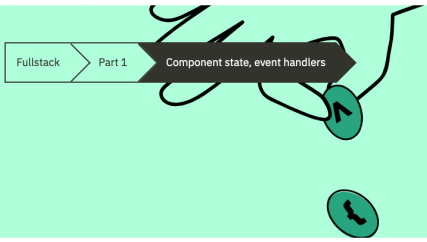


c Component state, event handlers

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>
  )
}
```

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>
  )
}
```

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,
}
```

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>
  )
}
```

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

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>
  )
}
```

If the object we are destructuring has the values

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

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>
  )
}
```

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
```

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 }) => {
```

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
```

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} />
)
```

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
```

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} />
  )
}

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)
```

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 />)
```

and *App.jsx* changes to the following:

```
import { useState } from 'react'

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

  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'
```

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

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

The function call adds *state* to the component and renders it initialized with the value of 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
)
```

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

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

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>
  )
}
```

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)
```

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

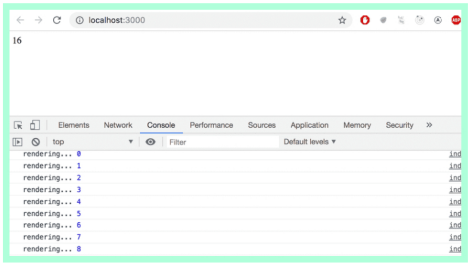
Every time the `setCounter` modifies the state it causes the component to re-render. The value of the state will be incremented again after one second, and this will continue to repeat for as long as the application is running.

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>
  )
}
```

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

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

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

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

By changing the event handler to the following form

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

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

  return (
    <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:

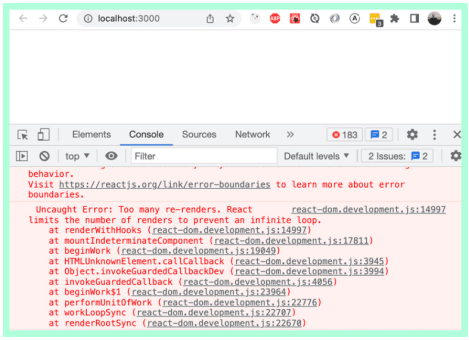
```
<button onClick={() => setCounter(counter + 1)}>                       copy
  plus
</button>
```

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

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

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

This would completely break our application:



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)}>
```

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 rerender...

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

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

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>
  )
}
```

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>
```

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>
  )
}
```

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>
  )
}
```

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

Our *App* component now looks like this:

```
const App = () => {                                            copy
  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>
  )
}
```

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 rerendering

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.

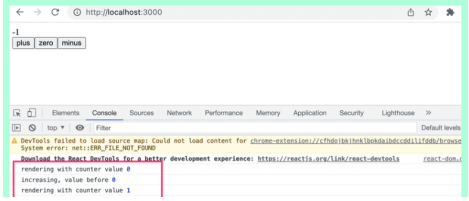
When one of the buttons is clicked, the event handler is executed. The event handler changes the state of the `App` component with the `setCounter` function. **Calling a function that changes the state causes the component to rerender.**

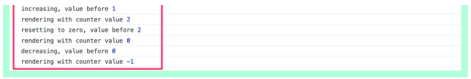
So, if a user clicks the *plus* button, the button's event handler changes the value of `counter` to 1, and the `App` component is rerendered. This causes its subcomponents `Display` and `Button` to also be re-rendered. `Display` receives the new value of the counter, 1, as props. The `Button` components 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 = () => {                                            copy
  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>
  )
}
```

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





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>
  )
}
```

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>
  )
}
```

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>
```

We can simplify the Button component as well.

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

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