braces. The idea of JSX is quite similar to many templating languages, such as Thymeleaf used along with Java Spring, which are used on servers.

JSX is "XML-like", which means that every tag needs to be closed. For example, a newline is an empty element, which in HTML can be written as follows:

```
<br>
```

[copy](#)

but when writing JSX, the tag needs to be closed:

```
<br />
```

[copy](#)

Multiple components

Let's modify the file `App.jsx` as follows:

```
const Hello = () => {
  return (
    <div>
      <p>Hello world</p>
    </div>
  )
}
```

[copy](#)

```

)
}

const App = () => {
  return (
    <div>
      <h1>Greetings</h1>
      <Hello />
    </div>
  )
}

```

We have defined a new component *Hello* and used it inside the component *App*. Naturally, a component can be used multiple times:

```

const App = () => {
  return (
    <div>
      <h1>Greetings</h1>
      <Hello />
      <Hello />
      <Hello />
    </div>
  )
}

```

copy

NB: `export` at the bottom is left out in these *examples*, now and in the future. It is still needed for the code to work

Writing components with React is easy, and by combining components, even a more complex application can be kept fairly maintainable. Indeed, a core philosophy of React is composing applications from many specialized reusable components.

Another strong convention is the idea of a *root component* called *App* at the top of the component tree of the application. Nevertheless, as we will learn in part 6, there are situations where the component *App* is not exactly the root, but is wrapped within an appropriate utility component.

props: passing data to components

It is possible to pass data to components using so-called props.

Let's modify the component *Hello* as follows:

```

const Hello = (props) => {
  return (
    <div>
      <p>Hello {props.name}</p>
    </div>
  )
}

```

copy

```
)  
}
```

Now the function defining the component has a parameter `props`. As an argument, the parameter receives an object, which has fields corresponding to all the "props" the user of the component defines.

The props are defined as follows:

```
const App = () => {  
  return (  
    <div>  
      <h1>Greetings</h1>  
      <Hello name='George' />  
      <Hello name='Daisy' />  
    </div>  
  )  
}
```

copy

There can be an arbitrary number of props and their values can be "hard-coded" strings or the results of JavaScript expressions. If the value of the prop is achieved using JavaScript it must be wrapped with curly braces.

Let's modify the code so that the component `Hello` uses two props:

```
const Hello = (props) => {  
  console.log(props)  
  return (  
    <div>  
      <p>  
        Hello {props.name}, you are {props.age} years old  
      </p>  
    </div>  
  )  
}
```

copy

```
const App = () => {  
  const name = 'Peter'  
  const age = 10  
  
  return (  
    <div>  
      <h1>Greetings</h1>  
      <Hello name='Maya' age={26 + 10} />  
      <Hello name={name} age={age} />  
    </div>  
  )  
}
```

The props sent by the component `App` are the values of the variables, the result of the evaluation of the sum expression and a regular string.

Component `Hello` also logs the value of the object props to the console.

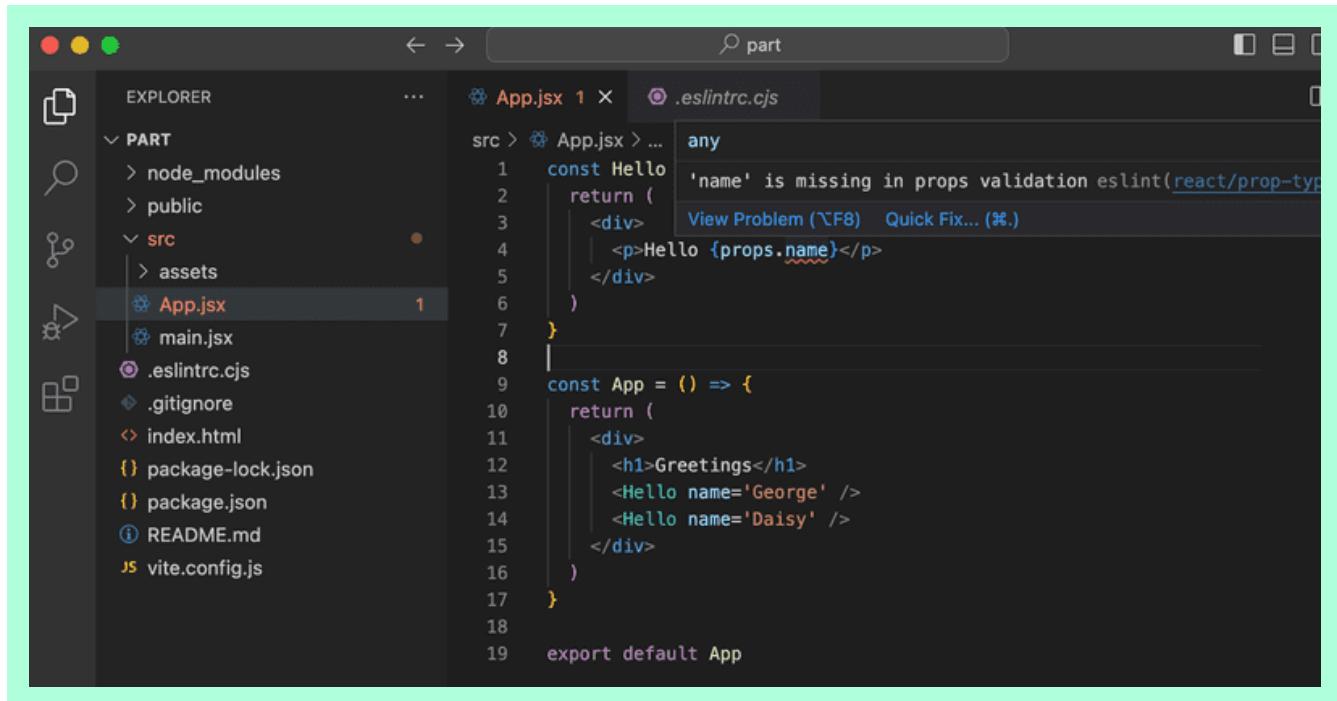
I really hope your console was open. If it was not, remember what you promised:

I promise to keep the console open all the time during this course, and for the rest of my life when I'm doing web development

Software development is hard. It gets even harder if one is not using all the possible available tools such as the web-console and debug printing with `console.log`. Professionals use both *all the time* and there is no single reason why a beginner should not adopt the use of these wonderful helper methods that will make his life so much easier.

Possible error message

Depending on the editor you are using, you may receive the following error message at this point:



The screenshot shows a code editor with the file `App.jsx` open. The code contains a `Hello` component and an `App` component. An ESLint warning is displayed in the editor's interface, indicating that the prop `'name'` is missing in the `Hello` component's props validation. The warning is linked to the `react/prop-types` rule in the ESLint configuration file `.eslintrc.cjs`.

```

const Hello = ({ name }) => {
  return (
    <div>
      <p>Hello {props.name}</p>
    </div>
  )
}

const App = () => {
  return (
    <div>
      <h1>Greetings</h1>
      <Hello name='George' />
      <Hello name='Daisy' />
    </div>
  )
}

export default App
  
```

It's not an actual error, but a warning caused by the `ESLint` tool. You can silence the warning `react/prop-types` by adding to the file `.eslintrc.cjs` the next line

```

module.exports = {
  root: true,
  env: { browser: true, es2020: true },
  extends: [
    'eslint:recommended',
    'plugin:react/recommended',
    'plugin:react/jsx-runtime',
  ],
}
  
```

copy

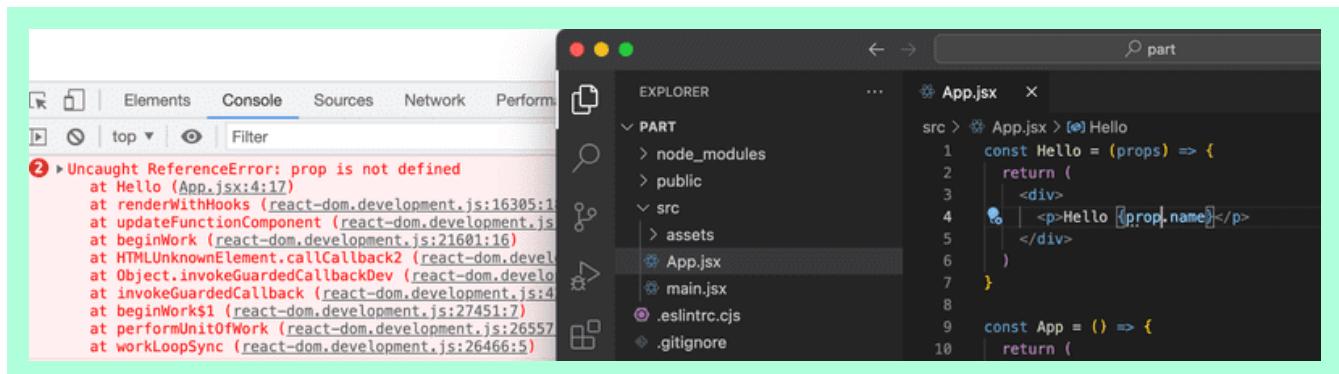
```
'plugin:react-hooks/recommended',
],
ignorePatterns: ['dist', '.eslintrc.cjs'],
parserOptions: { ecmaVersion: 'latest', sourceType: 'module' },
settings: { react: { version: '18.2' } },
plugins: ['react-refresh'],
rules: {
  'react-refresh/only-export-components': [
    'warn',
    { allowConstantExport: true },
  ],
  'react/prop-types': 0
},
}
```

We will get to know ESLint in more detail [in part 3.](#)

Some notes

React has been configured to generate quite clear error messages. Despite this, you should, at least in the beginning, advance in **very small steps** and make sure that every change works as desired.

The console should always be open. If the browser reports errors, it is not advisable to continue writing more code, hoping for miracles. You should instead try to understand the cause of the error and, for example, go back to the previous working state:



As we already mentioned, when programming with React, it is possible and worthwhile to write `console.log()` commands (which print to the console) within your code.

Also, keep in mind that **First letter of React component names must be capitalized**. If you try defining a component as follows:

```
const footer = () => {
  return (
    <div>
      greeting app created by <a href='https://github.com/mluukkai'>mluukkai</a>
    </div>
  )
}
```

copy

```
)  
}
```

and use it like this

```
const App = () => {  
  return (  
    <div>  
      <h1>Greetings</h1>  
      <Hello name='Maya' age={26 + 10} />  
      <Footer />  
    </div>  
  )  
}
```

[copy](#)

the page is not going to display the content defined within the `Footer` component, and instead React only creates an empty `Footer` element, i.e. the built-in HTML element instead of the custom React element of the same name. If you change the first letter of the component name to a capital letter, then React creates a `div`-element defined in the `Footer` component, which is rendered on the page.

Note that the content of a React component (usually) needs to contain **one root element**. If we, for example, try to define the component `App` without the outermost `div`-element:

```
const App = () => {  
  return (  
    <h1>Greetings</h1>  
    <Hello name='Maya' age={26 + 10} />  
    <Footer />  
  )  
}
```

[copy](#)

the result is an error message.

```
[plugin:vite:react-babel] /Users/mluukkai/opetus/hy-fs/vite/part1/part/src/App.jsx  
Adjacent JSX elements must be wrapped in an enclosing tag. Did you want a JSX  
fragment <>...</>? (13:6)  
16 | )  
  
/Users/mluukkai/opetus/hy-fs/vite/part1/part/src/App.jsx:13:6  
11 |  
12 |       <h1>Greetings</h1>  
13 |       <Hello name='George' />  
|  
14 |       ^  
15 |       <Hello name='Daisy' />
```

Using a root element is not the only working option. An *array* of components is also a valid solution:

```
const App = () => {
  return [
    <h1>Greetings</h1>,
    <Hello name='Maya' age={26 + 10} />,
    <Footer />
  ]
}
```

copy

However, when defining the root component of the application this is not a particularly wise thing to do, and it makes the code look a bit ugly.

Because the root element is stipulated, we have "extra" div elements in the DOM tree. This can be avoided by using fragments, i.e. by wrapping the elements to be returned by the component with an empty element:

```
const App = () => {
  const name = 'Peter'
  const age = 10

  return (
    <>
      <h1>Greetings</h1>
      <Hello name='Maya' age={26 + 10} />
      <Hello name={name} age={age} />
      <Footer />
    </>
  )
}
```

copy

It now compiles successfully, and the DOM generated by React no longer contains the extra div element.

Do not render objects

Consider an application that prints the names and ages of our friends on the screen:

```
const App = () => {
  const friends = [
    { name: 'Peter', age: 4 },
    { name: 'Maya', age: 10 },
  ]

  return (
    <div>
      <p>{friends[0]}</p>
    </div>
  )
}
```

copy

```

    <p>{friends[1]}</p>
  </div>
)
}

export default App

```

However, nothing appears on the screen. I've been trying to find a problem in the code for 15 minutes, but I can't figure out where the problem could be.

I finally remember the promise we made

I promise to keep the console open all the time during this course, and for the rest of my life when I'm doing web development

The console screams in red:

```

2 ► Uncaught Error: Objects are not valid as a React child (found: object with keys {name, age}). If you meant to render a collection of children, use an array instead.
  at throwOnInvalidObjectType (react-dom.development.js:14887:1)
  at reconcileChildFibers (react-dom.development.js:15828:1)
  at reconcileChildren (react-dom.development.js:19167:1)
  at updateHostComponent (react-dom.development.js:19924:1)
  at beginWork (react-dom.development.js:21618:1)
  at HTMLUnknownElement.callCallback (react-dom.development.js:4164:1)
  at Object.invokeGuardedCallbackDev (react-dom.development.js:4213:1)
  at invokeGuardedCallback (react-dom.development.js:4277:1)
  at beginWork$1 (react-dom.development.js:27451:1)
  at performUnitOfWork (react-dom.development.js:26557:1)

```

The core of the problem is *Objects are not valid as a React child*, i.e. the application tries to render *objects* and it fails again.

