React Training

Instructor
Jukka Tupamäki, jukka@lemanse.fi

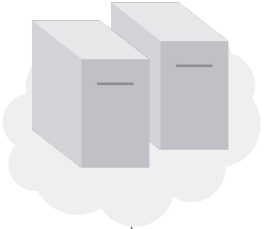
Introduction

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Single page application (SPA)

The URL can be changed via History API. Switching between views and changing the URL in the app does not load any HTML files from the server.

The index.html contains just a couple of HTML elements.

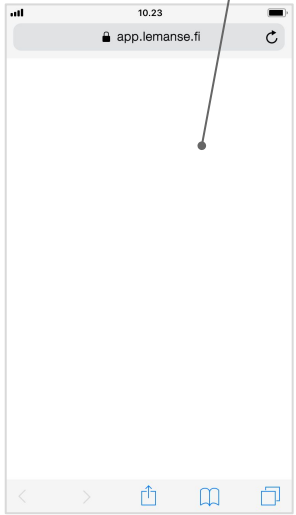


1. HTTP REQUEST:
GET https://app.lemanse.fi/

RESPONSE: **index.html**

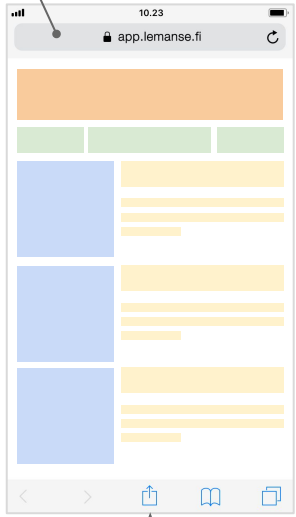
2. HTTP REQUEST:
GET /static/css/main.6c89f91b.css

3. HTTP REQUEST:
GET /static/js/main.38ac2680.js



JavaScript files are read and the application starts.

The application creates the HTML elements it needs.

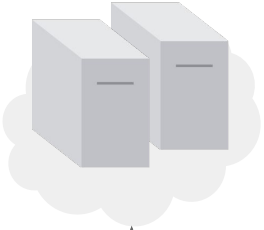


The app communicates with APIs using HTTP requests and / or web sockets.

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Server-side rendering

In this case the index.html contains all required HTML elements.
Server-side rendered apps become visible to search engines.

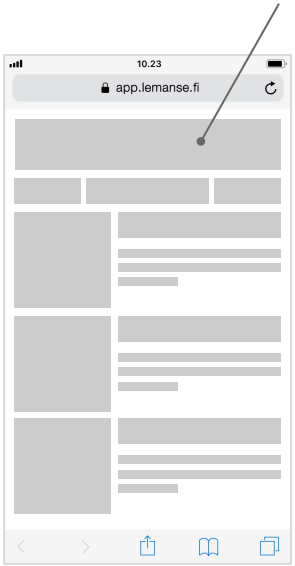


1. **GET https://app.lemanse.fi/**

The initial render happens on the server. The generated HTML is then sent to the browser.

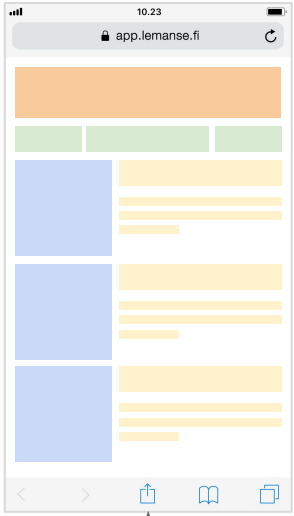
RESPONSE: **index.html**

2. Load all resources defined in HTML
GET /static/css/main.6c89f91b.css
GET /static/js/main.38ac2680.js
GET https://mycdn.com/images/photo1.jpg
GET https://mycdn.com/images/photo2.jpg
...



The JavaScript file is executed and the application starts in the browser.

The app re-uses the HTML elements created by the server to display content faster to the user.



INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

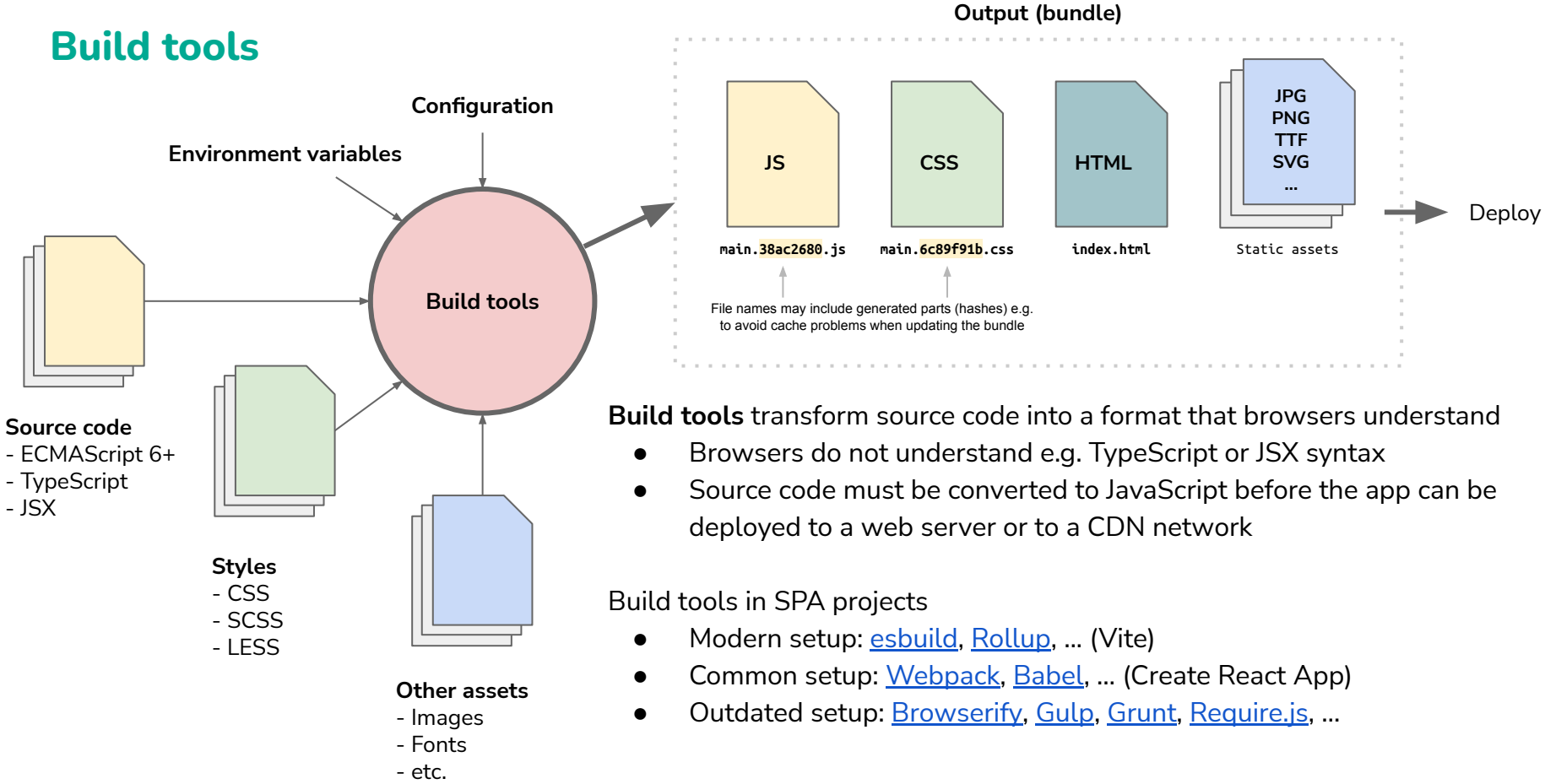
Server-side rendering (SSR)

- Pros
  - **Better visibility in search engines**: server responds with an HTML file that has real content and is not just an empty placeholder
  - **Initial page could load faster**: matters especially in consumer apps e.g. web shops, landing pages, etc.
- Cons
  - **Makes development more complex**: most of the code must work on the server and in the browser.
  - The server environment has limitations that do not exist in real browsers (lack of real DOM, window/document, browser APIs)

- Consider using a framework if SSR sounds useful, e.g. Next.js (https://nextjs.org/)

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Build tools



Build tools transform source code into a format that browsers understand

- Browsers do not understand e.g. TypeScript or JSX syntax
- Source code must be converted to JavaScript before the app can be deployed to a web server or to a CDN network

Build tools in SPA projects

- Modern setup: esbuild, Rollup, ... (Vite)
- Common setup: Webpack, Babel, ... (Create React App)
- Outdated setup: Browserify, Gulp, Grunt, Require.js, ...

