

```
{() => 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# npm 6.x (outdated, but still used by some):  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
```

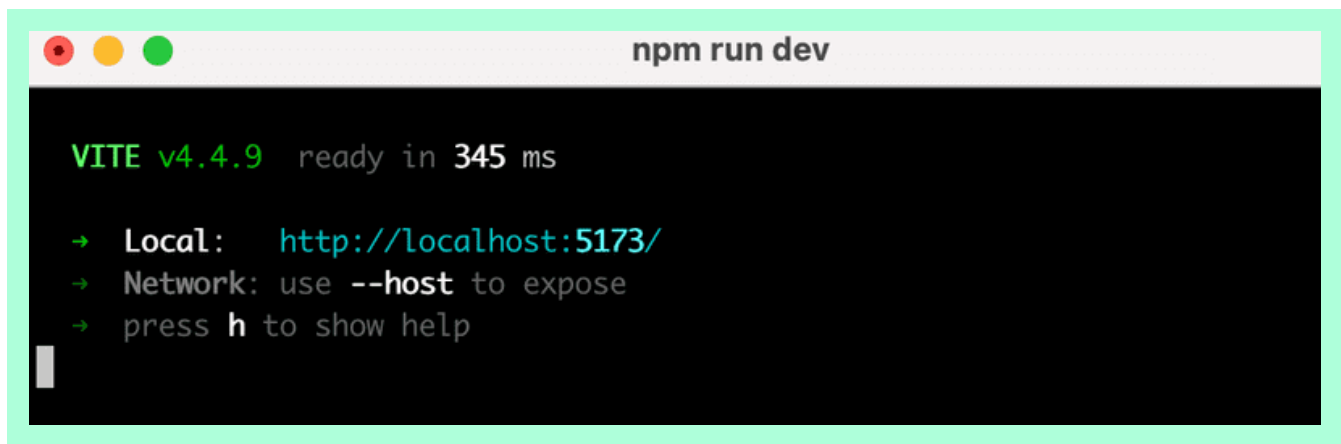
[copy](#)

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/>:



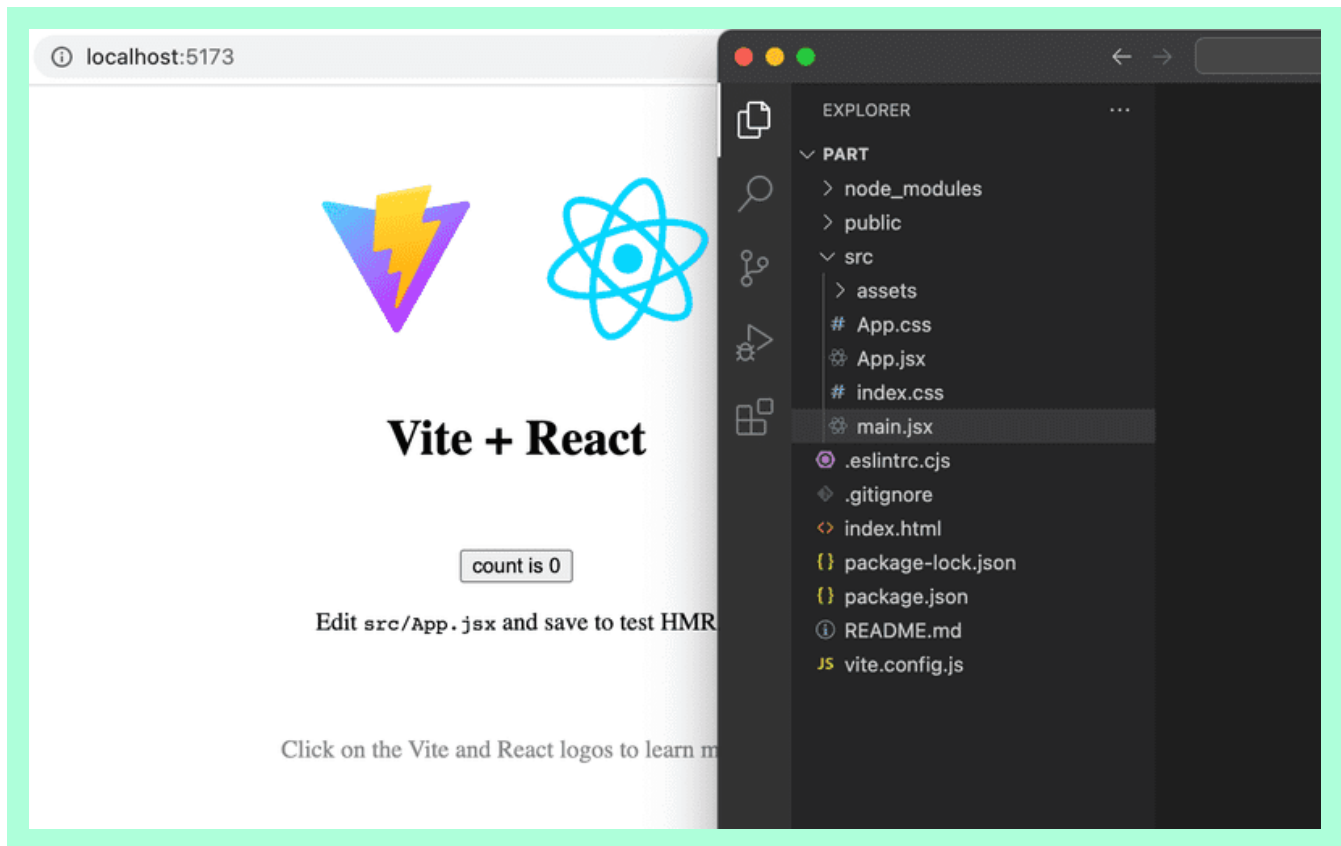
```
npm run dev

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 />)
```