The code tries to render the information of one friend as follows

<p>{friends[0]}</p>

copy

and this causes a problem because the item to be rendered in the braces is an object.

{ name: 'Peter', age: 4 }

copy

In React, the individual things rendered in braces must be primitive values, such as numbers or strings.

The fix is as follows

```

const App = () => {
  const friends = [
    { name: 'Peter', age: 4 },
    { name: 'Maya', age: 10 },
  ]
}

```

copy

```

return (
  <div>
    <p>{friends[0].name} {friends[0].age}</p>
    <p>{friends[1].name} {friends[1].age}</p>
  </div>
)
}

export default App

```

So now the friend's name is rendered separately inside the curly braces

{friends[0].name}

[copy](#)

and age

{friends[0].age}

[copy](#)

After correcting the error, you should clear the console error messages by pressing  and then reload the page content and make sure that no error messages are displayed.

A small additional note to the previous one. React also allows arrays to be rendered *if* the array contains values that are eligible for rendering (such as numbers or strings). So the following program would work, although the result might not be what we want:

```

const App = () => {
  const friends = [ 'Peter', 'Maya' ]

  return (
    <div>
      <p>{friends}</p>
    </div>
  )
}

```

[copy](#)

In this part, it is not even worth trying to use the direct rendering of the tables, we will come back to it in the next part.

Exercises 1.1.-1.2.

The exercises are submitted via GitHub, and by marking the exercises as done in the "my submissions" tab of the submission application.

The exercises are submitted **one part at a time**. When you have submitted the exercises for a part of the course you can no longer submit undone exercises for the same part.

Note that in this part, there are more exercises besides those found below. *Do not submit your work* until you have completed all of the exercises you want to submit for the part.

You may submit all the exercises of this course into the same repository, or use multiple repositories. If you submit exercises of different parts into the same repository, please use a sensible naming scheme for the directories.

One very functional file structure for the submission repository is as follows:

```
part0
part1
  courseinfo
  unicafe
  anecdotes
part2
  phonebook
  countries
```

copy

See this example submission repository!

For each part of the course, there is a directory, which further branches into directories containing a series of exercises, like "unicafe" for part 1.

Most of the exercises of the course build a larger application, eg. courseinfo, unicafe and anecdotes in this part, bit by bit. It is enough to submit the completed application. You can make a commit after each exercise, but that is not compulsory. For example the course info app is built in exercises 1.1.-1.5. It is just the end result after 1.5 that you need to submit!

For each web application for a series of exercises, it is recommended to submit all files relating to that application, except for the directory *node_modules*.

1.1: Course Information, step 1

The application that we will start working on in this exercise will be further developed in a few of the following exercises. In this and other upcoming exercise sets in this course, it is enough to only submit the final state of the application. If desired, you may also create a commit for each exercise of the series, but this is entirely optional.

Use Vite to initialize a new application. Modify *main.jsx* to match the following

```
import ReactDOM from 'react-dom/client'
```

copy

```
import App from './App'

ReactDOM.createRoot(document.getElementById('root')).render(<App />)
```

and *App.jsx* to match the following

```
const App = () => {
  const course = 'Half Stack application development'
  const part1 = 'Fundamentals of React'
  const exercises1 = 10
  const part2 = 'Using props to pass data'
  const exercises2 = 7
  const part3 = 'State of a component'
  const exercises3 = 14

  return (
    <div>
      <h1>{course}</h1>
      <p>
        {part1} {exercises1}
      </p>
      <p>
        {part2} {exercises2}
      </p>
      <p>
        {part3} {exercises3}
      </p>
      <p>Number of exercises {exercises1 + exercises2 + exercises3}</p>
    </div>
  )
}

export default App
```

copy

and remove the extra files *App.css* and *index.css*, also remove the directory assets.

Unfortunately, the entire application is in the same component. Refactor the code so that it consists of three new components: *Header*, *Content*, and *Total*. All data still resides in the *App* component, which passes the necessary data to each component using *props*. *Header* takes care of rendering the name of the course, *Content* renders the parts and their number of exercises and *Total* renders the total number of exercises.

Define the new components in the file *App.jsx*.

The *App* component's body will approximately be as follows:

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

  return (
```

copy

```

<div>
  <Header course={course} />
  <Content ... />
  <Total ... />
</div>
)
}

```

WARNING Don't try to program all the components concurrently, because that will almost certainly break down the whole app. Proceed in small steps, first make e.g. the component *Header* and only when it works for sure, you could proceed to the next component.

Careful, small-step progress may seem slow, but it is actually *by far the fastest* way to progress. Famous software developer Robert "Uncle Bob" Martin has stated

"The only way to go fast, is to go well"

that is, according to Martin, careful progress with small steps is even the only way to be fast.

1.2: Course Information, step 2

Refactor the *Content* component so that it does not render any names of parts or their number of exercises by itself. Instead, it only renders three *Part* components of which each renders the name and number of exercises of one part.

```

const Content = ... {
  return (
    <div>
      <Part .../>
      <Part .../>
      <Part .../>
    </div>
  )
}

```

copy

Our application passes on information in quite a primitive way at the moment, since it is based on individual variables. We shall fix that in part 2, but before that, let's go to part1b to learn about JavaScript.

[Propose changes to material](#)

Part 0

[Previous part](#)

Part 1b

[Next part](#)

About course

Course contents

FAQ

Partners

Challenge



HOUSTON

```
{() => fs}
```



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 2023 with the name ECMAScript®2023, otherwise known as ES14.

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 transpiling is by using Babel. Transpilation is automatically configured in React applications created with vite. 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. 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
```

copy

const does not 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 variable's data type can change during execution. At the start, y stores an integer; at the end, it stores 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 - see [JavaScript Variables - Should You Use let, var or const? on Medium](#) or [Keyword: var vs. let on JS Tips](#) for more information. 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
```

copy

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

copy

`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 creates a new array with the added item. This ensures the original array remains unchanged.

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

const t2 = t.concat(5) // creates new array

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

copy

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
```

copy

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
```

copy

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
```

copy

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',
  },
}
```

copy

```
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
```

[copy](#)

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
```

[copy](#)

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, the 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, of defining an arrow function is as follows:

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

[copy](#)

and the function is called as can be expected:

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

[copy](#)

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

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

[copy](#)

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
```

[copy](#)

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]
```

[copy](#)

The arrow function feature was added to JavaScript only a couple of years ago, with version [ES6](#). Before this, the only way to define functions was by using the keyword [function](#).

There are two ways to reference the function; one is giving a name in a [function declaration](#).

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

[copy](#)

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

The other way to define the function is by 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
```

[copy](#)

}

```
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>
}
```

copy

If and *when* you encounter an error message

Objects are not valid as a React child

keep in mind the things told here.

1.3: Course Information step 3

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
}
```

copy

```

        }
      }

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

```

1.4: Course Information step 4

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>
)
}

```

copy

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
}

```

copy

```

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

```

1.5: Course Information step 5

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>
  )
}

```

copy

Object methods and "this"

Because this course uses a version of React containing React Hooks we do not need to define 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)
  },
}
```

[copy](#)

```
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)
  },
}
```

[copy](#)

```
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)
  },
}
```

[copy](#)

```
}

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
```

copy

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

When calling the method through a reference, the method loses knowledge of what the original `this` was. 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 "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)
```

copy

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(arvo), 1000)
```

[copy](#)

Calling `arto.greet.bind(arvo)` 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 in JavaScript like the ones in object-oriented programming languages. There are, however, features to 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', 29)
adam.greet()

const janja = new Person('Janja Garnbret', 23)
janja.greet()
```

[copy](#)

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?](#) on Medium 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 features 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](#).

The free and highly engaging book [Eloquent JavaScript](#) takes you from the basics to interesting stuff quickly. It is a mixture of theory projects and exercises and covers general programming theory as well as the JavaScript language.

[Namaste 🌟 JavaScript](#) is another great and highly recommended free JavaScript tutorial in order to understand how JS works under the hood. Namaste JavaScript is a pure in-depth JavaScript course released for free on YouTube. It will cover the core concepts of JavaScript in detail and everything about how JS works behind the scenes inside the JavaScript engine.

[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](#)

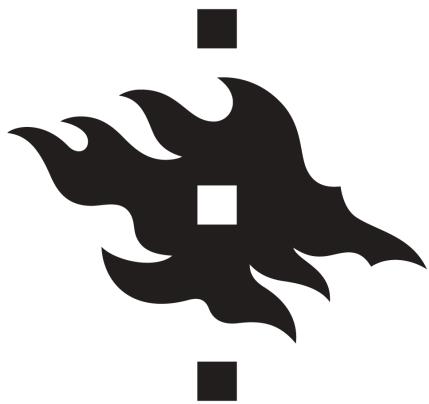
[About course](#)

[Course contents](#)

[FAQ](#)

Partners

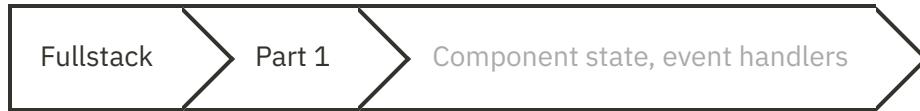
Challenge



UNIVERSITY OF HELSINKI

HOUSTON

```
{() => fs}
```



c Component state, event handlers

Let's go back to working with React.

We start with a new example:

```
const Hello = (props) => {
  return (
    <div>
      <p>
        Hello {props.name}, you are {props.age} years old
      </p>
    </div>
  )
}

const App = () => {
  const name = 'Peter'
  const age = 10

  return (
    <div>
      <h1>Greetings</h1>
      <Hello name="Maya" age={26 + 10} />
      <Hello name={name} age={age} />
    </div>
  )
}
```

copy

Component helper functions

Let's expand our *Hello* component so that it guesses the year of birth of the person being greeted:

```
const Hello = (props) => {
  const bornYear = () => {
    const yearNow = new Date().getFullYear()
    return yearNow - props.age
  }

  return (
    <div>
      <p>
        Hello {props.name}, you are {props.age} years old
      </p>
      <p>So you were probably born in {bornYear()}</p>
    </div>
  )
}
```

copy

The logic for guessing the year of birth is separated into a function of its own that is called when the component is rendered.

The person's age does not have to be passed as a parameter to the function, since it can directly access all props that are passed to the component.

If we examine our current code closely, we'll notice that the helper function is defined inside of another function that defines the behavior of our component. In Java programming, defining a function inside another one is complex and cumbersome, so not all that common. In JavaScript, however, defining functions within functions is a commonly-used technique.

Destructuring

Before we move forward, we will take a look at a small but useful feature of the JavaScript language that was added in the ES6 specification, that allows us to destructure values from objects and arrays upon assignment.

In our previous code, we had to reference the data passed to our component as `props.name` and `props.age`. Of these two expressions, we had to repeat `props.age` twice in our code.

Since `props` is an object

```
props = {
  name: 'Arto Hellas',
  age: 35,
}
```

copy

we can streamline our component by assigning the values of the properties directly into two variables `name` and `age` which we can then use in our code:

```
const Hello = (props) => {
  const name = props.name
  const age = props.age

  const bornYear = () => new Date().getFullYear() - age

  return (
    <div>
      <p>Hello {name}, you are {age} years old</p>
      <p>So you were probably born in {bornYear()}</p>
    </div>
  )
}
```

copy

Note that we've also utilized the more compact syntax for arrow functions when defining the `bornYear` function. As mentioned earlier, if an arrow function consists of a single expression, then the function body does not need to be written inside of curly braces. In this more compact form, the function simply returns the result of the single expression.

To recap, the two function definitions shown below are equivalent:

```
const bornYear = () => new Date().getFullYear() - age
```

copy

```
const bornYear = () => {
  return new Date().getFullYear() - age
}
```

Destructuring makes the assignment of variables even easier since we can use it to extract and gather the values of an object's properties into separate variables:

```
const Hello = (props) => {
  const { name, age } = props
  const bornYear = () => new Date().getFullYear() - age

  return (
    <div>
      <p>Hello {name}, you are {age} years old</p>
      <p>So you were probably born in {bornYear()}</p>
    </div>
  )
}
```

copy

When the object that we are destructuring has the values

```
props = {
  name: 'Arto Hellas',
  age: 35,
}
```

copy

the expression `const { name, age } = props` assigns the values 'Arto Hellas' to `name` and 35 to `age`.

We can take destructuring a step further:

```
const Hello = ({ name, age }) => {
  const bornYear = () => new Date().getFullYear() - age

  return (
    <div>
      <p>
        Hello {name}, you are {age} years old
      </p>
      <p>So you were probably born in {bornYear()}</p>
    </div>
  )
}
```

copy

The props that are passed to the component are now directly destructured into the variables, `name` and `age`.

This means that instead of assigning the entire `props` object into a variable called `props` and then assigning its properties to the variables `name` and `age`

```
const Hello = (props) => {
  const { name, age } = props
```

copy

we assign the values of the properties directly to variables by destructuring the `props` object that is passed to the component function as a parameter:

```
const Hello = ({ name, age }) => {
```

copy

Page re-rendering

So far all of our applications have been such that their appearance remains the same after the initial rendering. What if we wanted to create a counter where the value increased as a function of time or at the click of a button?

Let's start with the following. File *App.jsx* becomes:

```
const App = (props) => {
  const {counter} = props
  return (
    <div>{counter}</div>
  )
}

export default App
```

copy

And file *main.jsx* becomes:

```
import ReactDOM from 'react-dom/client'

import App from './App'

let counter = 1

ReactDOM.createRoot(document.getElementById('root')).render(
  <App counter={counter} />
)
```

copy

The App component is given the value of the counter via the `counter` prop. This component renders the value to the screen. What happens when the value of `counter` changes? Even if we were to add the following

```
counter += 1
```

copy

the component won't re-render. We can get the component to re-render by calling the `render` method a second time, e.g. in the following way:

```
let counter = 1

const refresh = () => {
  ReactDOM.createRoot(document.getElementById('root')).render(
    <App counter={counter} />
  )
}
```

copy

```
refresh()
counter += 1
refresh()
counter += 1
refresh()
```

The re-rendering command has been wrapped inside of the `refresh` function to cut down on the amount of copy-pasted code.

Now the component *renders three times*, first with the value 1, then 2, and finally 3. However, values 1 and 2 are displayed on the screen for such a short amount of time that they can't be noticed.

We can implement slightly more interesting functionality by re-rendering and incrementing the counter every second by using setInterval:

```
setInterval(() => {
  refresh()
  counter += 1
}, 1000)
```

copy

Making repeated calls to the `render` method is not the recommended way to re-render components. Next, we'll introduce a better way of accomplishing this effect.

Stateful component

All of our components up till now have been simple in the sense that they have not contained any state that could change during the lifecycle of the component.

Next, let's add state to our application's `App` component with the help of React's state hook.

We will change the application as follows. `main.jsx` goes back to:

```
import ReactDOM from 'react-dom/client'

import App from './App'

ReactDOM.createRoot(document.getElementById('root')).render(<App />)
```

copy

and `App.jsx` changes to the following:

```
import { useState } from 'react'

const App = () => {
  const [ counter, setCounter ] = useState(0)
```

copy

```

setTimeout(
  () => setCounter(counter + 1),
  1000
)

return (
  <div>{counter}</div>
)
}

export default App

```

In the first row, the file imports the `useState` function:

`import { useState } from 'react'`

copy

The function body that defines the component begins with the function call:

`const [counter, setCounter] = useState(0)`

copy

The function call adds *state* to the component and renders it initialized with the value zero. The function returns an array that contains two items. We assign the items to the variables `counter` and `setCounter` by using the destructuring assignment syntax shown earlier.

The `counter` variable is assigned the initial value of *state*, which is zero. The variable `setCounter` is assigned a function that will be used to *modify the state*.

The application calls the `setTimeout` function and passes it two parameters: a function to increment the counter state and a timeout of one second:

`setTimeout(
 () => setCounter(counter + 1),
 1000
)`

copy

The function passed as the first parameter to the `setTimeout` function is invoked one second after calling the `setTimeout` function

`() => setCounter(counter + 1)`

copy

When the state modifying function `setCounter` is called, *React re-renders the component* which means that the function body of the component function gets re-executed:

```
( ) => {
  const [ counter, setCounter ] = useState(0)

  setTimeout(
    ( ) => setCounter(counter + 1),
    1000
  )

  return (
    <div>{counter}</div>
  )
}
```

copy

The second time the component function is executed it calls the `useState` function and returns the new value of the state: 1. Executing the function body again also makes a new function call to `setTimeout`, which executes the one-second timeout and increments the `counter` state again. Because the value of the `counter` variable is 1, incrementing the value by 1 is essentially the same as an expression setting the value of `counter` to 2.

```
( ) => setCounter(2)
```

copy

Meanwhile, the old value of `counter` - "1" - is rendered to the screen.

Every time the `setCounter` modifies the state it causes the component to re-render. The value of the state will be incremented again after one second, and this will continue to repeat for as long as the application is running.

If the component doesn't render when you think it should, or if it renders at the "wrong time", you can debug the application by logging the values of the component's variables to the console. If we make the following additions to our code:

```
const App = ( ) => {
  const [ counter, setCounter ] = useState(0)

  setTimeout(
    ( ) => setCounter(counter + 1),
    1000
  )

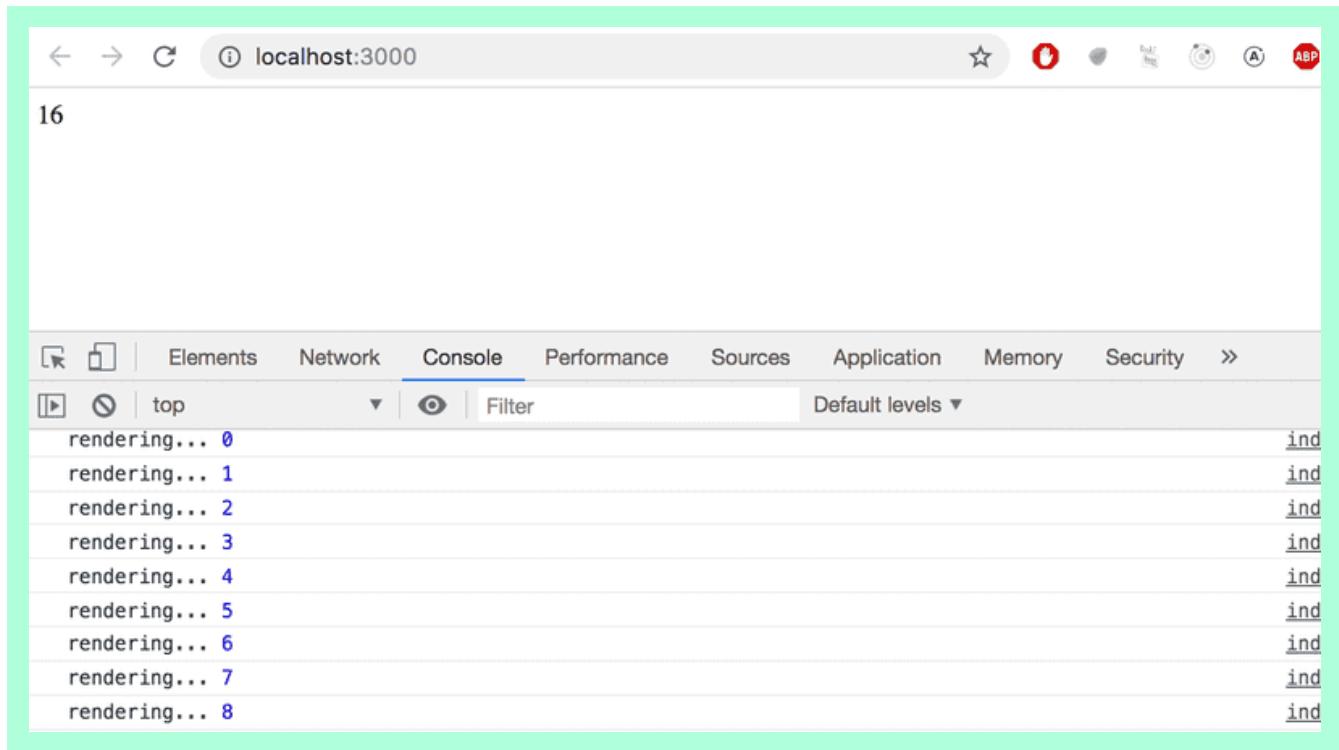
  console.log('rendering...', counter)

  return (
    <div>{counter}</div>
  )
}
```

copy

```
)  
}
```

It's easy to follow and track the calls made to the *App* component's render function:



Was your browser console open? If it wasn't, then promise that this was the last time you need to be reminded about it.

Event handling

We have already mentioned the *event handlers* that are registered to be called when specific events occur a few times in [part 0](#). A user's interaction with the different elements of a web page can cause a collection of various kinds of events to be triggered.

Let's change the application so that increasing the counter happens when a user clicks a button, which is implemented with the [button element](#).

Button elements support so-called [mouse events](#), of which [click](#) is the most common event. The click event on a button can also be triggered with the keyboard or a touch screen despite the name *mouse event*.

In React, registering an event handler function to the *click* event happens like this:

```
const App = () => {
  const [ counter, setCounter ] = useState(0)

  const handleClick = () => {
    console.log('clicked')
  }
}
```

copy

```

return (
  <div>
    <div>{counter}</div>
    <button onClick={handleClick}>
      plus
    </button>
  </div>
)
}

```

We set the value of the button's `onClick` attribute to be a reference to the `handleClick` function defined in the code.

Now every click of the `plus` button causes the `handleClick` function to be called, meaning that every click event will log a `clicked` message to the browser console.

The event handler function can also be defined directly in the value assignment of the `onClick`-attribute:

```

const App = () => {
  const [ counter, setCounter ] = useState(0)

  return (
    <div>
      <div>{counter}</div>
      <button onClick={() => console.log('clicked')}>
        plus
      </button>
    </div>
  )
}

```

[copy](#)

By changing the event handler to the following form

```

<button onClick={() => setCounter(counter + 1)}>
  plus
</button>

```

[copy](#)

we achieve the desired behavior, meaning that the value of `counter` is increased by one *and* the component gets re-rendered.

Let's also add a button for resetting the counter:

```

const App = () => {
  const [ counter, setCounter ] = useState(0)

  return (

```

[copy](#)

```
<div>
  <div>{counter}</div>
  <button onClick={() => setCounter(counter + 1)}>
    plus
  </button>
  <button onClick={() => setCounter(0)}>
    zero
  </button>
</div>
)
}
```

Our application is now ready!

An event handler is a function

We define the event handlers for our buttons where we declare their *onClick* attributes:

```
<button onClick={() => setCounter(counter + 1)}>
  plus
</button>
```

copy

What if we tried to define the event handlers in a simpler form?

```
<button onClick={setCounter(counter + 1)}>
  plus
</button>
```

copy

This would completely break our application:

The screenshot shows a browser window with the URL "localhost:3000". The developer tools console tab is active. The error message is as follows:

```

behavior.
Visit https://reactjs.org/link/error-boundaries to learn more about error boundaries.

Uncaught Error: Too many re-renders. React limits the number of renders to prevent an infinite loop.
    at renderWithHooks (react-dom.development.js:14997)
    at mountIndeterminateComponent (react-dom.development.js:17811)
    at beginWork (react-dom.development.js:19049)
    at HTMLUnknownElement.callCallback (react-dom.development.js:3945)
    at Object.invokeGuardedCallbackDev (react-dom.development.js:3994)
    at invokeGuardedCallback (react-dom.development.js:4056)
    at beginWork$1 (react-dom.development.js:23964)
    at performUnitOfWork (react-dom.development.js:22776)
    at workLoopSync (react-dom.development.js:22707)
    at renderRootSync (react-dom.development.js:22670)

```

What's going on? An event handler is supposed to be either a *function* or a *function reference*, and when we write:

```
<button onClick={setCounter(counter + 1)}>
```

[copy](#)

the event handler is actually a *function call*. In many situations this is ok, but not in this particular situation. In the beginning, the value of the *counter* variable is 0. When React renders the component for the first time, it executes the function call `setCounter(0+1)`, and changes the value of the component's state to 1. This will cause the component to be re-rendered, React will execute the `setCounter` function call again, and the state will change leading to another re-render...

Let's define the event handlers like we did before:

```
<button onClick={() => setCounter(counter + 1)}>
  plus
</button>
```

[copy](#)

Now the button's attribute which defines what happens when the button is clicked - *onClick* - has the value `() => setCounter(counter + 1)`. The `setCounter` function is called only when a user clicks

the button.

Usually defining event handlers within JSX-templates is not a good idea. Here it's ok, because our event handlers are so simple.

Let's separate the event handlers into separate functions anyway:

```
const App = () => {
  const [ counter, setCounter ] = useState(0)

  const increaseByOne = () => setCounter(counter + 1)

  const setToZero = () => setCounter(0)

  return (
    <div>
      <div>{counter}</div>
      <button onClick={increaseByOne}>
        plus
      </button>
      <button onClick={setToZero}>
        zero
      </button>
    </div>
  )
}
```

copy

Here, the event handlers have been defined correctly. The value of the `onClick` attribute is a variable containing a reference to a function:

```
<button onClick={increaseByOne}>
  plus
</button>
```

copy

Passing state - to child components

It's recommended to write React components that are small and reusable across the application and even across projects. Let's refactor our application so that it's composed of three smaller components, one component for displaying the counter and two components for buttons.

Let's first implement a `Display` component that's responsible for displaying the value of the counter.

One best practice in React is to lift the state up in the component hierarchy. The documentation says:

Often, several components need to reflect the same changing data. We recommend lifting the shared state up to their closest common ancestor.

So let's place the application's state in the *App* component and pass it down to the *Display* component through *props*:

```
const Display = (props) => {
  return (
    <div>{props.counter}</div>
  )
}
```

copy

Using the component is straightforward, as we only need to pass the state of the `counter` to it:

```
const App = () => {
  const [ counter, setCounter ] = useState(0)

  const increaseByOne = () => setCounter(counter + 1)
  const setToZero = () => setCounter(0)

  return (
    <div>
      <Display counter={counter}/>
      <button onClick={increaseByOne}>
        plus
      </button>
      <button onClick={setToZero}>
        zero
      </button>
    </div>
  )
}
```

copy

Everything still works. When the buttons are clicked and the *App* gets re-rendered, all of its children including the *Display* component are also re-rendered.

Next, let's make a *Button* component for the buttons of our application. We have to pass the event handler as well as the title of the button through the component's props:

```
const Button = (props) => {
  return (
    <button onClick={props.onClick}>
      {props.text}
    </button>
  )
}
```

copy

Our *App* component now looks like this:

```
const App = () => {
  const [ counter, setCounter ] = useState(0)

  const increaseByOne = () => setCounter(counter + 1)
  const decreaseByOne = () => setCounter(counter - 1)
  const setToZero = () => setCounter(0)

  return (
    <div>
      <Display counter={counter}>/>
      <Button
        onClick={increaseByOne}
        text='plus'
      />
      <Button
        onClick={setToZero}
        text='zero'
      />
      <Button
        onClick={decreaseByOne}
        text='minus'
      />
    </div>
  )
}
```

[copy](#)

Since we now have an easily reusable *Button* component, we've also implemented new functionality into our application by adding a button that can be used to decrement the counter.

The event handler is passed to the *Button* component through the `onClick` prop. The name of the prop itself is not that significant, but our naming choice wasn't completely random.

React's own official tutorial suggests: "In React, it's conventional to use `onSomething` names for props which take functions which handle events and `handleSomething` for the actual function definitions which handle those events."

Changes in state cause re-rendering

Let's go over the main principles of how an application works once more.

When the application starts, the code in `App` is executed. This code uses a `useState` hook to create the application state, setting an initial value of the variable `counter`. This component contains the `Display` component - which displays the counter's value, 0 - and three `Button` components. The buttons all have event handlers, which are used to change the state of the counter.

When one of the buttons is clicked, the event handler is executed. The event handler changes the state of the `App` component with the `setCounter` function. **Calling a function that changes the state causes the component to re-render.**

So, if a user clicks the *plus* button, the button's event handler changes the value of `counter` to 1, and the `App` component is re-rendered. This causes its subcomponents `Display` and `Button` to also be re-rendered. `Display` receives the new value of the counter, 1, as props. The `Button` components receive event handlers which can be used to change the state of the counter.

To be sure to understand how the program works, let us add some `console.log` statements to it

```
const App = () => {
  const [counter, setCounter] = useState(0)
  console.log('rendering with counter value', counter)

  const increaseByOne = () => {
    console.log('increasing, value before', counter)
    setCounter(counter + 1)
  }

  const decreaseByOne = () => {
    console.log('decreasing, value before', counter)
    setCounter(counter - 1)
  }

  const setToZero = () => {
    console.log('resetting to zero, value before', counter)
    setCounter(0)
  }

  return (
    <div>
      <Display counter={counter} />
      <Button onClick={increaseByOne} text="plus" />
      <Button onClick={setToZero} text="zero" />
      <Button onClick={decreaseByOne} text="minus" />
    </div>
  )
}
```

copy

Let us now see what gets rendered to the console when the buttons plus, zero and minus are pressed:

The screenshot shows a browser window with a URL of <http://localhost:3000>. At the top left, there's a counter component displaying the value **-1**. Below it are three buttons: **plus**, **zero**, and **minus**. The browser's developer tools are open, specifically the **Console** tab. The console output shows the following log messages:

```

rendering with counter value 0
increasing, value before 0
rendering with counter value 1
increasing, value before 1
rendering with counter value 2
resetting to zero, value before 2
rendering with counter value 0
decreasing, value before 0
rendering with counter value -1

```

A red box highlights the first seven lines of the log output.

Do not ever try to guess what your code does. It is just better to use `console.log` and *see with your own eyes* what it does.

Refactoring the components

The component displaying the value of the counter is as follows:

```
const Display = (props) => {
  return (
    <div>{props.counter}</div>
  )
}
```

[copy](#)

The component only uses the `counter` field of its `props`. This means we can simplify the component by using destructuring, like so:

```
const Display = ({ counter }) => {
  return (
    <div>{counter}</div>
  )
}
```

[copy](#)

The function defining the component contains only the return statement, so we can define the function using the more compact form of arrow functions:

```
const Display = ({ counter }) => <div>{counter}</div>
```

copy

We can simplify the Button component as well.

```
const Button = (props) => {
  return (
    <button onClick={props.onClick}>
      {props.text}
    </button>
  )
}
```

copy

We can use destructuring to get only the required fields from *props*, and use the more compact form of arrow functions:

NB: While building your own components, you can name their event handler props anyway you like, for this you can refer to the react's documentation on [Naming event handler props](#). It goes as following:

By convention, event handler props should start with `on`, followed by a capital letter. For example, the Button component's `onClick` prop could have been called `onSmash`:

```
const Button = ({ onClick, text }) => (
  <button onClick={onClick}>
    {text}
  </button>
)
```

copy

could also be called as following:

```
const Button = ({ onSmash, text }) => (
  <button onClick={onSmash}>
    {text}
  </button>
)
```

copy

We can simplify the Button component once more by declaring the return statement in just one line:

```
const Button = ({ onSmash, text }) => <button onClick={onSmash}>{text}</button>
```

copy

NB: However, be careful to not oversimplify your components, as this makes adding complexity a more tedious task down the road.

Propose changes to material

Part 1b

Previous part

Part 1d

Next part

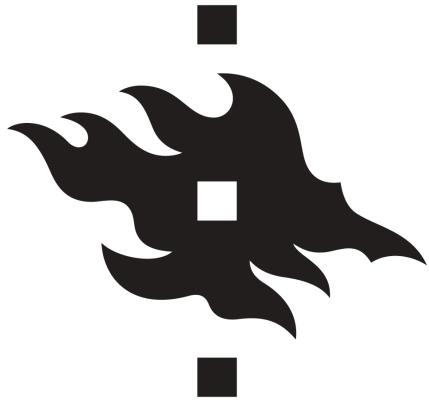
About course

Course contents

FAQ

Partners

Challenge



UNIVERSITY OF HELSINKI

HOUSTON

```
{() => fs}
```

Fullstack

Part 1

A more complex state, debugging React apps

d A more complex state, debugging React apps

Complex state

In our previous example, the application state was simple as it was comprised of a single integer. What if our application requires a more complex state?

In most cases, the easiest and best way to accomplish this is by using the `useState` function multiple times to create separate "pieces" of state.

In the following code we create two pieces of state for the application named `left` and `right` that both get the initial value of 0:

```
const App = () => {
  const [left, setLeft] = useState(0)
  const [right, setRight] = useState(0)

  return (
    <div>
      {left}
      <button onClick={() => setLeft(left + 1)}>
        left
      </button>
      <button onClick={() => setRight(right + 1)}>
        right
      </button>
      {right}
    </div>
  )
}
```

copy

```
)  
}
```

The component gets access to the functions `setLeft` and `setRight` that it can use to update the two pieces of state.

The component's state or a piece of its state can be of any type. We could implement the same functionality by saving the click count of both the *left* and *right* buttons into a single object:

```
{  
  left: 0,  
  right: 0  
}
```

copy

In this case, the application would look like this:

```
const App = () => {  
  const [clicks, setClicks] = useState({  
    left: 0, right: 0  
  })  
  
  const handleLeftClick = () => {  
    const newClicks = {  
      left: clicks.left + 1,  
      right: clicks.right  
    }  
    setClicks(newClicks)  
  }  
  
  const handleRightClick = () => {  
    const newClicks = {  
      left: clicks.left,  
      right: clicks.right + 1  
    }  
    setClicks(newClicks)  
  }  
  
  return (  
    <div>  
      {clicks.left}  
      <button onClick={handleLeftClick}>left</button>  
      <button onClick={handleRightClick}>right</button>  
      {clicks.right}  
    </div>  
  )  
}
```

copy

Now the component only has a single piece of state and the event handlers have to take care of changing the *entire application state*.

The event handler looks a bit messy. When the left button is clicked, the following function is called:

```
const handleLeftClick = () => {
  const newClicks = {
    left: clicks.left + 1,
    right: clicks.right
  }
  setClicks(newClicks)
}
```

[copy](#)

The following object is set as the new state of the application:

```
{
  left: clicks.left + 1,
  right: clicks.right
}
```

[copy](#)

The new value of the *left* property is now the same as the value of *left + 1* from the previous state, and the value of the *right* property is the same as the value of the *right* property from the previous state.

We can define the new state object a bit more neatly by using the object spread syntax that was added to the language specification in the summer of 2018:

```
const handleLeftClick = () => {
  const newClicks = {
    ...clicks,
    left: clicks.left + 1
  }
  setClicks(newClicks)
}

const handleRightClick = () => {
  const newClicks = {
    ...clicks,
    right: clicks.right + 1
  }
  setClicks(newClicks)
}
```

[copy](#)

The syntax may seem a bit strange at first. In practice `{ ...clicks }` creates a new object that has copies of all of the properties of the `clicks` object. When we specify a particular property - e.g. `right` in `{ ...clicks, right: 1 }`, the value of the `right` property in the new object will be 1.

In the example above, this:

```
{ ...clicks, right: clicks.right + 1 }
```

copy

creates a copy of the `clicks` object where the value of the `right` property is increased by one.

Assigning the object to a variable in the event handlers is not necessary and we can simplify the functions to the following form:

```
const handleLeftClick = () =>
  setClicks({ ...clicks, left: clicks.left + 1 })

const handleRightClick = () =>
  setClicks({ ...clicks, right: clicks.right + 1 })
```

copy

Some readers might be wondering why we didn't just update the state directly, like this:

```
const handleLeftClick = () => {
  clicks.left++
  setClicks(clicks)
}
```

copy

The application appears to work. However, *it is forbidden in React to mutate state directly*, since it can result in unexpected side effects. Changing state has to always be done by setting the state to a new object. If properties from the previous state object are not changed, they need to simply be copied, which is done by copying those properties into a new object and setting that as the new state.

Storing all of the state in a single state object is a bad choice for this particular application; there's no apparent benefit and the resulting application is a lot more complex. In this case, storing the click counters into separate pieces of state is a far more suitable choice.

There are situations where it can be beneficial to store a piece of application state in a more complex data structure. The official React documentation contains some helpful guidance on the topic.

Handling arrays

Let's add a piece of state to our application containing an array `allClicks` that remembers every click that has occurred in the application.

```
const App = () => {
  const [left, setLeft] = useState(0)
```

copy

```

const [right, setRight] = useState(0)
const [allClicks, setAll] = useState([])

const handleLeftClick = () => {
  setAll(allClicks.concat('L'))
  setLeft(left + 1)
}

const handleRightClick = () => {
  setAll(allClicks.concat('R'))
  setRight(right + 1)
}

return (
  <div>
    {left}
    <button onClick={handleLeftClick}>left</button>
    <button onClick={handleRightClick}>right</button>
    {right}
    <p>{allClicks.join(' ')</p>
  </div>
)
}

```

Every click is stored in a separate piece of state called `allClicks` that is initialized as an empty array:

```
const [allClicks, setAll] = useState([])
```

[copy](#)

When the `left` button is clicked, we add the letter `L` to the `allClicks` array:

```

const handleLeftClick = () => {
  setAll(allClicks.concat('L'))
  setLeft(left + 1)
}

```

[copy](#)

The piece of state stored in `allClicks` is now set to be an array that contains all of the items of the previous state array plus the letter `L`. Adding the new item to the array is accomplished with the concat method, which does not mutate the existing array but rather returns a *new copy of the array* with the item added to it.

As mentioned previously, it's also possible in JavaScript to add items to an array with the push method. If we add the item by pushing it to the `allClicks` array and then updating the state, the application would still appear to work:

```

const handleLeftClick = () => {
  allClicks.push('L')
  setAll(allClicks)
}

```

[copy](#)

```
    setLeft(left + 1)
}
```

However, **don't** do this. As mentioned previously, the state of React components, like `allClicks`, must not be mutated directly. Even if mutating state appears to work in some cases, it can lead to problems that are very hard to debug.

Let's take a closer look at how the clicking is rendered to the page:

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

  return (
    <div>
      {left}
      <button onClick={handleLeftClick}>left</button>
      <button onClick={handleRightClick}>right</button>
      {right}
      <p>{allClicks.join(' ')</p>
    </div>
  )
}
```

copy

We call the `join` method on the `allClicks` array, that joins all the items into a single string, separated by the string passed as the function parameter, which in our case is an empty space.

Update of the state is asynchronous

Let's expand the application so that it keeps track of the total number of button presses in the state `total`, whose value is always updated when the buttons are pressed:

```
const App = () => {
  const [left, setLeft] = useState(0)
  const [right, setRight] = useState(0)
  const [allClicks, setAll] = useState([])
  const [total, setTotal] = useState(0)

  const handleLeftClick = () => {
    setAll(allClicks.concat('L'))
    setLeft(left + 1)
    setTotal(left + right)
  }

  const handleRightClick = () => {
    setAll(allClicks.concat('R'))
    setRight(right + 1)
    setTotal(left + right)
  }
}
```

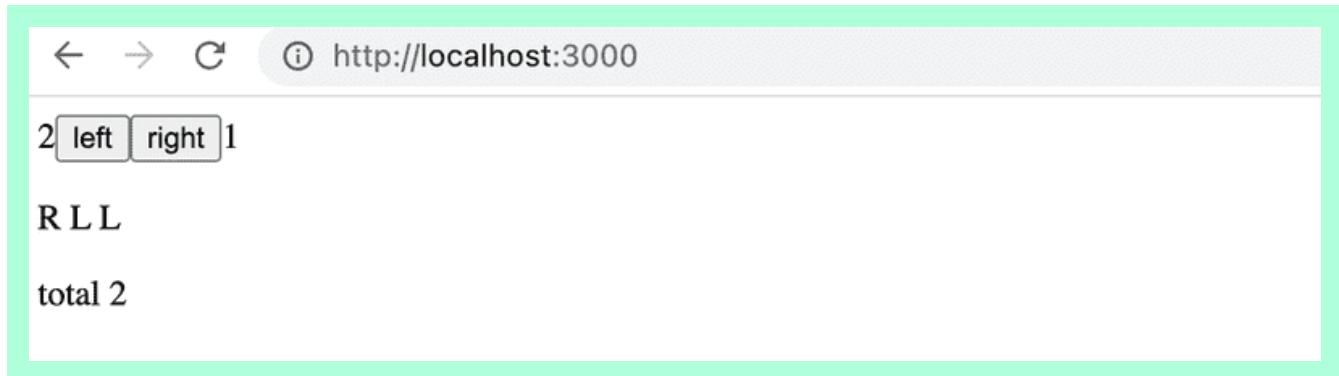
copy

```

return (
  <div>
    {left}
    <button onClick={handleLeftClick}>left</button>
    <button onClick={handleRightClick}>right</button>
    {right}
    <p>{allClicks.join(' ')</p>
    <p>total {total}</p>
  </div>
)
}

```

The solution does not quite work:



The total number of button presses is consistently one less than the actual amount of presses, for some reason.

