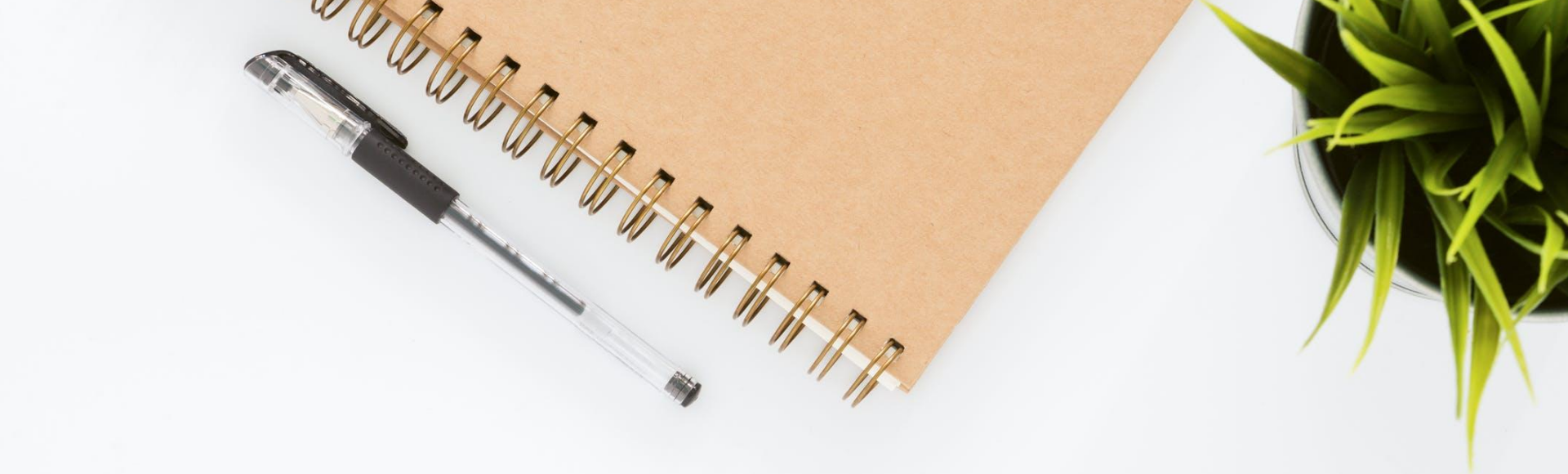

React design patterns

adapt

Function component

Function components are the simplest way to declare reusable components. They're just functions, also works with hooks :)

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

```
const Greeting = () => (<div>Hi there!</div>);
```

adapt

Function component

Function components are the simplest way to declare reusable components. They're just functions, also works with hooks :)

Collect **props** from the first argument of your function.

```
function Greeting(props) {
  return <div>Hi there! {props.name}</div>;
}
```

```
const Greeting = (props) => <div>Hi there {props.name}</div>
```

```
const Greeting = props => <div>Hi there {props.name}</div>
```

adapt

Function component

Function declaration vs constant declaration, what's the difference?

```
function myFunction() {
    console.log("hi");
}
```

VS

```
const myFunction = () => console.log("hi");
```

adapt

Function component

Function declaration vs constant declaration, the difference

```
myFunction();
function myFunction() {
    console.log("hi");
}

// hi
```

```
myFunction();
const myFunction = () => console.log("hi");

// Uncaught ReferenceError:
// Cannot access 'myFunction' before initialization
```

adapt

Function component

Function components are the simplest way to declare reusable components. They're just functions, also works with hooks :)

Set defaults for any required **props** using **defaultProps**.

```
function Greeting(props) {
  return <div>Hi {props.name}!</div>;
}
Greeting.defaultProps = {
  name: "Guest"
};
```

adapt

Destructuring props

Destructuring assignment is a JavaScript feature.
It was added to the language in ES2015.
So it might not look familiar.

Think of it like the opposite of literal assignment.

```
let person = { name: "Ovidijus" };
let { name } = person;
```

```
let things = ["one", "two"];
let [first, second] = things;
```

adapt

Destructuring props

Destructuring assignment is used a lot in function components.
These component declarations below are equivalent.

```
function Greeting(props) {
  return <div>Hi {props.name}!</div>;
}

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

adapt

Destructuring props

There's a syntax for collecting remaining props into an object.
It's called rest parameter syntax and looks like this.