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Create React App (CRA)

- A command-line tool for creating React projects from scratch
  - Suits most projects out of the box
  - Removes the need to manually build and maintain the development stack for each project
  - Before CRA you had to choose which tools to use (Browserify, Webpack, Grunt, Gulp etc.) and maintain the configurations yourself (luckily those times are long gone)

- Packed with features and pre-configured tools
  - Documentation https://create-react-app.dev/
  - Uses Webpack, Babel and PostCSS (These tools have been around for a while...)
  - For both JavaScript and TypeScript projects (--template)
  - *Does not support server-side rendering -> Next.js or Vite*

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Useful commands

Create a new project

```
npx create-react-app my-app-name --template typescript
```

Start development server (in project's root)

```
npm start
```

Start test runner (in project's root)

```
npm test
```

Build project (in project's root)

```
npm run build
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Vite

- Documentation & guides https://vitejs.dev/
  - Uses native ES Modules (ESM) instead of bundling files with Webpack
  - ESM is supported in all major browsers since 2017
  - To support older browsers, consider using *@vitejs/plugin-legacy*
- Pros
  - A lot faster than Create React App
  - Uses modern build tools (Rollup, esbuild, etc.)
  - Extendable out of the box (3rd party modules / hacks not required)
- Cons
  - Test framework not included / preconfigured (Try Vitest or wait for Jest's ESM support)

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Vite: Usage

```
npm create vite@latest my-app-name -- --template react-ts   # Create a new React & TypeScript project
cd my-app-name                                              # Switch to the project dir
npm install                                                 # Install dependencies
npm run dev                                                 # Start the dev server
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Project structure

- Full control over project structure
  - CRA, Vite or React do not require a specific structure to exist under `src/`
- Keep related files close to each other
  - `src/components/Counter.tsx`
  - `src/components/Counter.test.tsx`
  - `src/components/Counter.css`
- Avoid too much nesting e.g.
  - `src/components/Counter/index.tsx` (this style creates a lot of index.tsx files)
  - `src/components/Counter/index.test.tsx`
  - `src/components/Counter/style.css`
- Read: https://reactjs.org/docs/faq-structure.html

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Basic project structure

In most cases a React app contains the following parts:

- Components
  - All components are placed in this folder
  - Maybe App.tsx should be there too..? You decide
- Data / state management (Redux)
  - Reducers
  - Action creators
  - Selectors
- Services
  - E.g. i18n, l10n, analytics...
  - HTTP queries to APIs

(Create any structure you need, just remember to keep it simple)

```
src
├── index.tsx
├── App.tsx
├── App.css
├── App.test.tsx
└── components
    ├── Counter.tsx
    ├── Counter.css
    ├── Counter.test.tsx
    ├── User.tsx
    └── ...
└── data
    ├── store.ts
    └── selectors
        ├── users.ts
        └── counter.ts
    └── slices
        ├── users.ts
        └── counter.ts
└── services
    ├── i18n.ts
    ├── analytics.ts
    └── api
        └── users.ts
└── utils
    └── ...
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Browser extensions

- React Developer Tools
  - For inspecting React components: hierarchy, state, props etc.
  - Firefox: https://addons.mozilla.org/en-US/firefox/addon/react-devtools/
  - Chrome: https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi

- Redux Devtools
  - For inspecting Redux actions, store state and state changes
  - Firefox: https://addons.mozilla.org/en-US/firefox/addon/reduxdevtools/
  - Chrome: https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Frameworks

Next.js

https://nextjs.org/

- A framework for building web apps using React (web shops, homepages, etc.)
- Includes advanced features e.g. server-side rendering, image optimization, custom APIs

Gatsby

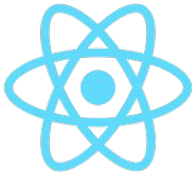
https://www.gatsbyjs.com/

- A framework for building static websites using React
- Generates static HTML files
  - Similar to Hugo, Jekyll, etc. but uses React instead of a template syntax (Handlebars, EJS, etc.)

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

React

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi



https://reactjs.org/

React is a JavaScript library for building UIs

React apps are **declarative** and **component based**

Components use **JSX syntax** instead of plain HTML

Composition over inheritance

React is soon 10 years old. The first public version was published in 2013.

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Environments

- Web apps (React)
  - Single page apps, static sites, etc.

- Hybrid native apps (Electron or similar)
  - These are web apps wrapped in a native app
  - Uses Chromium for running the app but has access to operating system's APIs
  - macOS, Windows and Linux

- Native apps (React Native)
  - iOS & Android: https://reactnative.dev/ (Original)
  - Windows & macOS: https://microsoft.github.io/react-native-windows/ (Microsoft's fork)

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Declarative code (e.g. React)

Declarative code tells what should be done, but not how to implement it.

Less code, higher abstraction.

```
function Link({ href, children }) {
  return <a href={href}>{children}</a>;
}
```

Component's definition does not tell how the 'a' element is created or when it is added into the DOM tree.

```
ReactDOM.render(
  <Link href="https://lemanse.fi">Lemanse</Link>,
  document.getElementById('root')
);
```

↓

```
<div id="root">
  <a href="https://lemanse.fi">Lemanse</a>
</div>
```

App's root is rendered by the ReactDOM library by telling which element to use as a root element. React takes care of the details.

(ReactDOM is invoked only once in an apps lifecycle.)

The output looks the same as the imperative version but how it was achieved is totally different.

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Imperative code (... about 99,5% of the web is coded like this?)

Imperative code uses low-level APIs to do exactly what & how the developer wants.

More code, less abstraction.

```
var href = 'https://lemanse.fi';
var text = 'Lemanse';

var a = $('<a href="' + href + '">' + text + '</a>');

$('#root').append(a);
```

This anchor element is built manually by combining strings. Additionally, you specify exactly where the element is placed in the DOM tree and when.

```
<div id="root">
  <a href="https://lemanse.fi">Lemanse</a>
</div>
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Reconciliation algorithm

A change in component's **state**

A change in component's **props**

Render

Output

```
<a href="https://lemanse.fi">Lemanse.fi</a>
```

Reconciliation

Diffing

Old value

```
<a href="https://reactjs.org">React.js</a>
```

New value

```
<a href="https://lemanse.fi">Lemanse.fi</a>
```

DOM tree update

The attributes and the text of the <a> element are changed but *the element itself does not get removed and re-added.*

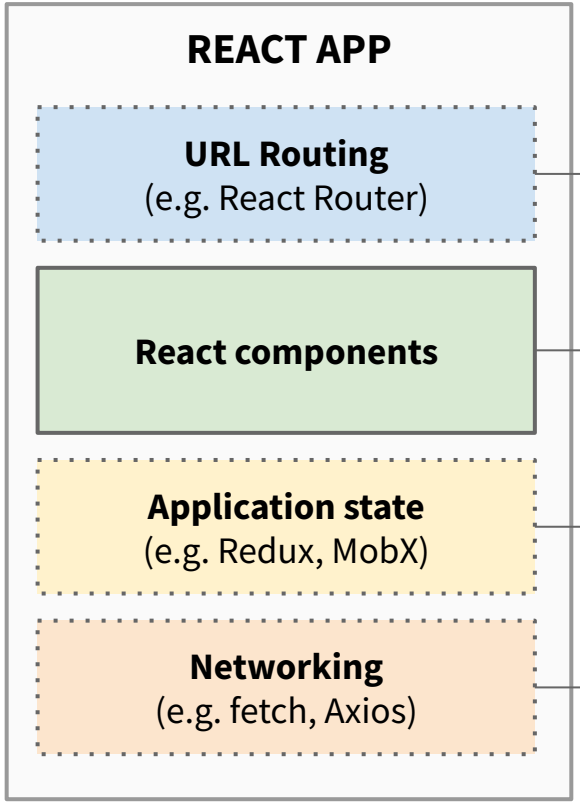
INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Reconciliation algorithm

- A component may render many times during its lifecycle *but the rendered output is not always updated into the DOM tree*
- React's internal reconciliation algorithm decides...
  - ...can existing elements be reused
  - ...which elements get deleted and which are only modified
  - ...which elements need to be created
  - **...whether any changes are required at all**
- "Virtual DOM" in React refers to the reconciliation algorithm
  - The algorithm compares component's output to the values in the DOM (diff)
  - The algorithm is one of the key advances in React regarding frontend apps' performance. Other libraries have since adopted a similar approach.
  - https://reactjs.org/docs/reconciliation.html

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Building blocks



A routing library is needed in order to communicate with the address bar and to provide page structure for the app.

Components are mostly custom-made and have different roles. Components made by others can be downloaded from NPM.

Application state consists of e.g. data downloaded from the server or entered by the user. In a non-trivial app, proper state management is crucial.

Communication with HTTP APIs and Web Sockets is usually handled in a centralized manner via an external library or Fetch API.

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

JSX syntax

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Some HTML attributes are named a bit differently than in HTML

Most **HTML attributes** are written in camelCase

JavaScript values in curly braces (e.g. event handlers)

Static string values can be written without curly braces

React element

```
<div onClick={onClick} className="Button">
    Lorem ipsum dolor sit amet...
</div>
```

Inline text

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

React component

```
<Icon name={iconName} small />
```

Components receive props
instead of attributes

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

```
<div onClick={onClick} className="Button">
  <Icon name={iconName} small /> {text}
</div>
```

Any valid value can be consumed in JSX using curly braces

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Data attributes can be used as well e.g. for identifying elements in test environment

```
<div onClick={onClick} className="Button" data-testid={`button-${id}`}>
  <Icon name={iconName} small /> {text}
</div>
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

A component or an element?

React element

– ***Tag name is always in lowercase***
– Refers to a specific HTML / SVG tag

```
<div className="wrapper">
  Text
</div>
```

JSX transform interprets a lowercase tag name as a DOM element (SVG or HTML).

```
React.createElement(
  'div',
  { className: 'wrapper' },
  'Text'
);
```

React component

– ***Name begins with an uppercase letter***
– The name must be a valid JavaScript variable name
– Use **<UpperCamelCase />** style
– The name refers to a variable in the current scope

```
<Div className="wrapper">
  Text
</Div>
```

JSX transform knows that an uppercase first letter refers to a component variable in the current scope.