Let us add couple of `console.log` statements to the event handler:

```

const App = () => {
  // ...
  const handleLeftClick = () => {
    setAll(allClicks.concat('L'))
    console.log('left before', left)
    setLeft(left + 1)
    console.log('left after', left)
    setTotal(left + right)
  }
  // ...
}

```

[copy](#)

The console reveals the problem

Even though a new value was set for `left` by calling `setLeft(left + 1)`, the old value persists despite the update. As a result, the attempt to count button presses produces a result that is too small:

```
setTotal(left + right)
```

[copy](#)

The reason for this is that a state update in React happens asynchronously, i.e. not immediately but "at some point" before the component is rendered again.

We can fix the app as follows:

```
const App = () => {
  // ...
  const handleLeftClick = () => {
    setAll(allClicks.concat('L'))
    const updatedLeft = left + 1
    setLeft(updatedLeft)
    setTotal(updatedLeft + right)
  }
  // ...
}
```

[copy](#)

So now the number of button presses is definitely based on the correct number of left button presses.

Conditional rendering

Let's modify our application so that the rendering of the clicking history is handled by a new *History* component:

```

const History = (props) => {
  if (props.allClicks.length === 0) {
    return (
      <div>
        the app is used by pressing the buttons
      </div>
    )
  }
  return (
    <div>
      button press history: {props.allClicks.join(' ')}
    </div>
  )
}

const App = () => {
  // ...

  return (
    <div>
      {left}
      <button onClick={handleLeftClick}>left</button>
      <button onClick={handleRightClick}>right</button>
      {right}
      <History allClicks={allClicks} />
    </div>
  )
}

```

[copy](#)

Now the behavior of the component depends on whether or not any buttons have been clicked. If not, meaning that the `allClicks` array is empty, the component renders a div element with some instructions instead:

<div>the app is used by pressing the buttons</div>

[copy](#)

And in all other cases, the component renders the clicking history:

<div>
 button press history: {props.allClicks.join(' ')}
</div>

[copy](#)

The `History` component renders completely different React elements depending on the state of the application. This is called *conditional rendering*.

React also offers many other ways of doing conditional rendering. We will take a closer look at this in part 2.

Let's make one last modification to our application by refactoring it to use the `Button` component that we defined earlier on:

```
const History = (props) => {
  if (props.allClicks.length === 0) {
    return (
      <div>
        the app is used by pressing the buttons
      </div>
    )
  }

  return (
    <div>
      button press history: {props.allClicks.join(' ')}
    </div>
  )
}

const Button = ({ handleClick, text }) => (
  <button onClick={handleClick}>
    {text}
  </button>
)

const App = () => {
  const [left, setLeft] = useState(0)
  const [right, setRight] = useState(0)
  const [allClicks, setAll] = useState([])

  const handleLeftClick = () => {
    setAll(allClicks.concat('L'))
    setLeft(left + 1)
  }

  const handleRightClick = () => {
    setAll(allClicks.concat('R'))
    setRight(right + 1)
  }

  return (
    <div>
      {left}
      <Button handleClick={handleLeftClick} text='left' />
      <Button handleClick={handleRightClick} text='right' />
      {right}
      <History allClicks={allClicks} />
    </div>
  )
}
```

copy

Old React

In this course, we use the state hook to add state to our React components, which is part of the newer versions of React and is available from version 16.8.0 onwards. Before the addition of hooks, there was no way to add state to functional components. Components that required state had to be defined as class components, using the JavaScript class syntax.

In this course, we have made the slightly radical decision to use hooks exclusively from day one, to ensure that we are learning the current and future variations of React. Even though functional components are the future of React, it is still important to learn the class syntax, as there are billions of lines of legacy React code that you might end up maintaining someday. The same applies to documentation and examples of React that you may stumble across on the internet.

We will learn more about React class components later on in the course.

Debugging React applications

A large part of a typical developer's time is spent on debugging and reading existing code. Every now and then we do get to write a line or two of new code, but a large part of our time is spent trying to figure out why something is broken or how something works. Good practices and tools for debugging are extremely important for this reason.

Lucky for us, React is an extremely developer-friendly library when it comes to debugging.

Before we move on, let us remind ourselves of one of the most important rules of web development.

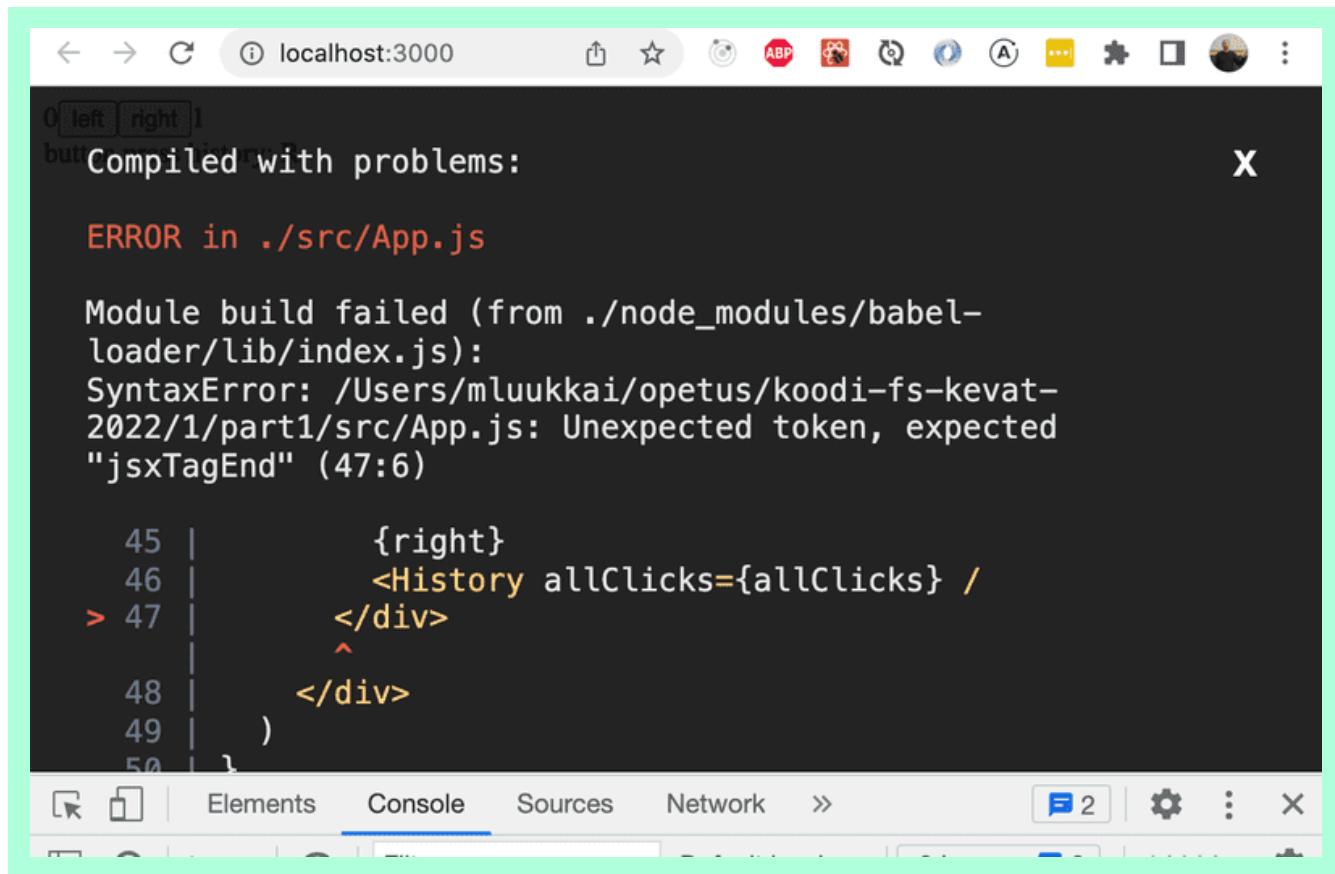
The first rule of web development

Keep the browser's developer console open at all times.

The *Console* tab in particular should always be open, unless there is a specific reason to view another tab.

Keep both your code and the web page open together **at the same time, all the time.**

If and when your code fails to compile and your browser lights up like a Christmas tree:



don't write more code but rather find and fix the problem **immediately**. There has yet to be a moment in the history of coding where code that fails to compile would miraculously start working after writing large amounts of additional code. I highly doubt that such an event will transpire during this course either.

Old-school, print-based debugging is always a good idea. If the component

```
const Button = ({ handleClick, text }) => (
  <button onClick={handleClick}>
    {text}
  </button>
)
```

copy

is not working as intended, it's useful to start printing its variables out to the console. In order to do this effectively, we must transform our function into the less compact form and receive the entire props object without destructuring it immediately:

```
const Button = (props) => {
  console.log(props)
  const { handleClick, text } = props
  return (
    <button onClick={handleClick}>
      {text}
    </button>
  )
}
```

copy

```
</button>
)
}
```

This will immediately reveal if, for instance, one of the attributes has been misspelled when using the component.

NB When you use `console.log` for debugging, don't combine `objects` in a Java-like fashion by using the plus operator:

```
console.log('props value is ' + props)
```

copy

If you do that, you will end up with a rather uninformative log message:

```
props value is [object Object]
```

copy

Instead, separate the things you want to log to the console with a comma:

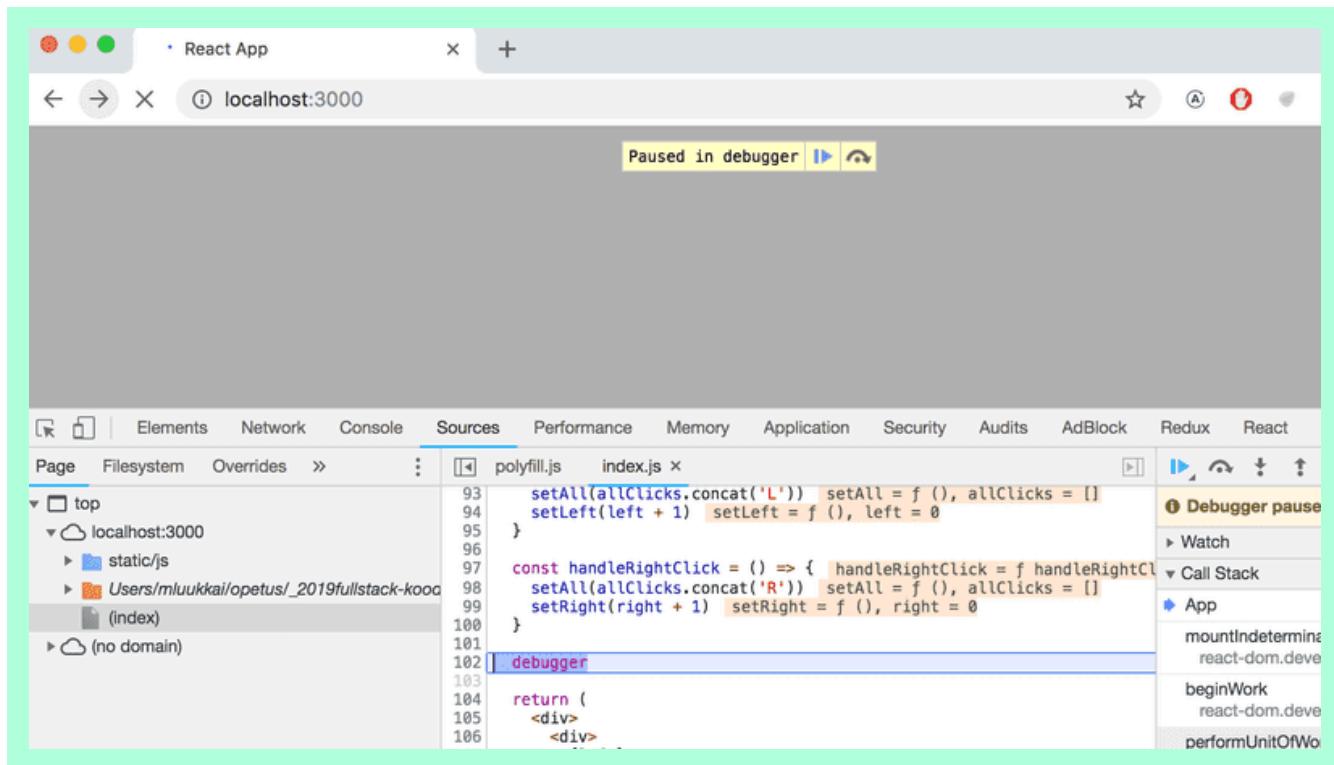
```
console.log('props value is', props)
```

copy

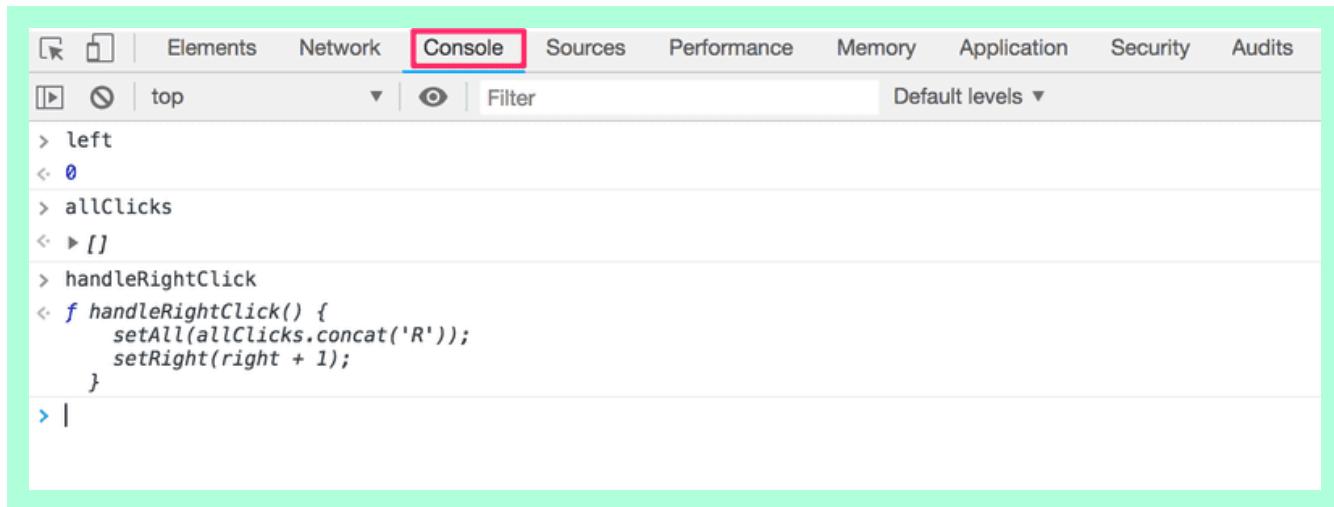
In this way, the separated items will all be available in the browser console for further inspection.

