React Basics

Learning Objectives

- [ ] Understanding what React is and why it is used
- [ ] Understanding JSX and differences to HTML
- [ ] Understanding the declarative approach of React
- [ ] Creating React components
- [ ] Understanding rendering with React
- [ ] Knowing about the React ecosystem

What is React and why do we use it?

React is a JavaScript library with the purpose of making the developer's life easier: you don't need to work directly with the DOM API (e.g. `createElement`) in most cases. You just write simpler (declarative) code describing what the user interface should look like and React handles the DOM under the hood.

To write declarative code for React, you use JSX.

Using JSX

JSX is a syntax extension to JavaScript. JSX is neither a string, nor HTML as we know it. JSX expressions can be used anywhere a JavaScript expression can be used.

```
const element = <p>Some Text</p>;
```

We use JSX to create React elements. React elements are an intermediary format that React converts to DOM elements during the rendering process. This allows us to declaratively describe our user interface using JSX.

Creating Elements

Just like in HTML, JSX elements are described using opening and closing tags. The opening tag contains the tag name or the component type (see Using Components) and any attributes. The closing tag contains the same tag name or the same component type as the opening tag does and nothing else. The children of the element are placed between the opening and closing tag. If the element has no children, the closing tag can be omitted and the element is self-closing.

```
// Element with children
//
//              opening tag         children
//              |  attributes       |        closing tag
//              |  |                |        |
const element = <p className="text">Some Text</p>;
//               | |         |                 |
//               | |         attribute value   |
//                | attribute name            |
//                tag name or component type ---+

// Self-closing element
//
//             self closing tag   slash denotes self closing
//             |      attributes  |
//             |      |           |
const input = <input type="text" />;
//              |     |    |
//              |     |    attribute value
//              |     attribute name
//              tag name or component type
```

> 💡 Elements that do not support closing tags in HTML like `<br>` or `<input>` must be self-closing in JSX (like `<br />` or `<input type="text" />`).

> 💡 Unlike HTML, which is resilient to missing closing tags, JSX is not. If you forget to close a tag, you will get an error.

Using Components

To create an element from a component, we can simply refer to it by the function name in JSX and treat it just like any built-in component:

```
const element = <MyComponent />;
```

Regarding attributes and children, creating elements from component types works just like with any (HTML) tag name.

> 💡 JSX makes the distinction between built-in (HTML) tag names and components by looking at the first character inside the JSX tag. If it is lowercase it's treated as a built-in tag name, if it is uppercase it looks for any defined JavaScript function with that name. That is why it is important to use PascalCase for component names.

> 📙 Read more about **Writing Markup with JSX** in the React Docs.

Attributes

Attributes for built-in HTML elements use JavaScript-centric names from the DOM API. In most cases the names are the same as in HTML, but there are some exceptions. For example, the `class` attribute from HTML is called `className` in JSX.

Passing string values to attributes is done by using double quotes. To pass any JavaScript expression use curly braces.

```
const element = <p className="text">Some Text</p>;
const myValue = "This is a string";
const input = <input type="text" value={myValue} minLength={5} />;
```

Nesting Elements

React elements can be nested the same way we have been nesting our HTML elements.

```
const element = (
  <div>
    <p>Some Text</p>
    <p>Some more Text</p>
  </div>
);
```

> 💡 Multiline JSX expressions are wrapped in parentheses to make them easier to read. No worries: Prettier will take care of that for you.

Interpolating Expressions

We can use any JavaScript expression inside JSX by wrapping it in curly braces. This is called interpolation. It is similar to string interpolation in JavaScript template strings.

```
const name = "Pawtricia";
const element = <p>My cat's name is {name}</p>;
```

```
const a = 5;
const b = 10;

const element = (
  <p>
    {a} + {b} = {a + b}
  </p>
);
```

> 💡 You can only use expressions inside JSX. Statements like `if` or `for` are not allowed.

> 💡 To learn how to interpolate JavaScript expressions inside JSX attributes, refer to the Attributes section.

> 📙 Read more about **JavaScript in JSX with Curly Braces** in the React Docs.

React Components

React applications are built using components. A component is an independent and reusable piece of the user interface that contains its own structure, logic, and potentially styling.