```
React.createElement(
  Div,
  { className: 'wrapper' },
  'Text'
);
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

JSX Transform (e.g. Babel)

- *JSX syntax* must be compiled to JavaScript
  - JSX is compiled to React API calls
  - JSX is not supported by browsers
  - **Try it:** https://babeljs.io/repl/

- **Some benefits**
  - Compilation fails if source code contains invalid syntax -> such errors never go into production

```
<div className="wrapper">
  Text
</div>
```

↘

```
React.createElement(
  'div',
  { className: 'wrapper' },
  'Text'
);
```

React 17 has small changes in the JSX transform. The above mentioned works despite the changes.

```
<div className="wrapper">
  Text
</div>
```

↘

```
_jsx(
  'div',
  {
    className: 'wrapper',
    children: 'Text'
  },
);
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Writing JSX

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Writing JSX

- You can...
  - ...assign JSX to a variable
  - ...use any variable within JSX
  - ...pass JSX as an argument to a function
  - ...store JSX in an array or an object for later use
  - ...do anything with JSX that you can do with any other ***value*** in JavaScript/TypeScript!

- Each file that contains JSX must have **a .tsx extension**

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Writing JSX: Allowed values within JSX

Null values	`null`
String values	`'Lorem ipsum'`
Numbers	`3892` `0.230`
React components	`<MyButton text={'My button'} />`
React elements (SVG, HTML, ...)	`<div className="MyButton" />`
A list containing any of the above	`[<MyButton key={0} />, <MyButton key={1} />]`

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Writing JSX: Consuming variables

```
<span className={myClassNames}>Lorem ipsum...</span>;
<div>Text: {text}</div>;
```

Inline text

Value of the **text** variable will be used here

- Use {curlyBraces} to place variables or values into JSX
  - Note, that (static) inline text does not need curly braces around it
  - I.e. values written without curly braces will be outputted as strings

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Writing JSX: Assigning JSX to a variable

```
const links = [
  <a href="https://www.lemanse.fi" key={0}>Lemanse.fi</a>,
  <a href="https://www.reactjs.org" key={1}>React Docs</a>
];

const menu = (
  <div className="menu">
    {links}
  </div>
);
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Writing JSX: Conditional statements

```
const { isVisible, text, value }: { isVisible: boolean, text: string, value: number} = props;

const content1 = (
  isVisible && <div>This becomes visible!</div>
);
const content2 = isVisible ? <div>Visible</div> : <div />;
const content3 = <span>{isVisible && 'Show something'}</span>;
const content4 = <p>{text || 'Default text'}</p>;
const content5 = <p>{text ?? 'Default text'}</p>; // Nullish coalescing operator

let content6;
if (value > 3) {
  content6 = <div>Greater than 3</div>;
} else {
  content6 = <div>Less than 4</div>;
}
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Early return

```
interface User {
  id: string;
  name: string;
}

function User({ data }: { data: User | null }) {
  if (!data) {
    return <></>;
  }

  return <div>Name: {data.name}</div>;
}
```

Return as early as possible if the required data is not available yet

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Writing JSX: Comments within JSX

```
<ul>
  {/*
  <li>List item 1<li/>
  */}
  <li>List item 2</li>
  <li>List item 3</li>
</ul>
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Writing JSX: Transforming data to JSX

```
const data: NavItem[] = [
  {
    id: "bc4f84a9-5bee-4365-bae5-4cda48ba5585",
    label: "Home",
    href: "/"
  },
  {
    id: "75640627-5ede-4f62-8e98-b290636f83f1",
    label: "About",
    href: "/about"
  }
];

const links = data.map(item => (
  <a key={item.id} href={item.href}>
    {item.label}
  </a>
));
```

- Dynamically (i.e. in a loop) created JSX structures must include a unique *key* prop in the root element of the structure
- Use e.g. database IDs for keys
  - The key prop helps React with its internal book keeping so that React knows which item you are trying to refer to even if the list's order changes.
  - If the key prop is not unique (e.g. you use a list index as a key), the created structure likely acts weird if the list's items are sorted differently (e.g. click handlers point to wrong data)

Read more: https://reactjs.org/docs/lists-and-keys.html

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Fragments

```
function MyComponent() {
  return <>
    <span>Value 1</span>
    <span>Value 2</span>
  </>;
}
```

Fragment starts

Fragment ends

- Fragment is "a tag without a tag name"
- Fragments are not visible in the DOM tree
- Fragments are used when you want to return adjacent elements and avoid creating a dummy wrapper element
- Fragments can be created in loops but you have to use **React.Fragment** and you must provide a unique key prop

```
function MyComponent({ items }) {
  return items.map((item, i) => {
    <React.Fragment key={item.id}>
      <span>Item {i}: {item.value1}</span>
      <span>Item {i}: {item.value2}</span>
    </React.Fragment>;
  });
}
```

```
<span>Item 0: value 1</span>
<span>Item 0: value 2</span>
<span>Item 1: value 1</span>
<span>Item 1: value 2</span>
<span>Item 2: value 1</span>
<span>Item 2: value 2</span>
```

Fragments

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Download example code / templates: Day 1 / 2

1. Download: http://files.lemanse.fi/react/react-training-examples-day-1.zip
2. Extract the file to your React project directory
3. Place folders in src/
   a. src/hooks
   b. src/components
4. Open the project in Visual Studio Code

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Exercise: Map data to JSX

Create a menu component that lists some links

1. Locate the component file at: **src/components/Menu.tsx**
2. Place the Menu component into the App component so that you can see it
3. Map the list items to **<a>** elements
4. Remember to define the **key** prop

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

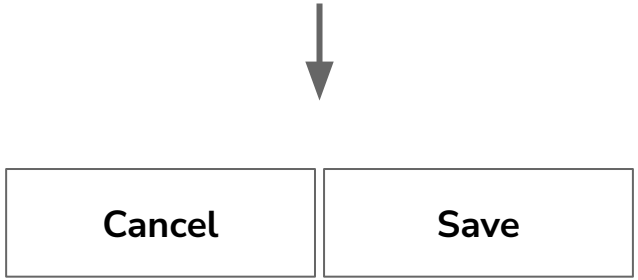
Components

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Components

```
import "./Button.css";

type ButtonProps = {
  onClick: () => void;
  label: string;
}

function Button({ onClick, label }: ButtonProps) {
 return (
   <button className="Button" onClick={onClick}>
     {label}
   </button>
 );
}

export default Button;
```

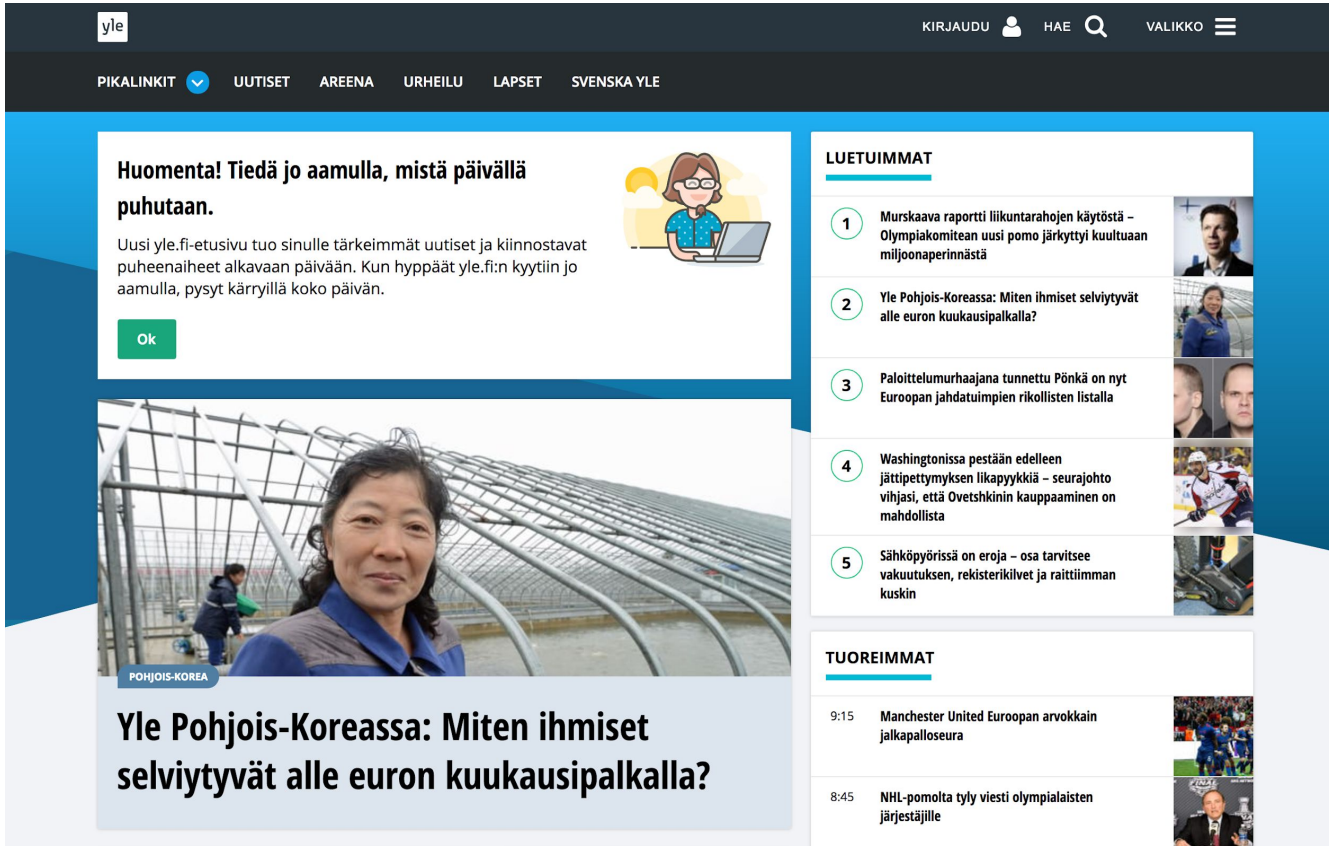
```
<div>
  <Button onClick={handleSave} label="Save" />
  <Button onClick={handleCancel} label="Cancel" />
</div>
```



The basic idea is reusability. But not all components have to be reusable. Start small and generalise when you know the use cases but don't start with the generalised version.