Logging output to the console is by no means the only way of debugging our applications. You can pause the execution of your application code in the Chrome developer console's *debugger*, by writing the command `debugger` anywhere in your code.

The execution will pause once it arrives at a point where the `debugger` command gets executed:



By going to the *Console* tab, it is easy to inspect the current state of variables:



Once the cause of the bug is discovered you can remove the `debugger` command and refresh the page.

The debugger also enables us to execute our code line by line with the controls found on the right-hand side of the *Sources* tab.

You can also access the debugger without the `debugger` command by adding breakpoints in the *Sources* tab. Inspecting the values of the component's variables can be done in the `Scope`-section:

The screenshot shows the Chrome DevTools interface with the 'Sources' tab selected. In the left sidebar, there's a tree view of files: 'top', 'localhost:3000', 'static/js', 'Users/mluukkai/opetus/_2019fullstack-koodi', and '(index)'. A red arrow points from the left sidebar to the line number 92 in the code editor. The code editor shows a component named 'App' with several methods like 'handleLeftClick' and 'handleRightClick'. The right panel shows the 'Scope' inspector with variables like 'this', 'Closure (App)', and 'allClicks'. The status bar at the bottom indicates 'Line 92, Column 1'.

It is highly recommended to add the React developer tools extension to Chrome. It adds a new Components tab to the developer tools. The new developer tools tab can be used to inspect the different React elements in the application, along with their state and props:

The screenshot shows the Chrome DevTools interface with the 'Components' tab selected. The URL in the address bar is 'localhost:3000'. The main area displays the state of the 'App' component: 'left' is 3 and 'right' is 2, with a history of button presses: 'L L R R L'. Below this, the 'Components' tree shows 'App' with children 'Button' and 'History'. A red arrow points to the 'History' entry. The details panel on the right shows the 'props' object with 'allClicks: ["L", "L", "R", "R", "L"]' and 'new prop: ""'. It also shows 'rendered by App' and 'source src/index.js:119'. A red box highlights the 'props' object.

The `App` component's state is defined like so:

```
const [left, setLeft] = useState(0)
const [right, setRight] = useState(0)
const [allClicks, setAll] = useState([])
```

[copy](#)

Dev tools show the state of hooks in the order of their definition:

The screenshot shows a browser window with developer tools open. The address bar says 'localhost:3000'. The developer tools navigation bar includes Elements, Console, Sources, Network, Performance, Memory, Application, Security, Components, and a warning icon. The Components tab is active. Under the 'App' component, the 'hooks' section is highlighted with a red box. It shows three entries: 'State: 3', 'State: 2', and 'State: ["L", "L", "R", "R", "L"]'. Below the hooks, it says 'source src/index.js:125'.

The first *State* contains the value of the *left* state, the next contains the value of the *right* state and the last contains the value of the *allClicks* state.

Rules of Hooks

There are a few limitations and rules that we have to follow to ensure that our application uses hook-based state functions correctly.

The `useState` function (as well as the `useEffect` function introduced later on in the course) *must not be called* from inside of a loop, a conditional expression, or any place that is not a function defining a component. This must be done to ensure that the hooks are always called in the same order, and if this isn't the case the application will behave erratically.

To recap, hooks may only be called from the inside of a function body that defines a React component:

```
const App = () => {
  // these are ok
  const [age, setAge] = useState(0)
  const [name, setName] = useState('Juha Tauriainen')

  if (age > 10) {
    // this does not work!
    const [foobar, setFoobar] = useState(null)
  }

  for (let i = 0; i < age; i++) {
    // also this is not good
    const [rightWay, setRightWay] = useState(false)
  }
}
```

copy

```
const notGood = () => {
  // and this is also illegal
  const [x, setX] = useState(-1000)
}

return (
  //...
)
```

Event Handling Revisited

Event handling has proven to be a difficult topic in previous iterations of this course.

For this reason, we will revisit the topic.

Let's assume that we're developing this simple application with the following component *App*:

```
const App = () => {
  const [value, setValue] = useState(10)

  return (
    <div>
      {value}
      <button>reset to zero</button>
    </div>
  )
}
```

copy

We want the clicking of the button to reset the state stored in the `value` variable.

In order to make the button react to a click event, we have to add an *event handler* to it.

Event handlers must always be a function or a reference to a function. The button will not work if the event handler is set to a variable of any other type.

If we were to define the event handler as a string:

```
<button onClick="crap...">button</button>
```

copy

React would warn us about this in the console:

```
index.js:2178 Warning: Expected `onClick` listener to be a function, instead got a value
of `string` type.
in button (at index.js:20)
```

```
in div (at index.js:18)
in App (at index.js:27)
```

The following attempt would also not work:

```
<button onClick={value + 1}>button</button>
```

copy

We have attempted to set the event handler to `value + 1` which simply returns the result of the operation. React will kindly warn us about this in the console:

```
index.js:2178 Warning: Expected `onClick` listener to be a function, instead got a value of `number` type.
```

This attempt would not work either:

```
<button onClick={value = 0}>button</button>
```

copy

The event handler is not a function but a variable assignment, and React will once again issue a warning to the console. This attempt is also flawed in the sense that we must never mutate state directly in React.

What about the following:

```
<button onClick={console.log('clicked the button')}>
  button
</button>
```

copy

The message gets printed to the console once when the component is rendered but nothing happens when we click the button. Why does this not work even when our event handler contains a function `console.log` ?

The issue here is that our event handler is defined as a *function call* which means that the event handler is assigned the returned value from the function, which in the case of `console.log` is *undefined*.

The `console.log` function call gets executed when the component is rendered and for this reason, it gets printed once to the console.

The following attempt is flawed as well:

```
<button onClick={setValue(0)}>button</button>
```

copy

We have once again tried to set a function call as the event handler. This does not work. This particular attempt also causes another problem. When the component is rendered the function `setValue(0)` gets executed which in turn causes the component to be re-rendered. Re-rendering in turn calls `setValue(0)` again, resulting in an infinite recursion.

Executing a particular function call when the button is clicked can be accomplished like this:

```
<button onClick={() => console.log('clicked the button')}>
  button
</button>
```

copy

Now the event handler is a function defined with the arrow function syntax `() => console.log('clicked the button')`. When the component gets rendered, no function gets called and only the reference to the arrow function is set to the event handler. Calling the function happens only once the button is clicked.

We can implement resetting the state in our application with this same technique:

```
<button onClick={() => setValue(0)}>button</button>
```

copy

The event handler is now the function `() => setValue(0)`.

Defining event handlers directly in the attribute of the button is not necessarily the best possible idea.

You will often see event handlers defined in a separate place. In the following version of our application we define a function that then gets assigned to the `handleClick` variable in the body of the component function:

```
const App = () => {
  const [value, setValue] = useState(10)

  const handleClick = () =>
    console.log('clicked the button')

  return (
    <div>
      {value}
      <button onClick={handleClick}>button</button>
    </div>
  )
}
```

copy

The `handleClick` variable is now assigned to a reference to the function. The reference is passed to the button as the `onClick` attribute:

```
<button onClick={handleClick}>button</button>
```

[copy](#)

Naturally, our event handler function can be composed of multiple commands. In these cases we use the longer curly brace syntax for arrow functions:

```
const App = () => {
  const [value, setValue] = useState(10)

  const handleClick = () => {
    console.log('clicked the button')
    setValue(0)
  }

  return (
    <div>
      {value}
      <button onClick={handleClick}>button</button>
    </div>
  )
}
```

[copy](#)

A function that returns a function

Another way to define an event handler is to use a *function that returns a function*.

You probably won't need to use functions that return functions in any of the exercises in this course. If the topic seems particularly confusing, you may skip over this section for now and return to it later.

Let's make the following changes to our code:

```
const App = () => {
  const [value, setValue] = useState(10)

  const hello = () => {
    const handler = () => console.log('hello world')
    return handler
  }

  return (
    <div>
      {value}
      <button onClick={hello()}>button</button>
    </div>
  )
}
```

[copy](#)

The code functions correctly even though it looks complicated.

The event handler is now set to a function call:

```
<button onClick={hello()}>button</button>
```

copy

Earlier on we stated that an event handler may not be a call to a function and that it has to be a function or a reference to a function. Why then does a function call work in this case?

When the component is rendered, the following function gets executed:

```
const hello = () => {
  const handler = () => console.log('hello world')

  return handler
}
```

copy

The *return value* of the function is another function that is assigned to the `handler` variable.

When React renders the line:

```
<button onClick={hello()}>button</button>
```

copy

It assigns the return value of `hello()` to the `onClick` attribute. Essentially the line gets transformed into:

```
<button onClick={() => console.log('hello world')}>
  button
</button>
```

copy

Since the `hello` function returns a function, the event handler is now a function.

What's the point of this concept?

Let's change the code a tiny bit:

```
const App = () => {
  const [value, setValue] = useState(10)

  const hello = (who) => {
    const handler = () => {
```

copy

```

        console.log('hello', who)
    }
    return handler
}

return (
    <div>
        {value}
        <button onClick={hello('world')}>button</button>
        <button onClick={hello('react')}>button</button>
        <button onClick={hello('function')}>button</button>
    </div>
)
}

```

Now the application has three buttons with event handlers defined by the `hello` function that accepts a parameter.

The first button is defined as

```
<button onClick={hello('world')}>button</button>
```

[copy](#)

The event handler is created by *executing* the function call `hello('world')`. The function call returns the function:

```
( ) => {
    console.log('hello', 'world')
}
```

[copy](#)

The second button is defined as:

```
<button onClick={hello('react')}>button</button>
```

[copy](#)

The function call `hello('react')` that creates the event handler returns:

```
( ) => {
    console.log('hello', 'react')
}
```

[copy](#)

Both buttons get their individualized event handlers.

Functions returning functions can be utilized in defining generic functionality that can be customized with parameters. The `hello` function that creates the event handlers can be thought of as a factory

that produces customized event handlers meant for greeting users.

Our current definition is slightly verbose:

```
const hello = (who) => {
  const handler = () => {
    console.log('hello', who)
  }

  return handler
}
```

copy

Let's eliminate the helper variables and directly return the created function:

```
const hello = (who) => {
  return () => {
    console.log('hello', who)
  }
}
```

copy

Since our `hello` function is composed of a single return command, we can omit the curly braces and use the more compact syntax for arrow functions:

```
const hello = (who) =>
  () => {
    console.log('hello', who)
  }

```

copy

Lastly, let's write all of the arrows on the same line:

```
const hello = (who) => () => {
  console.log('hello', who)
}
```

copy

We can use the same trick to define event handlers that set the state of the component to a given value. Let's make the following changes to our code:

```
const App = () => {
  const [value, setValue] = useState(10)

  const setToValue = (newValue) => () => {
```

copy

```

    console.log('value now', newValue) // print the new value to console
    setValue(newValue)
}

return (
  <div>
    {value}
    <button onClick={setToValue(1000)}>thousand</button>
    <button onClick={setToValue(0)}>reset</button>
    <button onClick={setToValue(value + 1)}>increment</button>
  </div>
)
}

```

When the component is rendered, the *thousand* button is created:

```
<button onClick={setToValue(1000)}>thousand</button>
```

[copy](#)

The event handler is set to the return value of `setToValue(1000)` which is the following function:

```

() => {
  console.log('value now', 1000)
  setValue(1000)
}

```

[copy](#)

The increase button is declared as follows:

```
<button onClick={setToValue(value + 1)}>increment</button>
```

[copy](#)

The event handler is created by the function call `setToValue(value + 1)` which receives as its parameter the current value of the state variable `value` increased by one. If the value of `value` was 10, then the created event handler would be the function:

```

() => {
  console.log('value now', 11)
  setValue(11)
}

```

[copy](#)

Using functions that return functions is not required to achieve this functionality. Let's return the `setToValue` function which is responsible for updating state into a normal function:

```
const App = () => {
  const [value, setValue] = useState(10)

  const setToValue = (newValue) => {
    console.log('value now', newValue)
    setValue(newValue)
  }

  return (
    <div>
      {value}
      <button onClick={() => setToValue(1000)}>
        thousand
      </button>
      <button onClick={() => setToValue(0)}>
        reset
      </button>
      <button onClick={() => setToValue(value + 1)}>
        increment
      </button>
    </div>
  )
}
```

[copy](#)

We can now define the event handler as a function that calls the `setToValue` function with an appropriate parameter. The event handler for resetting the application state would be:

```
<button onClick={() => setToValue(0)}>reset</button>
```

[copy](#)

Choosing between the two presented ways of defining your event handlers is mostly a matter of taste.