[copy](#)

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.

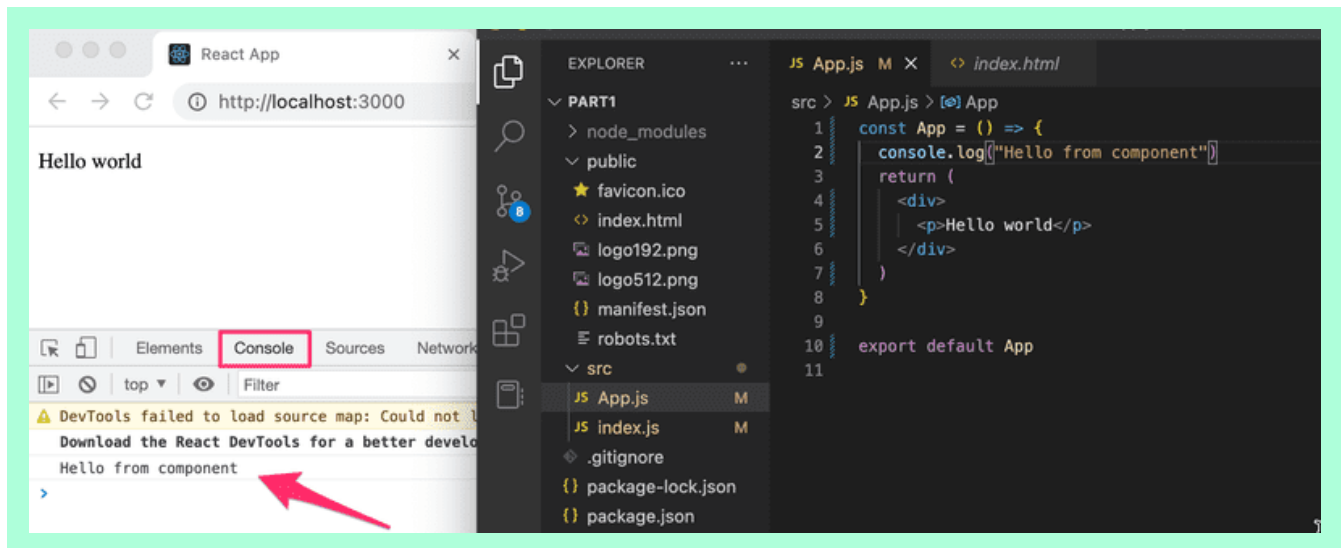
The function defining the component may contain any kind of JavaScript code. Modify your component to be as follows:

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

[copy](#)

```
export default App
```

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>  
  )  
}  
  
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](#)

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.