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

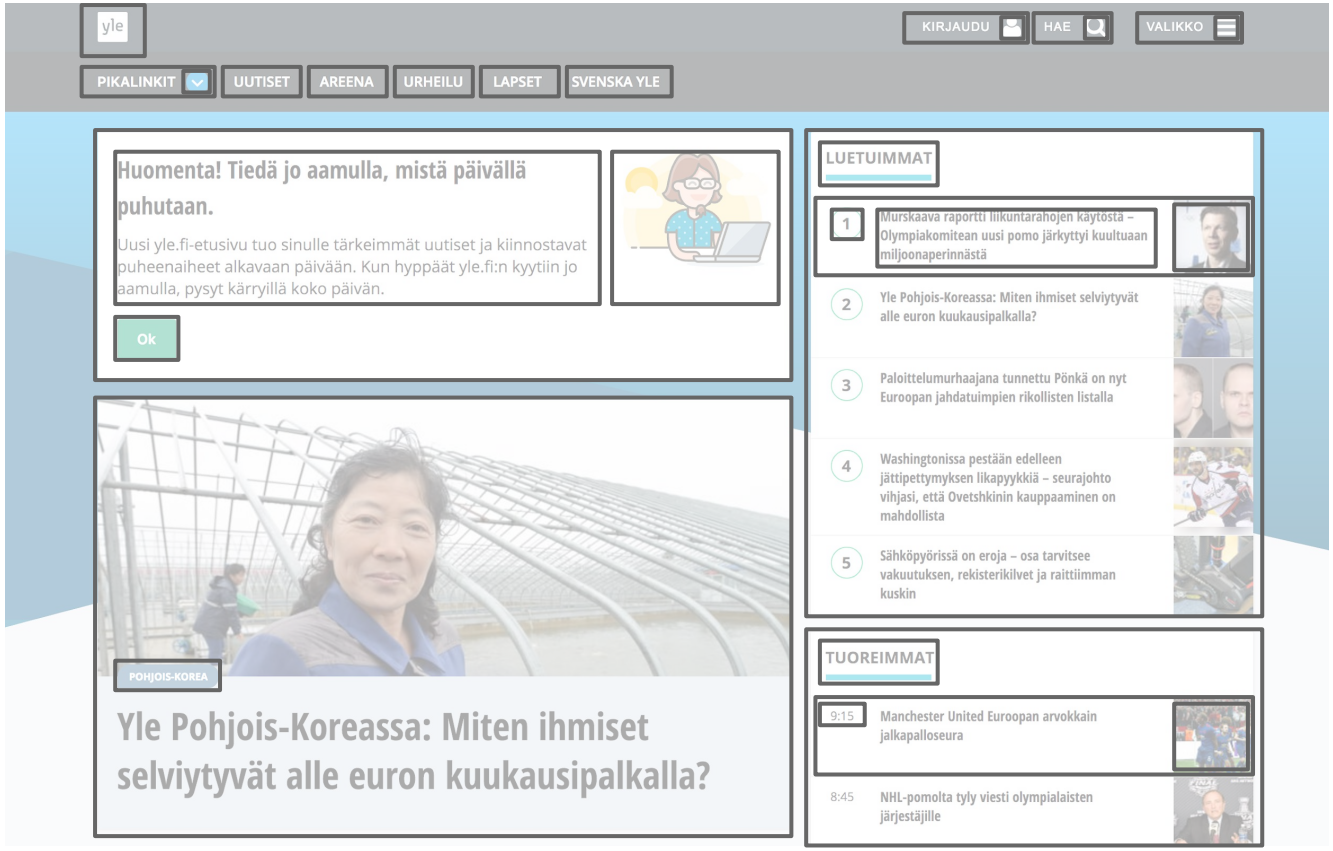
Turning a layout into components



Screenshot: yle.fi

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Turning a layout into components



Screenshot: yle.fi

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Component renders usually without real data on the first time

Component is rendered again when the data is received from the server

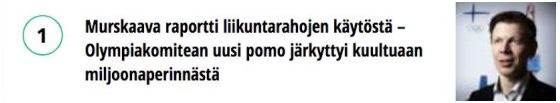
Input
e.g. Data structure, simple values, complex values, other components, ...

```
const newsItem = {
  id: null,
  position: 0,
  title: '',
  image: null
};
```

```
const newsItem = {
  id: '79e3a944-105b-4a02-9ee7-c8f762c6c4f0',
  position: 1,
  title: 'Murskaava raportti ...',
  image: 'https://cdn.com/img/123.jpg'
};
```

Component

Output
e.g. an HTML structure, a new component, ...



INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Components

- A component usually produces something visible (not always)
  - E.g. a set of HTML elements or just a single button
  - Most components are **re-usable**, but not all have to be

- React does not include any UI component library that implements a specific UI style out of the box

Bootstrap: https://github.com/react-bootstrap/react-bootstrap
Material UI: https://www.material-ui.com/
Ant Design: https://github.com/ant-design/ant-design/
Tailwind CSS: https://tailwindcss.com/
More: https://github.com/enaqx/awesome-react#react-component-libraries

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Composition

- Components can be re-used by **composition**
  - Composition is one of React's and JSX's core concepts
  - Composition is simply using an existing component as a part of a new component
  - Simple components with less logic are easier to re-use
  - Recursive components e.g. tree structures can be created easily through composition

- Designing a reusable component takes time and effort
  - **Common factors:** How the component is used in different contexts?
  - **Abstraction:** Which details are allowed to be controlled with props?
  - Bad generalisation causes a *leaky abstraction* and the component becomes hard to use
  - Be prepared to refactor the component several times to find a proper generalisation
  - *Not all components have to be reusable. Avoid overthinking!*

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Composition

```
class Button extends Component {
  render() {
    return (
      <button type={this.props.type}>
        {this.props.children}
      </button>
    );
  }
}
```

When similar pieces of HTML/JSX is needed here and there, copy-paste is usually the first thing that pops into mind. This kind of code can easily be refactored by taking the copied code into a new component.

```
<Button type="submit">Submit</Button>
...
<Button type="submit">Submit</Button>
...
<Button type="submit">Submit</Button>
...
<Button type="submit">Submit</Button>
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Composition

Make sure you always extend the Component base class that React provides.

```
class SubmitButton extends Component {
  render() {
    return <Button type="submit">Submit</Button>;
  }
}
```

Here is the copy pasted code brought into a new component. If needed, pass any dynamic / changing values as props to the new component.

Composition is about using other components (either your own or from npm) as part of a new component.

Any changes made in SubmitButton will be visible in all places where the component is used, and you don't have to modify multiple places.

```
<SubmitButton />
...
<SubmitButton />
...
<SubmitButton />
...
<SubmitButton />
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Inheritance is not allowed

Do not do this! You must not *inherit* other components than Component. *Inheritance* and *composition* do not work similarly.

```
class SubmitButton extends Button {
  // ...
}
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Component definitions

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Component definitions: classes and functions

Class component (legacy)

```
class Hello extends Component {
  render() {
    return <div>Hi!</div>;
  }
}
```

Function component (preferred)

```
function Hello() {
  return <div>Hi!</div>;
}
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Class component

Types for props and state

Default values for optional props

Initial state
(state contains private data that others cannot access or modify directly)

Lifecycle methods

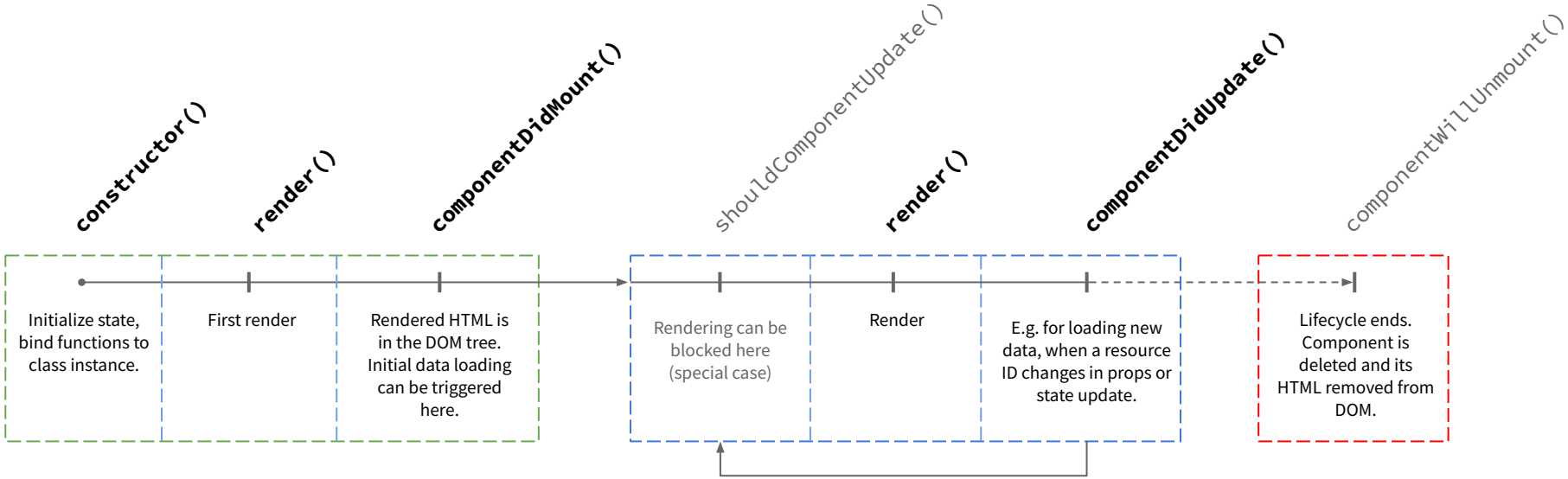
A custom member function defined using an arrow function and a member variable