Passing Event Handlers to Child Components

Let's extract the button into its own component:

```
const Button = (props) => (
  <button onClick={props.handleClick}>
    {props.text}
  </button>
)
```

[copy](#)

The component gets the event handler function from the `handleClick` prop, and the text of the button from the `text` prop. Lets use the new component:

```
const App = (props) => {
  // ...
  return (
    <div>
      {value}
      <Button handleClick={() => setToValue(1000)} text="thousand" />
      <Button handleClick={() => setToValue(0)} text="reset" />
      <Button handleClick={() => setToValue(value + 1)} text="increment" />
    </div>
  )
}
```

[copy](#)

Using the *Button* component is simple, although we have to make sure that we use the correct attribute names when passing props to the component.

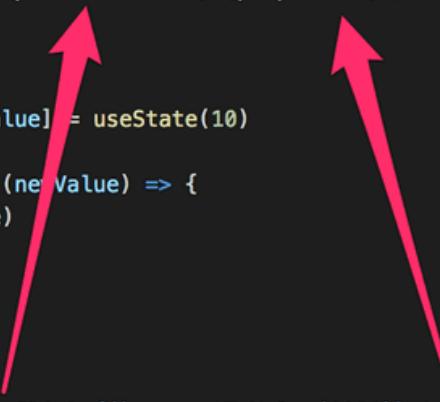
```
import React, {useState} from 'react'
import ReactDOM from 'react-dom'

const Button = (props) => (
  <button onClick={props.handleClick}>{props.text}</button>
)

const App = () => {
  const [value, setValue] = useState(10)

  const setToValue = (newValue) => {
    setValue(newValue)
  }

  return (
    <div>
      {value}
      <Button handleClick={() => setToValue(1000)} text="thousand" />
      <Button handleClick={() => setToValue(0)} text="reset" />
      <Button handleClick={() => setToValue(value + 1)} text="increment" />
    </div>
  )
}
```


[copy](#)

Do Not Define Components Within Components

Let's start displaying the value of the application in its *Display* component.

We will change the application by defining a new component inside of the *App* component.

```
// This is the right place to define a component
const Button = (props) => (
  <button onClick={props.handleClick}>
    {props.text}
  </button>
)
```

[copy](#)

```

        </button>
    )

const App = () => {
  const [value, setValue] = useState(10)

  const setToValue = newValue => {
    console.log('value now', newValue)
    setValue(newValue)
  }

  // Do not define components inside another component
  const Display = props => <div>{props.value}</div>

  return (
    <div>
      <Display value={value} />
      <Button handleClick={() => setToValue(1000)} text="thousand" />
      <Button handleClick={() => setToValue(0)} text="reset" />
      <Button handleClick={() => setToValue(value + 1)} text="increment" />
    </div>
  )
}

```

The application still appears to work, but **don't implement components like this!** Never define components inside of other components. The method provides no benefits and leads to many unpleasant problems. The biggest problems are because React treats a component defined inside of another component as a new component in every render. This makes it impossible for React to optimize the component.

Let's instead move the *Display* component function to its correct place, which is outside of the *App* component function:

```

const Display = props => <div>{props.value}</div>

const Button = (props) => (
  <button onClick={props.handleClick}>
    {props.text}
  </button>
)

const App = () => {
  const [value, setValue] = useState(10)

  const setToValue = newValue => {
    console.log('value now', newValue)
    setValue(newValue)
  }

  return (
    <div>
      <Display value={value} />

```



```

<Button handleClick={() => setToValue(1000)} text="thousand" />
<Button handleClick={() => setToValue(0)} text="reset" />
<Button handleClick={() => setToValue(value + 1)} text="increment" />
</div>
)
}

```

Useful Reading

The internet is full of React-related material. However, we use the new style of React for which a large majority of the material found online is outdated.

You may find the following links useful:

- The [official React documentation](#) is worth checking out at some point, although most of it will become relevant only later on in the course. Also, everything related to class-based components is irrelevant to us;
- Some courses on [Egghead.io](#) like [Start learning React](#) are of high quality, and the recently updated [Beginner's Guide to React](#) is also relatively good; both courses introduce concepts that will also be introduced later on in this course. **NB** The first one uses class components but the latter uses the new functional ones.

Web programmers oath

Programming is hard, that is why I will use all the possible means to make it easier

- I will have my browser developer console open all the time
- I progress with small steps
- I will write lots of `console.log` statements to make sure I understand how the code behaves and to help pinpointing problems
- If my code does not work, I will not write more code. Instead I start deleting the code until it works or just return to a state when everything was still working
- When I ask for help in the course Discord or Telegram channel or elsewhere I formulate my questions properly, see [here](#) how to ask for help

Utilization of Large language models

Large language models such as [ChatGPT](#), [Claude](#) and [GitHub Copilot](#) have proven to be very useful in software development.

Personally, I mainly use Copilot, which integrates seamlessly with VS Code thanks to the [plugin](#).

Copilot is useful in a wide variety of scenarios. Copilot can be asked to generate code for an open file by describing the desired functionality in text:

```
39
40     return []
41     <div>
42       <div>
43         {left}
44
45         <Button handleClick={handleLeftClick} text='left' />
46         <Button handleClick={handleRightClick} text='right' />
47         {right}
48         <History allClicks={allClicks} />
49       </div>
50     </div>
51   ]
52 }
```

add a button that will reset the left and right



...

You can also type / for commands

If the code looks good, Copilot adds it to the file:

```
return (
  <div>
    <div>
      {left}
      <Button handleClick={handleResetClick} text='reset' />
      <Button handleClick={handleLeftClick} text='left' />
      <Button handleClick={handleRightClick} text='right' />
    {right}
```

In the case of our example, Copilot only created a button, the event handler `handleResetClick` is undefined.

An event handler may also be generated. By writing the first line of the function, Copilot offers the functionality to be generated:

```

◆ const handleResetClick = () => {
  setAll([])
  setLeft(0)
  setRight(0)
}

return (
  <div>
    <div>
      {left}
      <Button handleClick={handleResetClick} text='reset' />
      <Button handleClick={handleLeftClick} text='left' />
      <Button handleClick={handleRightClick} text='right' />
    {right}
    <History allClicks={allClicks} />
  </div>
</div>

```

In Copilot's chat window, it is possible to ask for an explanation of the function of the painted code area:

The screenshot shows a GitHub Copilot interface. On the left, there's a sidebar with a user icon, the name 'mluukkai', and a message '@workspace /explain'. Below that, it says 'Used 1 reference' and provides a detailed explanation of the code. On the right, the code is displayed in a monospaced font.

```

src > % App.jsx > [diff] App
25  const App = () => {
30  const handleLeftClick = () => {
31    setAll(allClicks.concat('L'))
32    setLeft(left + 1)
33  }
34
35  const handleRightClick = () => {
36    setAll(allClicks.concat('R'))
37    setRight(right + 1)
38  }
39
40  const handleResetClick = () => {
41    setAll([])
42    setLeft(0)
43    setRight(0)
44  }
45
46  return (
47    <div>
48      <div>
49        {left}
50        <Button handleClick={handleResetClick} text='reset' />
51        <Button handleClick={handleLeftClick} text='left' />
52        <Button handleClick={handleRightClick} text='right' />
53        {right}
54        <History allClicks={allClicks} />
55      </div>
56    </div>
57  )
58
59
60  export default App

```

This JavaScript code is written in React, a popular library for building user interfaces. The code is a part of a React component's render method, which is responsible for describing what the component's UI should look like.

The code returns a JSX expression, which is a syntax extension for JavaScript that allows you to write HTML-like code in your JavaScript files. JSX is used by React to define and create elements in a declarative and readable way.

The returned JSX expression consists of a parent `div` element that contains another `div` element. Inside the inner `div`, there are several elements and components being rendered:

- 1. `{left}`:** This is a JavaScript expression wrapped in curly braces. It will be replaced by the value of the `left` variable, which is not defined in the provided code snippet but is presumably defined elsewhere in the component.
- 2. `Button` components:** There are three instances of a `Button` component, each with different props. The `handleClick` prop is presumably a function that will be called when the button is clicked. The `text` prop is a string that will be displayed as the button's label. The functions `handleResetClick`, `handleLeftClick`, and `handleRightClick` are not defined in the provided code snippet but are presumably defined elsewhere in the component.

Copilot is also useful in error situations, by copying the error message into Copilot's chat, you will get an explanation of the problem and a suggested fix:

The screenshot shows a browser developer tools window with the 'Console' tab selected. On the left, the code editor displays a file named 'App.jsx' with the following content:

```
[plugin:vite:react-babel] /Users/mluukkai/opetus/2024-fs/ai/part1ai/src/App.jsx: Unexpected token
  47 |   <div>
    |
  42 |     setLeft(0)
  43 |     setRight(0)
  44 |   }
    |
  45 |
  46 |   return (
    |
  at constructor (/Users/mluukkai/opetus/2024-fs/ai/part1ai/src/App.jsx?t=1711019142540 net::ERR_ABORTED 500 (Intended to reload /src/App.jsx. This could be due to syntax errors or incomplete requests.
```

The error message is: [plugin:vite:react-babel] /Users/mluukkai/opetus/2024-fs/ai/part1ai/src/App.jsx: Unexpected token, expected "," (44:2) 47 | /Users/mluukkai/opetus/2024-fs/ai/part1ai/src/App.jsx:44:2 42 | setLeft(0) 43 | setRight(0) 44 | } 45 | 46 | return (at constructor (/Users/mluukkai/opetus/2024-fs/ai/part1ai/src/App.jsx?t=1711019142540 net::ERR_ABORTED 500 (Intended to reload /src/App.jsx. This could be due to syntax errors or incomplete requests.

The right side of the interface shows a 'CHAT: GITHUB COPILOT' window. It has a sidebar with icons for file, search, copy, and GitHub. The main area shows a message from 'mluukkai' and a response from 'GitHub Copilot'. The GitHub Copilot message says:

The error message is indicating that there's a syntax error in your code. It's expecting a comma (",") or a closing parenthesis ("}") on line 44, but it found a closing brace ("}") instead.

The issue is with the `setRight(0)` line. It seems like you forgot to close the parenthesis.

Here's the corrected code:

```
const handleResetClick = () => {
  setAll([])
  setLeft(0)
  setRight(0)
}
```

This should resolve the syntax error you're encountering.

↳ How can I fix the syntax error in my React code?

Copilot's chat also enables the creation of larger set of functionality

m luukkai

@workspace create a component that makes it possible to log in to the app

 GitHub Copilot used @workspace

Used 7 references >

You can create a simple login component using React's useState hook to manage the state of the username and password fields. Here's a basic example:

```
import React, { useState } from 'react';

const Login = () => {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');

  const handleLogin = (event) => {
    event.preventDefault();
    // Handle login logic here
    console.log(`Logging in with username: ${username} and password: ${password}`);
  };

  return (
    <div>
      <form onSubmit={handleLogin}>
        <label>
          Username:
          <input type="text" value={username} onChange={(e) => setUsername(e.target.value)} />
        </label>
        <label>
          Password:
          <input type="password" value={password} onChange={(e) => setPassword(e.target.value)} />
        </label>
        <input type="submit" value="Login" />
      </form>
    </div>
  );
}
```

The degree of usefulness of the hints provided by Copilot and other language models varies. Perhaps the biggest problem with language models is hallucination, they sometimes generate completely convincing-looking answers, which, however, are completely wrong. When programming, of course, the hallucinated code is often caught quickly if the code does not work. More problematic situations are those where the code generated by the language model seems to work, but it contains more difficult to detect bugs or e.g. security vulnerabilities.

Another problem in applying language models to software development is that it is difficult for language models to "understand" larger projects, and e.g. to generate functionality that would require changes to several files. Language models are also currently unable to generalize code, i.e. if the code has, for example, existing functions or components that the language model could use with minor changes for the requested functionality, the language model will not bend to this. The result of this can be that the code base deteriorates, as the language models generate a lot of repetition in the code, see more e.g. [here](#).

When using language models, the responsibility always stays with the programmer.

The rapid development of language models puts the student of programming in a challenging position: is it worth and is it even necessary to learn programming in a detailed level, when you can get almost everything ready-made from language models?

At this point, it is worth remembering the old wisdom of Brian Kernighan, the developer of the programming language C:



“Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?”

— Brian Kernighan

In other words, since debugging is twice as difficult as programming, it is not worth programming such code that you can only barely understand. How can debugging be even possible in a situation where programming is outsourced to a language model and the software developer does not understand the debugged code at all?

So far, the development of language models and artificial intelligence is still at the stage where they are not self-sufficient, and the most difficult problems are left for humans to solve. Because of this, even novice software developers must learn to program really well just in case. It may be that, despite the development of language models, even more in-depth knowledge is needed. Artificial intelligence does the easy things, but a human is needed to sort out the most complicated messes caused by AI. GitHub Copilot is a very well-named product, it's Copilot, a second pilot who helps the main pilot in an aircraft. The programmer is still the main pilot, the captain and bears the ultimate responsibility.

It may be in your own interest that you turn off Copilot by default when you do this course and rely on it only in a real emergency.

Exercises 1.6.-1.14.

Submit your solutions to the exercises by first pushing your code to GitHub and then marking the completed exercises into the "my submissions" tab of the submission application.