```
function Greeting({ name, ...restProps }) {
  return <div>Hi {name}!</div>;
}
```

adapt

Conditional rendering

You can't use if/else statements inside a component declarations.
So conditional (ternary) operator and short-circuit evaluation are your friends.

```
// IF
{
  condition && <span>Rendered when `truthy`</span>;
}
```

adapt

Conditional rendering

You can't use if/else statements inside a component declarations.
So conditional (ternary) operator and short-circuit evaluation are your friends.

```
// UNLESS
{
  condition || <span>Rendered when `falsy`</span>;
}
```

adapt

Conditional rendering

You can't use if/else statements inside a component declarations.
So conditional (ternary) operator and short-circuit evaluation are your friends.

```
// IF-ELSE
{
  condition ? (
    <span>Rendered when `truthy`</span>
  ) : (
    <span>Rendered when `falsy`</span>
  );
}
```

adapt

Children types

React can render **children** from most types.
In most cases it's either an **array** or a **string**.

```
// String
<div>Hello <span>World!</span></div>
// Array
<div>{["Hello ", <span>World</span>, "!"]}</div>
```

adapt

State hoisting

Events are changes in state. Their data needs to be passed to stateful container components parents.

This is called "state hoisting". It's accomplished by passing a callback from a container component to a child component.

```
class NameContainer extends React.Component {
  render() {
    return <Name onChange={newName => alert(newName)} />;
  }
}

const Name = ({ onChange }) => (
  <input onChange={e => onChange(e.target.value)} />
);
```

adapt

React Router

React Router is a collection of navigational components that compose declaratively with your application.

adapt

React Router: Routers

```
import { BrowserRouter } from "react-router-dom";
ReactDOM.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>,
  holder
);
```

adapt

React Router: Matching

There are two route matching components: <Route> and <Switch>.

Route matching is done by comparing a <Route>'s path prop to the current location’s pathname. When a <Route> matches it will render its content and when it does not match, it will render null. A <Route> with no path will always match.

You can include a <Route> anywhere that you want to render content based on the location. It will often make sense to list a number of possible <Route>s next to each other. The <Switch> component is used to group <Route>s together.

adapt

React Router: Routers

```
import { Route, Switch } from "react-router-dom";

// when location = { pathname: '/about' }
<Route path='/about' component={About}/> // renders <About/>
<Route path='/contact' component={Contact}/> // renders null
<Route component={Always}/> // renders <Always/>

<Switch>
  <Route exact path="/" component={Home} />
  <Route path="/about" component={About} />
  <Route path="/contact" component={Contact} />
</Switch>

<Switch>
  <Route exact path="/" component={Home} />
  <Route path="/about" component={About} />
  <Route path="/contact" component={Contact} />
  {/* when none of the above match, <NoMatch> will be rendered */}
  <Route component={NoMatch} />
</Switch>
```

adapt

React Router: Rendering Props

You have three prop choices for how you render a component for a given <Route>: **component**, **render**, and **children**.

Component should be used when you have an existing component (either a React.Component or a stateless functional component) that you want to render. **Render**, which takes an inline function, should only be used when you have to pass in-scope variables to the component you want to render. You should not use the component prop with an inline function to pass in-scope variables because you will get undesired component unmounts/remounts.

adapt

React Router: Rendering Props

```
const Home = () => <div>Home</div>;

const App = () => {
  const someVariable = true;

  return (
    <Switch>
      {/* these are good */}
      <Route exact path="/" component={Home} />
      <Route
        path="/about"
        render={props => <About {...props} extra={someVariable} />}
      />
      {/* do not do this */}
      <Route
        path="/contact"
        component={props => <Contact {...props} extra={someVariable} />}
      />
    </Switch>
  );
};
```

adapt

React Router: Navigation

React Router provides a <Link> component to create links in your application. Wherever you render a <Link>, an anchor (<a>) will be rendered in your application's HTML.

The <NavLink> is a special type of <Link> that can style itself as "active" when its to prop matches the current location.

Any time that you want to force navigation, you can render a <Redirect>. When a <Redirect> renders, it will navigate using its to prop.

adapt

React Router: Navigation

```
<Link to="/">Home</Link>
// <a href='/'>Home</a>


// location = { pathname: '/react' }
<NavLink to="/react" activeClassName="hurray">
  React
</NavLink>
// <a href='/react' className='hurray'>React</a>


<Redirect to="/login" />
```

adapt

Exercises @ lesson9/readme.md

adapt