State, props and methods are accessed via *this*

```
import { Component } from "react";

type HelloProps = { name?: string; }
type HelloState = { greeting: string; }

class Hello extends Component<HelloProps, HelloState> {
  static defaultProps: HelloProps = {
    name: "Unknown",
  };

  state: HelloState = { greeting: "Hello" };

  componentDidMount() {
    console.log('Mounted');
  }

  hi = () => this.setState({ greeting: "Hi!" });

  render() {
    const { greeting } = this.state;
    const { name } = this.props;

    return (
      <div onClick={this.hi}>
        {greeting}, {name}!
      </div>
    );
  }
}

export default Hello;
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Lifecycle methods in class components



1. Mounting
Component is ***instantiated*** and ***rendered*** for the first time.

2. Updating
Component is ***re-rendered*** always when ***state*** or ***props*** get updated.

3. Unmounting
Component's instance will be ***removed*** from the DOM and garbage collected after a while

More lifecycle methods for special cases, see this link:
https://reactjs.org/docs/react-component.html#the-component-lifecycle

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Lifecycle methods in class components

- Lifecycle methods allow controlling the component
  - Common use-cases
    - Load new data when an ID value changes (because URL changed...)
    - Re-initialize state when a specific value in props changes
    - Integrate with a non-React library, e.g. D3.js or Highcharts
  - Special use-cases
    - Rendering optimizations when props or state change often
    - Blocking rendering after initial render (non-React libs)
  - Some methods are familiar from object-oriented programming
    - **Constructor = constructor()**
    - **Destructor = componentWillUnmount()**

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Function component

Types for props and state

Default values for optional props

Use hooks for defining state

Use hooks for side-effects (e.g. network requests) and as *a replacement for lifecycle methods* (more on this later)

Custom functions can be defined within the component

```
import { useEffect, useState } from "react";

type HelloProps = { name?: string; };
type Greeting = string;

function Hello({ name = "Unknown" }: HelloProps) {
  const [greeting, setGreeting] = useState<Greeting>("Hello");

  useEffect(() => {
    console.log("Mounted");
  }, []);

  function hi() {
    setGreeting("Hi");
  }

  return (
    <div onClick={hi}>
      {greeting}, {name}!
    </div>
  );
}

export default Hello;
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Many ways to do the same thing

```
function Hello(props: HelloProps) {
  return <div onClick={props.hi}>
    {props.greeting}
  </div>;
}
```

```
const Hello = (props: HelloProps) => {
  return <div onClick={props.hi}>{props.greeting}</div>;
};
```

```
function Hello({ greeting, hi }: HelloProps) {
  return <div onClick={hi}>
    {greeting}
  </div>;
}
```

```
const Hello = ({ greeting, hi }: HelloProps) => {
  return <div onClick={hi}>{greeting}</div>;
};
```

```
const Hello = ({ greeting, hi }: HelloProps) =>
  <div onClick={hi}>{greeting}</div>;
```

All of these are equivalent e.g. performance-wise. What matters is consistency: pick a style you like and stick with it.

Hooks cannot be used in one-liners.

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Definition and instantiation

1. These are just definitions, zero instances created yet

```
class MyComponent extends React.Component {
  render() {
    return <div>{this.props.text}</div>;
  }
}

function MyComponent(props) {
  return <div>{props.text}</div>;
}

const MyComponent = ({ text }) => {
  return <div>{text}</div>;
};
```

2. Once angle brackets are used, an instance is created (doesn't guarantee that it renders though)

```
<MyComponent text="Lorem ipsum" />
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Component's files

- **Filenames must match with the component's name**
  - Reasons: maintains consistency, easier to read the codebase, ...
  - **Rule of thumb:** Only one exported component in one file!
- 2-3 files per component
  - MyComponent**.tsx**
  - MyComponent**.test.tsx**
  - Stylesheets:
    - MyComponent**.css**
    - MyComponent**.module.css** (CSS Modules)
    - MyComponent**.scss** (See: https://create-react-app.dev/docs/adding-a-sass-stylesheet)

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Props

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Props

```
<MyLink href="https://lemanse.fi" title="Lemanse.fi">Lemanse.fi</MyLink>
```

These are ***props***. Props are just like function's parameters. All props and their values are passed to the component as is.

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Props

```
<MyLink href="https://lemanse.fi" title="Lemanse.fi">Lemanse.fi</MyLink>
```

Static string values do not require curly braces around the value

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Props

```
<MyLink href="https://lemanse.fi" title="Lemanse.fi">Lemanse.fi</MyLink>
```

Any value given between the start and the end tag becomes available in the **children** prop

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Props

```
const title = 'Lemanse.fi';
const href = 'https://lemanse.fi';
const myLink = <MyLink href={href} title={title}>Lemanse.fi</MyLink>;
```

Props' values can be defined using variables

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Props

```
<MyLink href="https://lemanse.fi" title="Lemanse.fi" disabled>
  Lemanse.fi
</MyLink>
```

This is a **boolean** prop. If the name of the prop is given (no value), it's value becomes **true**.

Otherwise the value is **undefined** or a default prop value is used if such exists.

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Props

- ***Props are read-only***
  - One-way data binding
  - TypeScript does not guarantee immutability

- Props can transmit any value to a component, e.g.
  - Other components
  - JSX
  - Functions, objects, arrays (objects are passed by reference)
  - Primitive values: strings, numbers, booleans (copied, immutable by default)

- How props are accessed in the component depends on how the component is defined:
  - Class components: **this.props**
  - Function components: **the first parameter** contains all the props

- Reserved prop names **key** and **ref**: https://reactjs.org/warnings/special-props.html

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Props: Accessing props in a class component

```
const title = 'Lemanse.fi';
const href = 'https://lemanse.fi';
const myLink = <MyLink href={href} title={title}>Lemanse.fi</MyLink>;

class MyLink extends Component<MyLinkProps> {
 render() {
   const { href, title, children } = this.props;
   return <a href={href} title={title}>
     {children}
   </a>;
 }
}
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Props: Accessing props in a function component

```
const title = 'Lemanse.fi';
const href = 'https://lemanse.fi';
const myLink = <MyLink href={href} title={title}>Lemanse.fi</MyLink>;
```

The props object is the first parameter

```
const MyLink = (props: MyLinkProps) => (
  <a href={props.href} title={props.title}>{props.children}</a>
);
```

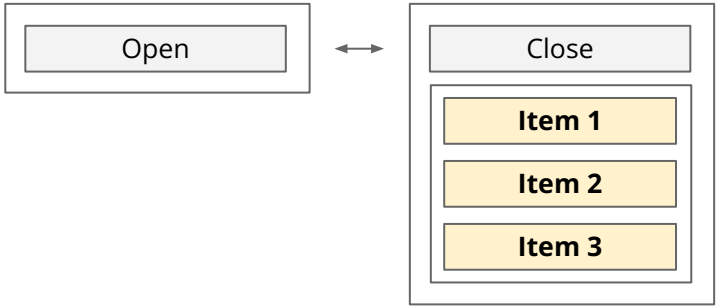
Destructuring can be used to avoid repetition

```
const MyLink = ({ href, title, children }: MyLinkProps) => (
  <a href={href} title={title}>{children}</a>
);
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Children prop

```
<ExpandableList>
  <ListItem>Item 1</ListItem>
  <ListItem>Item 2</ListItem>
  <ListItem>Item 3</ListItem>
</ExpandableList>
```



- **props.children** is a special prop that contains all child elements
  - **You can place any valid value between the opening and closing tags**
  - The children prop is like any other you can use your own props for the same purpose
  - Proper type for children is: `React.ReactNode`
  - Read more: https://reactjs.org/docs/react-api.html#reactchildren
    - React.Children - contains utility functions for dealing with the children prop
    - React.cloneElement() - for manipulating child elements' props

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Exercise: Children

1. Open src/components/Children.tsx
2. Place the component in the App component
3. Write a couple of child elements for the Children component
4. Notice the types

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

One-way data binding

```
const data = {
  value: 1
};
```

```
<Child1 data={data} />
```

```
<Child2 data={data} />
```

```
data.value = 2;
```

```
data.value = 6;
```

Assigning new values mutates the data structure but the change ***won't be automatically updated*** to the components that use the data.

- Props can be used for passing data to child components but not back to parents
  - If a prop value is changed, the change will not be updated to other components
  - Data might change in the data structure but nobody will notice the change!
  - How to make it work properly:
    - Move the state up in the component hierarchy and provide functions for updating it
    - Use a state management library e.g. **Redux**

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Communicating with a parent component

2a. Update data using the provided update function

```
// Parent component owns state / data and
// provides functions for modifying it.

state = {
  value: 0
};

update = (newValue) => {
  this.setState({
    value: newValue
  });
}
```

```
<Child1 value={value} update={update} />
```

```
update(2);
```

1. Data and updater functions passed as props.
(Updated data goes through this as well.)

```
<Child2 value={value} update={update} />
```

```
update(6);
```

2b. Update data using the provided update function

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Hooks

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Hooks

```
const [value, setValue] = useState(0);
const [value, setValue] = useState<number>(0);
```

- Hooks allow adding all kinds of logic to function components
  - Hooks are a good tool for creating reusable features
  - Hooks have class component's lifecycle features built-in
  - ***Hooks cannot be used in class components!***

- All hooks begin with the word ***use***
  - **use**State, **use**Effect
  - Custom hooks follow this as well: **use**Counter, **use**SpecialFeature

Docs: https://reactjs.org/docs/hooks-reference.html

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

useState() - State in function components

```
import { useState } from "react";

function Counter() {
  const [value, setValue] = useState(0);

  return <div onClick={() => setValue(value + 1)}>
    {value}
  </div>;
}
```

Built-in hooks, such as useState, are included in the 'react' module

1 Initialize the state container with data

2 Destructuring helps obtaining the values returned by useState. **The first item is the value**, and **the second is an updater function**.

3 Use the updater function to update the state's value

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

useState() - State in function components

```
const [loading, setLoading] = useState(false);
const [failed, setFailed] = useState(false);

const [user, setUser] = useState<User | null>(null);
```

The useState hook can be used multiple times in the same component. All created state containers are independent.

useState can be initialized with any value. Type is inferred from the initial value if such exists.