React components are JavaScript functions that return React elements. Those elements are then turned into DOM elements by React during the rendering process.

In order to create a React component, we write a named function (using PascalCase) and have it return the desired elements using JSX.

```
function MyButton() {
  return (
    <button type="button" className="default-button">
      I'm a button
    </button>
  );
}
```

This is a very powerful concept, because it allows us to reuse the same component in multiple places in our application.

> 💡 See Using Components for more information on how to use components in JSX.

> 💡 There are hardly any limitations to how 'small' a component can be (i.e. a button), or how 'big' (i.e. an entire page).

> 📙 Read about **Your First Component** in the React Docs.
>
> **Note**: *Exporting the component* and *Nesting and organizing components* are out of scope for this first session.

Imperative vs. Declarative Programming

The main difference between imperative and declarative code is that imperative code describes *how* something should be built, and declarative code describes *what* needs to be built.

> 💡 Imagine building a stool. Imperative "code" would describe the steps you need to take to build the stool. Declarative "code" would describe the stool itself.
>
> Imperative:
>
> - take 4 wooden slats
> - take 1 wooden board
> - take 4 screws
> - take a screwdriver
> - screw the slats under the board perpendicularly
> - position your work so that the board is on top
>
> Declarative:
>
> - a stool with 4 legs and a seat, standing upright

In imperative programming, your code performs a series of actions.

In declarative programming, your code describes a desired outcome.

The way we have used JavaScript during this course so far has been mostly imperative. We have described what needs to be done to get a certain result.

```
const p = document.createElement("p");
p.classList.add("introText");
p.textContent = "Hello World!";
rootElement.append(p);
```

Now, React allows us to use JavaScript in a declarative way. We describe to React what we want, and React figures out how to update the DOM according to our description.

```
root.render(<p className="introText">Hello World!</p>);
// React could interpret this to do the following:
// const p = document.createElement("p");
// p.classList.add("introText");
// p.textContent = "Hello World!";
// rootElement.append(p);
// …
```

How React Renders

React needs to know where to render the elements it creates. We select the DOM element we want to render into by using `document.querySelector()`. We then create a React root object. The root object has a `render()` method that we can use to render React elements into the DOM.

HTML

```
<div id="root"></div>
```

JavaScript

```
const rootElement = document.querySelector("#root");
const root = ReactDOM.createRoot(rootElement);
root.render(<h1>Hello, world</h1>);
```

You'll probably never have to write this code yourself, because it is already included in all templates and starters. In the real world it usally looks like this:

```
const rootElement = document.getElementById("root");
const root = ReactDOM.createRoot();

root.render(
  <React.StrictMode>
```

```
    <App />
  </React.StrictMode>
);
```

Here we have an imported `<App />` element that is wrapped in `<React.StrictMode>`.

> 💡 `StrictMode` sets up React to run in strict mode. In strict mode React points out potential problems in an application.

React works smart, not hard

React only updates the DOM elements that have changed compared to the last render. This is very efficient and provides a great user experience (focus stays consistent, inputs keep their values, etc.) as well as a great developer experience (declaritive code is much easier to reason about).

Nice to know: React, JSX, Transpilers and Bundlers

Since JSX is not a standard JavaScript syntax, we need to use a transpiler (a tool that translates one variant of a language into another) to transform it into standard JavaScript, that can be understood by the browser.

A bundler is a tool that combines all the files of our codebase into one file, that we can include in our HTML. The bundler also takes care of running the transpiler when needed.

The bundler creates a development server when we run `npm run start` locally. CodeSandbox does this for us automatically.

> 💡 You might notice that in the challenges we are using an `import` statement to import `.css` files into our JavaScript files. This is not a standard JavaScript feature, but it is supported by the bundler. A css import statement is transformed into a `<link>` element in the HTML automatically.
>
> ```
> import "./styles.css";
> ```

Resources

- What is React: A Visual Introduction For Beginners on learnreact.design
- Writing Markup with JSX in the React Docs
- JavaScript in JSX with Curly Braces in the React Docs
- Your First Component in the React Docs
- Difference between a Framework and a Library on freecodecamp