Remember, submit **all** the exercises of one part **in a single submission**. Once you have submitted your solutions for one part, **you cannot submit more exercises to that part anymore**.

Some of the exercises work on the same application. In these cases, it is sufficient to submit just the final version of the application. If you wish, you can make a commit after every finished exercise, but it is not mandatory.

In some situations you may also have to run the command below from the root of the project:

```
rm -rf node_modules/ && npm i
```

copy

If and *when* you encounter an error message

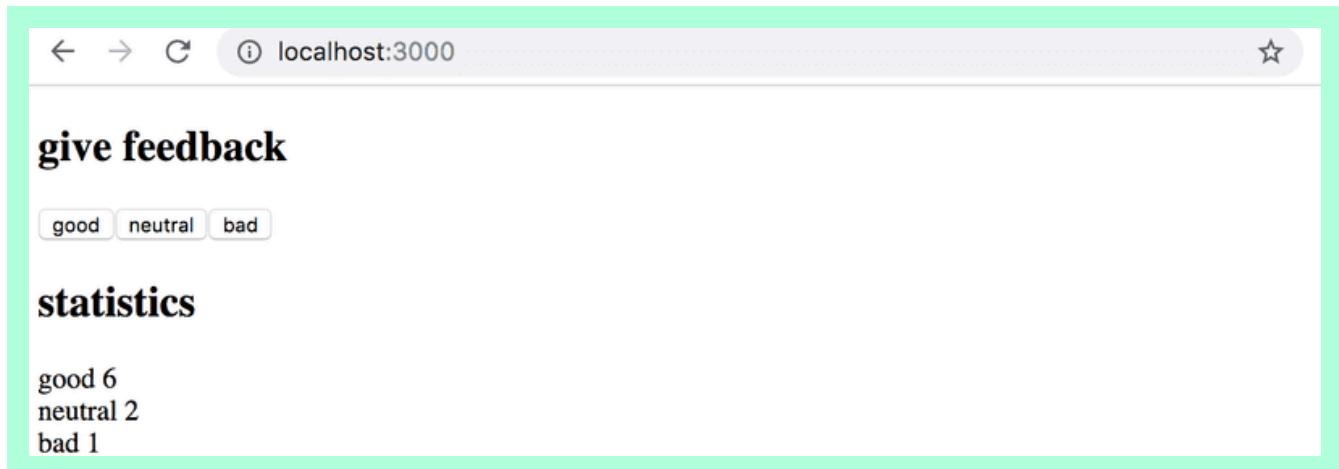
Objects are not valid as a React child

keep in mind the things told here.

1.6: unicafe step 1

Like most companies, the student restaurant of the University of Helsinki Unicafe collects feedback from its customers. Your task is to implement a web application for collecting customer feedback. There are only three options for feedback: *good*, *neutral*, and *bad*.

The application must display the total number of collected feedback for each category. Your final application could look like this:



Note that your application needs to work only during a single browser session. Once you refresh the page, the collected feedback is allowed to disappear.

It is advisable to use the same structure that is used in the material and previous exercise. File *main.jsx* is as follows:

```
import React from 'react'
import ReactDOM from 'react-dom/client'

import App from './App'

ReactDOM.createRoot(document.getElementById('root')).render(<App />)
```

copy

You can use the code below as a starting point for the *App.jsx* file:

```
import { useState } from 'react'

const App = () => {
  // save clicks of each button to its own state
  const [good, setGood] = useState(0)
  const [neutral, setNeutral] = useState(0)
```

copy

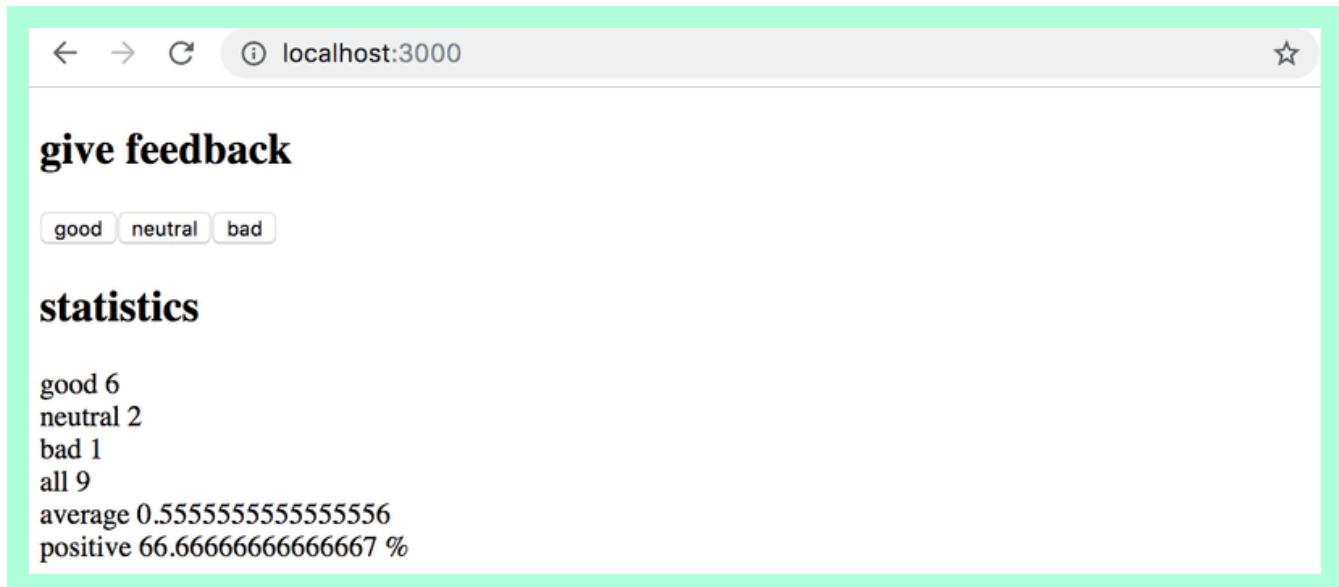
```
const [bad, setBad] = useState(0)

return (
  <div>
    code here
  </div>
)
}

export default App
```

1.7: unicafe step 2

Expand your application so that it shows more statistics about the gathered feedback: the total number of collected feedback, the average score (good: 1, neutral: 0, bad: -1) and the percentage of positive feedback.



1.8: unicafe step 3

Refactor your application so that displaying the statistics is extracted into its own *Statistics* component. The state of the application should remain in the *App* root component.

Remember that components should not be defined inside other components:

```
// a proper place to define a component
const Statistics = (props) => {
  // ...
}

const App = () => {
  const [good, setGood] = useState(0)
  const [neutral, setNeutral] = useState(0)
```

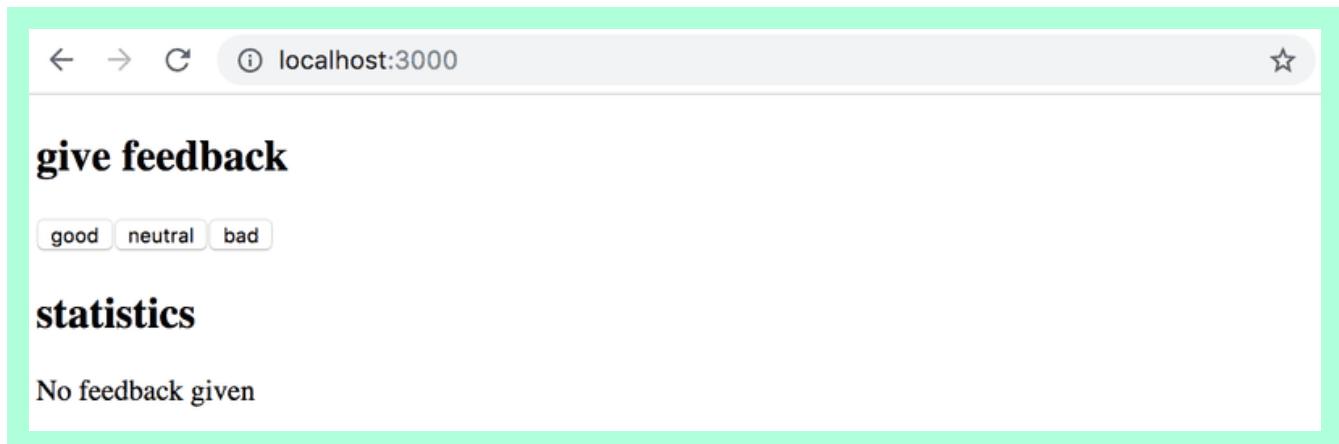
```
const [bad, setBad] = useState(0)

// do not define a component within another component
const Statistics = (props) => {
  // ...
}

return (
  // ...
)
}
```

1.9: unicafe step 4

Change your application to display statistics only once feedback has been gathered.



1.10: unicafe step 5

Let's continue refactoring the application. Extract the following two components:

- *Button* handles the functionality of each feedback submission button.
- *StatisticLine* for displaying a single statistic, e.g. the average score.

To be clear: the *StatisticLine* component always displays a single statistic, meaning that the application uses multiple components for rendering all of the statistics:

```
const Statistics = (props) => {
  // ...
  return(
    <div>
      <StatisticLine text="good" value={...} />
      <StatisticLine text="neutral" value={...} />
      <StatisticLine text="bad" value={...} />
      // ...
    </div>
  )
}
```

copy

```

    )
}

```

The application's state should still be kept in the root *App* component.

1.11*: unicafe step 6

Display the statistics in an HTML table, so that your application looks roughly like this:

good	5
neutral	2
bad	1
all	8
average	0.5
positive	62.5 %

Remember to keep your console open at all times. If you see this warning in your console:

```

in App (at src/index.js:56)
✖ Warning: validateDOMNesting(...): <tr> cannot appear as a child of <table>. Add a <tbody> to your code to match
the DOM tree generated by the browser.
  in tr (at src/index.js:5)
  in Statistic (at src/index.js:34)
  in table (at src/index.js:28)
  in div (at src/index.js:26)
  in Statistics (at src/index.js:51)
  in div (at src/index.js:46)
  in App (at src/index.js:56)

```

Then perform the necessary actions to make the warning disappear. Try pasting the error message into a search engine if you get stuck.

Typical source of an error Unchecked runtime.lastError: Could not establish connection. Receiving end does not exist. *is from a Chrome extension. Try going to chrome://extensions/ and try disabling them one by one and refreshing React app page; the error should eventually disappear.*

Make sure that from now on you don't see any warnings in your console!

1.12*: anecdotes step 1

The world of software engineering is filled with anecdotes that distill timeless truths from our field into short one-liners.

Expand the following application by adding a button that can be clicked to display a *random* anecdote from the field of software engineering:

```
import { useState } from 'react'

const App = () => {
  const anecdotes = [
    'If it hurts, do it more often.',
    'Adding manpower to a late software project makes it later!',
    'The first 90 percent of the code accounts for the first 90 percent of the development time...The remaining 10 percent of the code accounts for the other 90 percent of the development time.',
    'Any fool can write code that a computer can understand. Good programmers write code that humans can understand.',
    'Premature optimization is the root of all evil.',
    'Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.',
    'Programming without an extremely heavy use of console.log is same as if a doctor would refuse to use x-rays or blood tests when diagnosing patients.',
    'The only way to go fast, is to go well.'
  ]

  const [selected, setSelected] = useState(0)

  return (
    <div>
      {anecdotes[selected]}
    </div>
  )
}

export default App
```

copy

Content of the file *main.jsx* is the same as in previous exercises.

Find out how to generate random numbers in JavaScript, eg. via a search engine or on Mozilla Developer Network. Remember that you can test generating random numbers e.g. straight in the console of your browser.

Your finished application could look something like this:

If it hurts, do it more often
next anecdote

1.13*: anecdotes step 2

Expand your application so that you can vote for the displayed anecdote.

Premature optimization is the root of all evil.
has 7 votes
vote next anecdote

NB store the votes of each anecdote into an array or object in the component's state. Remember that the correct way of updating state stored in complex data structures like objects and arrays is to make a copy of the state.

You can create a copy of an object like this:

```
const points = { 0: 1, 1: 3, 2: 4, 3: 2 }
```

copy

```
const copy = { ...points }
// increment the property 2 value by one
copy[2] += 1
```

OR a copy of an array like this:

```
const points = [1, 4, 6, 3]
```

copy

```
const copy = [...points]
// increment the value in position 2 by one
copy[2] += 1
```

Using an array might be the simpler choice in this case. Searching the Internet will provide you with lots of hints on how to create a zero-filled array of the desired length.

1.14*: anecdotes step 3

Now implement the final version of the application that displays the anecdote with the largest number of votes:

Anecdote of the day

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

has 0 votes

[vote](#) [next anecdote](#)

Anecdote with most votes

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

has 7 votes

If multiple anecdotes are tied for first place it is sufficient to just show one of them.

This was the last exercise for this part of the course and it's time to push your code to GitHub and mark all of your finished exercises to the "my submissions" tab of the [submission application](#).

[Propose changes to material](#)

Part 1c

[Previous part](#)

Part 2

[Next part](#)

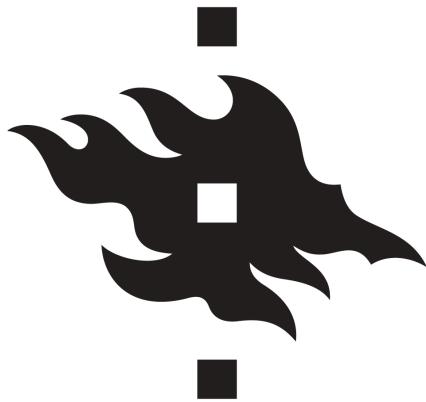
[About course](#)

[Course contents](#)

[FAQ](#)

[Partners](#)

[Challenge](#)



UNIVERSITY OF HELSINKI

HOUSTON