```
setUser({
  id: 'abc',
  name: 'Frank',
  email: 'developer@company.com'
});
```

```
setUser({
  ...user,
  name: 'Joe'
});
```

...when updating objects / arrays you must take care of properly merging the new value into the old value.

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Exercise: Counter component

1. Open **src/components/Counter.tsx**
2. Write a **function component** called Counter:
   a. Create a type for the props (CounterProps):
      - initialValue (number)
      - step (number, optional)
   b. Add a default value for the **step** prop: 1
3. Create elements
   a. Add a div element to the root of the component
   b. Add **increment (+)** and **decrement (-)** buttons and **a div element for displaying the value**
   c. Use the CSS classes defined in **Counter.module.css**: **root**, **value**, **button**, **increment**, **decrement**
   d. **Import** the component in the App component and use it
4. Initialize counter's value to e.g. 123 and display it
5. Write **onClick handlers** for the + and - buttons, and assign them to the button elements

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

useEffect() - For side-effects, HTTP requests, timers, etc.

```
import React, { useState, useEffect } from "react";

function Counter() {
  const [value, setValue] = useState(0);

  useEffect(() => {
    // Side-effect producing code here

    return () => {
      // Clean-up here
    };
  }, []);

  return <div onClick={() => setValue(value + 1)}>
    {value}
  </div>;
}
```

1. Define a function that runs the side-effect
2. If the function needs clean-up, the returned function runs when the component unmounts
3. If the function uses values from upper scopes, they must be listed here as dependencies.

An empty array means that the function is executed only once (compare to componentDidMount() in classes).

If the array is left out, it means that the function is executed on every render.

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

useEffect() - Producing side-effects

```
import React, { useState, useEffect } from "react";

function Counter() {
  const [value, setValue] = useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      setValue(value + 1);
    }, 1000);

    return () => clearInterval(id);
  }, [value]);

  return <div onClick={() => setValue(value + 1)}>
    {value}
  </div>;
}
```

Updates the value in 1 second intervals using **setValue** and the previous **value**. The timer's ID returned by setInterval() must be stored.

When the component is unmounted, this function is called. The timer must be deleted in order to stop it. Otherwise it will leak memory.

value must be listed as a dependency as it is used by the function's implementation.

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Exercise: Re-initialize the counter on props change

The **initialValue** prop should be updateable. The prop itself can be updated in a parent component but how to react to the change?

1. Open the **Counter.tsx** file
2. Write a **useEffect** hook with a callback function and a dependency list
   a. The hook should run only when initialValue changes
   b. The hook should re-initialize the counter's value to what initialValue contains

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

useEffect() - Dependency list and comparison to classes

<u>If the dependency list is not defined</u>, the function is <u>called every time</u> the component updates. It is the same as calling the function once in a class component's componentDidMount() and on every render in componentDidUpdate().

```
useEffect(() => {
  console.log('Called on every render');
});
```

If the dependency <u>list is empty</u> (but defined), the function is <u>called only once</u>. This is the same as calling the function once in a class component's componentDidMount().

```
useEffect(() => {
  console.log('Called only on first render');
}, []);
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

useEffect() - Dependency list and comparison to classes

If the dependency list contains one or more variables, the function is called on the first render and when any of the values change. This is the same as calling the function once in componentDidMount() and conditionally in componentDidUpdate() when any of the values change.

```
useEffect(() => {
  console.log('Called on first render and if `value` changes', value);
}, [value]);

useEffect(() => {
  console.log('Called on first render and if `value` or `otherValue` changes', value, otherValue);
}, [value, otherValue]);
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

useEffect() - Return value

If the callback function returns a function, the returned function is called when the component unmounts. This is where you can clean up timers, on going requests etc. that would otherwise cause a memory leak. Similar to the componentWillUnmount() lifecycle method.

```
useEffect(() => {
  console.log('Called on first render and if `value` changes', value);

  return () => console.log("Unmounted");
}, [value]);
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Rules of hooks

Hooks can be used in other hooks
(the "use" naming convention)

```
function useRandomValue(intervalMs = 1000) {
  const [value, setValue] = useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      setValue(Math.random());
    }, intervalMs);

    return () => clearInterval(id);
  }, [intervalMs]);

  return value;
}
```

Hooks can be used in
function components

```
function MyComponent() {
  const randomValue = useRandomValue(2000);
  return <div>{randomValue}</div>
}
```

```
class MyComponent extends Component {
  render() {
    const randomValue = useRandomValue(2000);
    return <div>{randomValue}</div>;
  }
}
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Rules of hooks

Hooks cannot be placed inside control structures. This produces a linter error.

```
if (isVisible) {
  useEffect(() => {
    loadData();
  }, [loadData]);
}
```

To fix the problem, move the control structure inside the callback function

```
useEffect(() => {
  if (isVisible) {
    loadData();
  }
}, [loadData, isVisible]);
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Rules of hooks

- Hooks can be used in **function components** and in **other hooks**
  - Hooks cannot be used in functions that do not obey the naming convention
  - A hook must be called at the root level of the component or inside another hook
  - Hooks do not work in class components

- A hook must not be surrounded with a control structure
  - if, for, while etc.
  - Place the control structure inside the hook function

See https://reactjs.org/docs/hooks-rules.html

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Storing state in a function component - useState, useEffect, useCallback

```
function Counter({ initialValue = 0 }) {
  const [value, setValue] = useState(initialValue);

  useEffect(() => {
    setValue(initialValue);
  }, [initialValue]);

  const increment = useCallback(() => {
    setValue(value + 1);
  }, [value]);

  const decrement = useCallback(() => {
    setValue(value - 1);
  }, [value]);

  return (
    ...
  );
}
```

- **useState()** is initialized with an initial value
- **useEffect()** takes care of setting the initial value if it updates (the same as componentDidUpdate())
- **useCallback()** creates event handlers for updating the state. Without useCallback(), the functions would be created again on every render. When useCallback() is used, the functions will be updated only if the dependencies update.

See https://gist.github.com/tukkajukka/5637d8280fef9ea6c3fdd433a63ecee3

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Event handling

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Event handling

```
function Counter() {
  const [counter, setCounter] = useState(0);

  function increment() {
    setCounter(counter + 1);
  }

  return <div onClick={increment}>{counter}</div>;
}
```

In function components, event handlers are usually defined within the function component if not received through props.

If you pass the function to a child component, remember to use useCallback() to avoid creating a new function instance on every render.

```
const increment = useCallback(() => {
  setCounter(counter + 1);
}, [counter]);
```

List of all supported events:
https://reactjs.org/docs/events.html#supported-events

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Event handling

```
class Counter extends React.Component {
  state = { counter: 0 };

  clicked = () => {
    this.setState({
      counter: this.state.counter + 1
    });
  }

  render() {
    return <div onClick={this.clicked}>{this.state.counter}</div>;
  }
}
```

If you don't need the event object, just leave it out.

To register an event handler, use props. Do not call the event handler at this point. One element can listen to as many events as needed e.g. onClick, onMouseOver, ...

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Types

- React's built-in types include e.g.
  - Base type for all events: **SyntheticEvent**
  - Click / tap / multitouch: **MouseEvent**, **PointerEvent**
  - Forms: **ChangeEvent**, **FocusEvent**, **FormEvent**
  - See: https://react-typescript-cheatsheet.netlify.app/docs/basic/getting-started/forms_and_events

- Use type parameters to specify the HTML element type
  - `React.MouseEvent<HTMLDivElement>`

- In inline functions, the HTML element type is inferred from the element (div)

```
(parameter) e: React.MouseEvent<HTMLDivElement, MouseEvent>
<div onClick={e => console.log(e.currentTarget.style.width)}>
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Forms

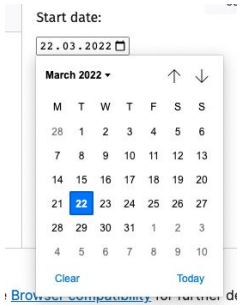
INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

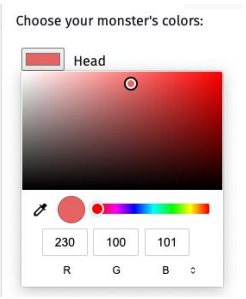
Controlled component

- A controlled component (or its user) is responsible for everything related to the implementation:
  - State management
  - Input handling and validation
  - UI implementation
  - ...
- Usually means a component that gives full control over what is happening within the component
  - E.g. a form field that is implemented entirely in React and does not rely on any browser features
- With UI libraries, you probably need to use **refs** in order to get full access to the implementation / HTML elements

Uncontrolled component

- An uncontrolled component is controlled by a third party e.g. the browser or a UI library
  - Usually gives only a handful of parameters for changing the component's behavior
  - Internals of such component cannot be accessed (state, UI, ...)
- Some examples
  - **A basic HTML input element**: various types exist e.g. datepicker, color, email, number. In all cases the browser handles state changes, validation etc. The elements value is read when the form is submitted.
  - A 3rd party charting library e.g. D3.js





INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Controlled input field

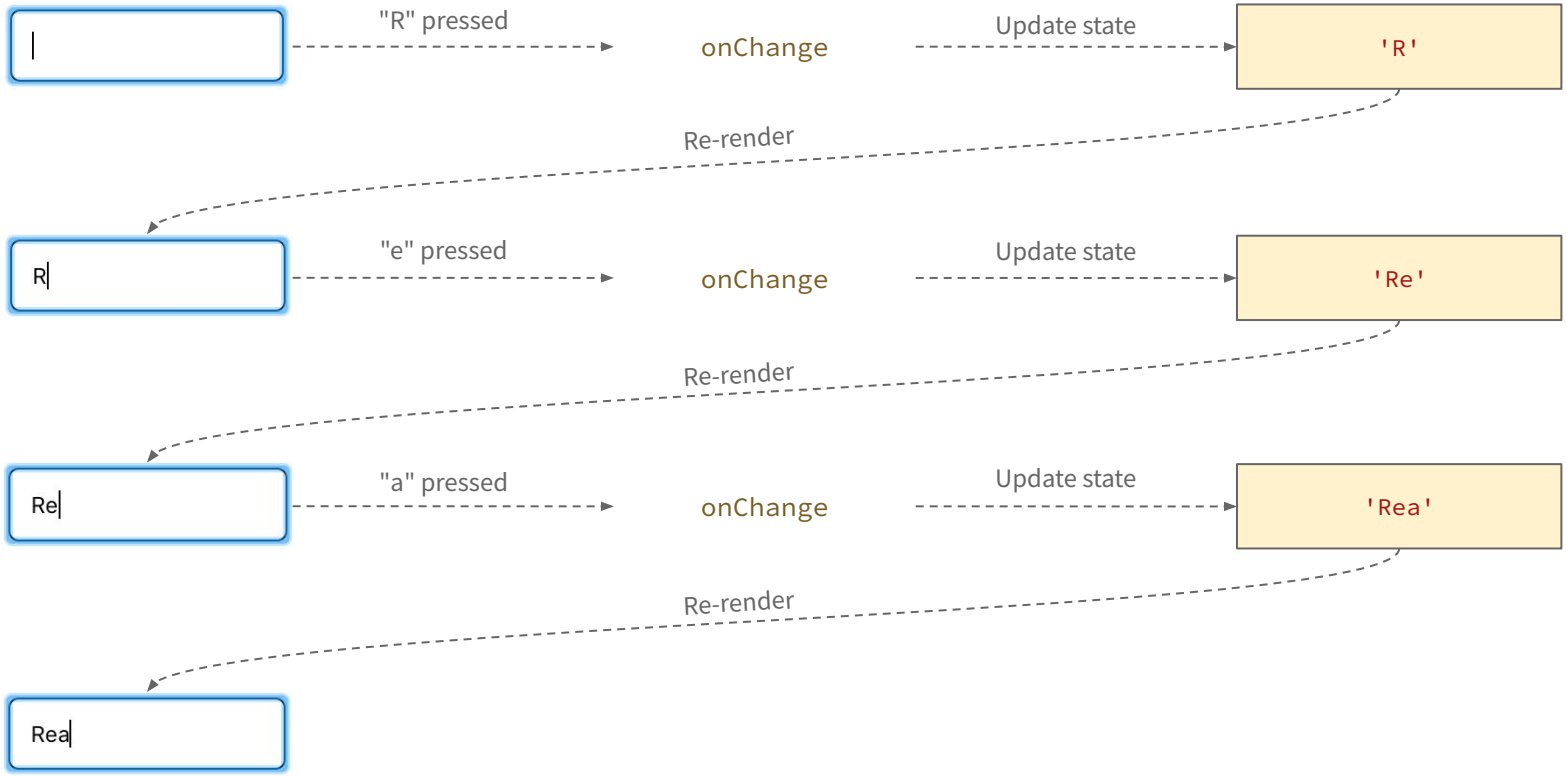
```
function Form({ onSubmit }: { onSubmit: (name: string) => void }) {
  const [name, setName] = useState("");