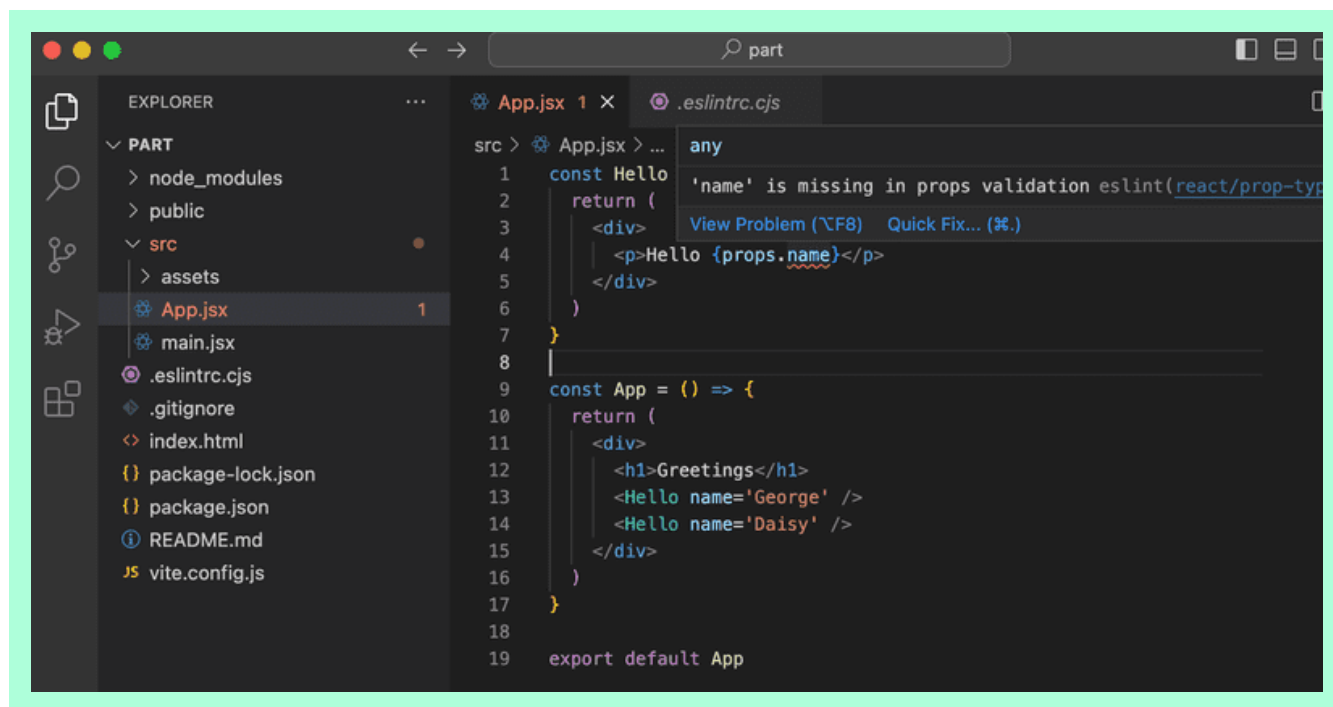
I really hope your console was open. If it was not, remember what you promised:

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 life so much easier.

Possible error message

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



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
  },
},
}

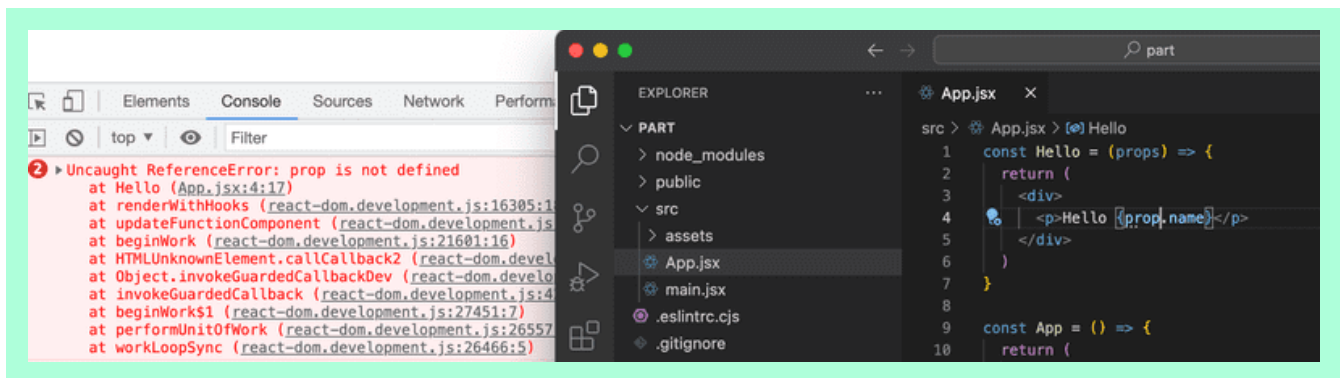
```

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 |     <Hello name='Daisy' />  
15 |
```

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:

```

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, step1

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 extra files *App.css* and *index.css*, and 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, step2

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



UNIVERSITY OF HELSINKI