  return (
    <form
      onSubmit={(event) => {
        event.preventDefault();
        onSubmit(name);
      }}
    >
      <input
        name="name"
        value={name}
        onChange={(event) => setName(event.currentTarget.value)}
      />
      <button>Submit</button>
    </form>
  );
}
```

Input's value is stored in state

All changes must go through an onChange handler once the value prop is given. Each change re-renders the component with the updated values.

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Typing in the input field circulates the input's value through component's state back to the DOM



INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Exercise: Form with controlled input fields

1. Open **src/components/Form.tsx** and place it in the App component
2. Define types
   a. FormProps for the component's props
      - **onSubmit** (function)
   b. FormState for the form's state (an object with the following keys)
      - **name** (string), **email** (string), **phone** (string)
3. Create a state container for the form's state using the **useState** hook
   a. Remember to use **a type parameter** (FormState)
4. Use a <form> element as the root element of the component
   a. Create an onSubmit handler that **1)** prevents the default form action and **2)** calls the onSubmit prop
5. Add <input> elements for all fields defined in FormState
   a. Set **name** and **value** props for each field (try typing in the fields at this point, what happens?)
   b. Add **onChange** handlers. How would you update the state?
6. Write an updater function that can update any keys available in FormState
   a. Signature: update(fieldName: keyof FormState, value: string) => void
   b. Remember to preserve earlier form values, how?
7. Input elements can be components too: try using Input.tsx for a couple of fields

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Form libraries

- React Hook Form
  - A form library that encourages use of uncontrolled components (plain HTML inputs)
  - Works with hooks so class components are not supported
- Formik (unmaintained)
  - Used to be a good library but hasn't been updated in a while
  - Can be used with class components
- React Final Form (unmaintained)
  - Successor of Redux Form which is unmaintained as well

If you don't use a library you will eventually notice you have built one...

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

CSS styles

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

CSS without extras (i.e. vanilla CSS)

MyComponent.css

```
.MyComponent { width: 200px; }
.MyComponent-title { font-weight: bold; }
.MyComponent-menu { display: flex; }
```

MyComponent.tsx

```
import "./MyComponent.css";

function MyComponent() {
  return (
    <div className="MyComponent">
      <div className="MyComponent-title">...</div>
      <div className="MyComponent-menu">...</div>
    </div>
  );
}
```

- Prefix CSS class names with the component's name
  - ...to make it easier to see which component uses them
  - By doing this, it is harder to accidentally override other CSS classes used by other components
- *Works only if component names are unique*
  - e.g. two Title components could get similarly named styles

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

CSS Modules

```
// With CSS Modules, styles are imported into a variable
import styles from './MyComponent.module.css';

// The styles variable contains dynamically generated CSS class names:
<div className={styles.title}></div>

// If the CSS file is in path src/MyComponent.module.css, the final class name looks like this:
<div class="src__MyComponent-module___title"></div>
```

- With CSS Modules, each CSS file has its own namespace
- Now you can use the same class name in multiple files
  - `.title { font-weight: bold; }`
  - `.menu { display: flex; }`
  - `.menu-item { flex: 1; }`

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Styled Components

- Each "Styled Component" implements one HTML tag and its styles
- *Styles can be controlled with props*
- Styled Components takes care of creating and updating CSS styles
- Based on the *tagged template* syntax

Installation (e.g. Create React App project):

```
npm i styled-components @types/styled-components
```

```
import styled from "styled-components";

const Button = styled.a`
  background: transparent;
  color: white;

  ${props => props.primary && css`
    background: white;
    color: palevioletred;
  `}
`

<Button>Hi!</Button>
<Button primary>Hi!</Button>
```

Docs: https://styled-components.com/docs/basics

Template literal / tagged template: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

HTTP requests in components

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

```
const baseUrl = "https://pcei4.sse.codesandbox.io";

interface User {
  id: number;
  name: string;
  username: string;
  email: string;
}

function HttpRequest() {
  const [users, setUsers] = useState<User[]>([]);

  useEffect(() => {
    Axios.get<User[]>(`${baseUrl}/users`).then((response) => {
      setUsers(response.data);
    });
  }, []);

  return (
    <>
      {users.map((user) => {
        return <div key={user.id}>{user.name}</div>;
      })}
    </>
  );
}
```

Place the HTTP request inside a **useEffect** hook

Use a type parameter to tell Axios what kind of data the request produces

The type of the response is now `User[]`

Remember to define a dependency list for useEffect! Otherwise the component will make a lot of requests and causes an infinite loop.

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

```
class HttpRequest extends React.Component {
  state: { users: User[] } = {
    users: []
  };

  componentDidMount() {
    Axios.get<User[]>(`${baseUrl}/users`).then((response) => {
      const users = response.data;
      this.setState({ users });
    });
  }

  render() {
    return (
      <>
        {this.state.users.map((user) => {
          return <div key={user.id}>{user.name}</div>;
        })}
      </>
    );
  }
}
```

In class components, start the HTTP request in the **componentDidMount** lifecycle method

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Test API in CodeSandbox

Fork this project: https://codesandbox.io/s/test-api-pcei4

Once the server starts, the API address is shown in the terminal.

Terminal 79 Console 0 Problems 0

yarn start

```
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node src/index.js`
API running in https://pcei4.sse.codesandbox.io
GET / 200 581 - 8.368 ms
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Exercise

Loading data with hooks

1. Make sure you have the Test API running
2. Open: **src/components/HttpRequest.tsx**
3. Place the component in the App component
4. Make an HTTP request using a hook & Axios
   a. Which hook do you need?
   b. How do you write the dependency list to make the hook run only when the component was mounted for the first time?
5. Map the user objects to User components

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Decoupling code with custom hooks

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Decoupling

- The purpose of decoupling is to separate reusable code from component-specific parts so that the code can be generalized and reused in other components

- Problems
  - Similar components tend to use similar code
  - "Reusing" similar code by **copy-paste will not take you far and is considered bad practice**

- Solutions
  - **Custom hooks (https://reactjs.org/docs/hooks-custom.html)**
    - Hooks are very straightforward to use for separating app logic from presentation
  - Higher order component (https://reactjs.org/docs/higher-order-components.html, legacy)
    - Creates a wrapper component with special features around user's component
    - Data is passed via props to user's component
  - Render props (https://reactjs.org/docs/render-props.html, legacy)
    - More declarative and dynamic than higher-order components
    - Based on using an inline function that is given certain parameters

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Custom hooks

```
import { useState } from "react";

function useSimple() {
  const [value] = useState(0);
  return value;
}

export default useSimple;
```

A hook's name starts always with the word **use**. Otherwise it's just a normal function from React's point of view.

Other hooks (custom or built-in) can be used in the implementation

A custom hook can return any value

```
import useSimple from "../hooks/useSimple";

function SomeComponent() {
  const value = useSimple();

  return <div>{value}</div>
}
```

Custom hooks are usually located under a specific hooks folder

Custom hooks are called just like built-in hooks

The return value can be whatever the hook returns: a primitive value, an object, an array, a function, ...

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Exercise: Counter hook

1. Create a new file: src/**hooks**/**useCounter**.ts
2. Take the counter state and related functions from the Counter component into a custom hook
   a. **useState** for state
   b. **useCallback** for memoizing the increment and decrement functions
3. In the Counter component, replace the old implementation with the new hook (comment out the old code)

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Exercise: A custom hook for HTTP requests

1. Create a new file: src/hooks/**useHttpRequest**.ts
2. Extract the code from the HttpRequest component into the new hook
   a. Use **type parameters** to allow defining the return type of the HTTP response
   b. Signature:
   ```
   useHttpRequest<ResponseType>(url: string): ResponseType | null
   ```
3. Replace the existing implementation with the new **useHttpRequest** hook in the **HttpRequest** component

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

React Router

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Setup

```
import { BrowserRouter } from 'react-router-dom';

ReactDOM.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>,
  document.getElementById('root')
);
```

- **BrowserRouter** becomes the new root component of the app (index.tsx)
  - It provides route information via React's Context API to Route components
  - Use *MemoryRouter* in Jest tests when initializing the App component
- BrowserRouter uses the History API
  - Provides support for "clean URLs": **/about** without the #
  - If needed, HashRouter can be used instead, e.g. in Cordova apps and to support very old browsers

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Route configuration

Route configurations are placed inside the **Routes** component

Routes are defined using the Route component. The **path** and the **element** prop tell what should be rendered at which path.

```
<Routes>
  <Route path="/" element={<HomeRoute />} />
  <Route path="/users" element={<UsersRoute />} />
  <Route path="/users/:userId" element={<UserRoute someProp={123} />} />
</Routes>
```

Additional props can be defined here

A URL parameter whose value becomes available in the UserRoute component

- Installation: npm install react-router-**dom**
- The shown examples are for React Router version 6. Earlier versions work differently.
- Docs:
  - https://github.com/remix-run/react-router/blob/main/docs/getting-started/tutorial.md
  - https://reactrouter.com/docs/en/v6

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Reading URL parameters

```
<Route path="/users/:userId" element={<UserRoute someProp={123} />} />
```

```
function UserRoute({ someProp = 0 }: { someProp: number }) {
  const { userId } = useParams();

  const [user, setUser] = useState<User | null>(null);

  useEffect(() => {
    loadUser(userId).then((user) => setUser(user));
  }, [userId]);

  if (user === null) {
    return <LoadingIndicator />;
  }

  return <>...</>;
}
```

The **useParams** hook is the only way to read the URL parameters. Class components cannot be used as route elements.

The hook returns an object that contains all matched URL parameters as strings.

Parameter names are defined in the path configuration.

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Links

Link can be used anywhere in the app just like an anchor element (a)

```
<Link to="/">Home</Link>
<Link to="/about">About</Link>
```

NavLink is useful in menus when you want to display e.g. a highlight for the active link

```
<NavLink to="/" end>Home</NavLink>
<NavLink to="/about" end className={({ isActive }) => (isActive ? "active" : "")}>About</NavLink>
```

If the **end** prop is defined, **isActive** is set to true only if the current path matches exactly to the **to** prop's value

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Exercise

Getting started with React Router

1. Open **src/index.tsx**
2. Import **BrowserRouter** and add it to the root of your app
3. After saving the file, make sure the app runs in the browser without errors

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Exercise

Modify the Menu component to use NavLinks

1. Open **src/components/Menu.tsx**
2. Replace the **a** elements with the **NavLink** component
   a. Replace the existing **href**="" attribute with the prop **to**=""
   b. Give the **end** prop to the NavLinks

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Exercise

Using the Route component

1. Define some routes in the **App** component using the **Route** component:
   a. /
   b. /users
   c. /users/:userId
   d. /other
2. Print out each route's name to make sure they work (just add some text <Route to="/" element={<h1>Home</h1>} />)
3. Open: **src/components/UsersRoute.tsx**
   a. In the **Users<u>s</u>Route** display the route's name
4. Open: **src/components/UserRoute.tsx**
   a. In the **UserRoute**, print the **:userId** URL parameter using the **useParams** hook

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Nested routes & <Outlet />

```
<Routes>
  <Route path="/" element={<HomeRoute />} />
  <Route path="users/*" element={<UsersRoute />}>
    <Route path=":userId" element={<UserRoute />} />
  </Route>
</Routes>
```

To display a nested route, use the **Outlet** component in the parent route

This is a nested route.

- Nested routes can be defined by using child elements
- Nesting makes the config a bit easier to read
- Nested routes are placed to the DOM using the Outlet component
- Docs: https://reactrouter.com/docs/en/v6/components/outlet

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Unit testing

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

```
Snapshot Summary
 › 1 snapshot test failed in 1 test suite. Inspect your code changes or press `u` to update them.

Test Suites: 1 failed, 2 passed, 3 total
Tests:       1 failed, 3 passed, 4 total
Snapshots:   1 failed, 3 passed, 4 total
Time:        0.506s, estimated 1s
Ran all test suites.

Watch Usage: Press w to show more.
```

- Jest – a framework for running tests
  - Docs: https://jestjs.io/
  - Tests can be written for e.g. React components, Redux reducers, regular functions etc...
  - Can be used locally and in CI environment
  - Cheatsheet: https://github.com/sapegin/jest-cheat-sheet
- Jest is pre-configured in Create React App projects

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Running tests

- In the project directory, run: **npm test**
  - Starts Jest and runs all tests by default

- Using Jest: key commands
  - **p** – Filter test files by a pattern
  - **t** – Filter test cases by a pattern
  - **q** – Stop tests and exit Jest
  - **Enter** – Runs tests again (applies pattern if such exists)
  - **i** – Interactive snapshot mode
  - **u** – Updates snapshot tests if test run has outdated snapshots

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Structuring test files

```
describe("Navigation", () => {
  it("navigates to the Home route", () => {
    // Test code
  });

  it("navigates to the Users route", () => {
    // Test code
  });
});
```

```
PASS  src/App.test.js
  Navigation
    √ navigates to the Home route (1 ms)
    √ navigates to the Users route (1 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.813 s, estimated 2 s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

Jest's global functions for writing tests

- **describe**(name, callback)
  - For grouping tests
  - Setup / teardown: **beforeAll**(), **beforeEach**(), **afterAll**(), **afterEach**()
- **it**(name, callback) or **test**(name, callback)
  - Defines a test case
  - The test case can use async-await or return a Promise

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Writing a test case

```
function sum(a: number, b: number) {
  return a + b;
}

it("sums two integers", () => {
  const result = sum(1, 2);

  expect(result).toBe(3);
});
```

Function under test

expect() is a global function provided by Jest for inspecting values using *matcher* functions

toBe() is a matcher function. There are a lot of matcher functions, see the cheatsheet.

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Writing tests

- Organizing test cases
  - **describe(name, fn)** – Groups test cases
  - **it(name, fn)** or **test(name, fn)** – Defines a test case, used inside a describe function
  - https://jestjs.io/docs/en/api.html

- expect() and matchers
  - **expect(**value**).toBe(**expectedValue**)** – For inspecting values
  - https://jestjs.io/docs/en/expect.html

- Mocking functions and modules
  - **jest** – A global variable that contains the Jest API
  - **jest.mock** – For mocking module dependencies
  - https://jestjs.io/docs/en/jest-object.html

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Testing React components

- Additional libraries are needed to be able to test React components
- The following are included in CRA projects by default
  - @testing-library/react
    - Functions for rendering components in a test environment
  - @testing-library/jest-dom
    - A set of matcher functions for testing the DOM tree
  - @testing-library/user-event
    - Functions for triggering events in the DOM tree

- Enzyme used to be a popular tool but it hasn't been updated in ages (supports React 16, current version is 18)

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Testing Library: Basic usage

Renders a component to the 'screen'

Find a link that contains the text "Home"

Clicks the link

Finds a title

Make sure that the title exists

```
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";

it("navigates to the Home route", () => {
  render(
    <MemoryRouter>
      <App />
    </MemoryRouter>
  );

  const homeLink = screen.getByText("Home", { selector: "nav > a" });
  userEvent.click(homeLink);

  const header = screen.getByText("Home", { selector: "h1" });
  expect(header).toBeInTheDocument();
});
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Exercise: Tests for the Counter component

1. Open src/components/**Counter.test.tsx** and **Counter.tsx**
2. In the Counter component, add a **data-testid** attribute to the HTML element that contains the value

   Counter's value: data-testid="**Counter-value**"

3. Fill in the test cases
   a. Get the button elements using **screen.getByText()** and the value field by **screen.getByTestId()**
   b. Make sure that the elements exist: **expect(element).toBeInTheDocument()**
   c. You can click an element with **userEvent.click()**
   d. Textual value can be read with expect().**toHaveTextContent()**
   e. **Update props by triggering a re-render and test that the value changes correctly**:

```
const { rerender } = render(<Counter initialState={2} />);
rerender(<Counter initialState={4} />);
```


INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

Testing Library

- The idea is to test components the way the user would use them
  - Focus on input and output instead of testing component's internals
  - Make use of HTML attributes (role, label, name, ..) when querying for elements

- Hooks cannot be tested alone without components
  - Figure out a minimal "testbed" component for testing a hook if the real components are too hard to test with Jest

- Docs
  - Rendering: https://testing-library.com/docs/react-testing-library/api
  - Finding elements: https://testing-library.com/docs/queries/about/
  - Matchers: https://github.com/testing-library/jest-dom#custom-matchers
  - Events: https://github.com/testing-library/user-event#api

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi

About testing with Jest

- Jest can be used for testing almost any JavaScript code
  - React components are tested in a **browser-like** environment (jest-dom)
  - **Unit tests** ensure that the unit under test (a component, a function, etc.) works properly with different input isolated from the rest of the app
  - **Integration tests** ensure that components work together

- What cannot be tested with Jest?
  - How components look like (visual testing, screenshot comparison)
  - Components behavior on different devices (manual testing, BrowserStack.com)
  - ***--> End to end testing*** (Cypress.io, Playwright)

INSTRUCTOR Jukka Tupamäki
jukka@lemanse.fi | www.lemanse.fi