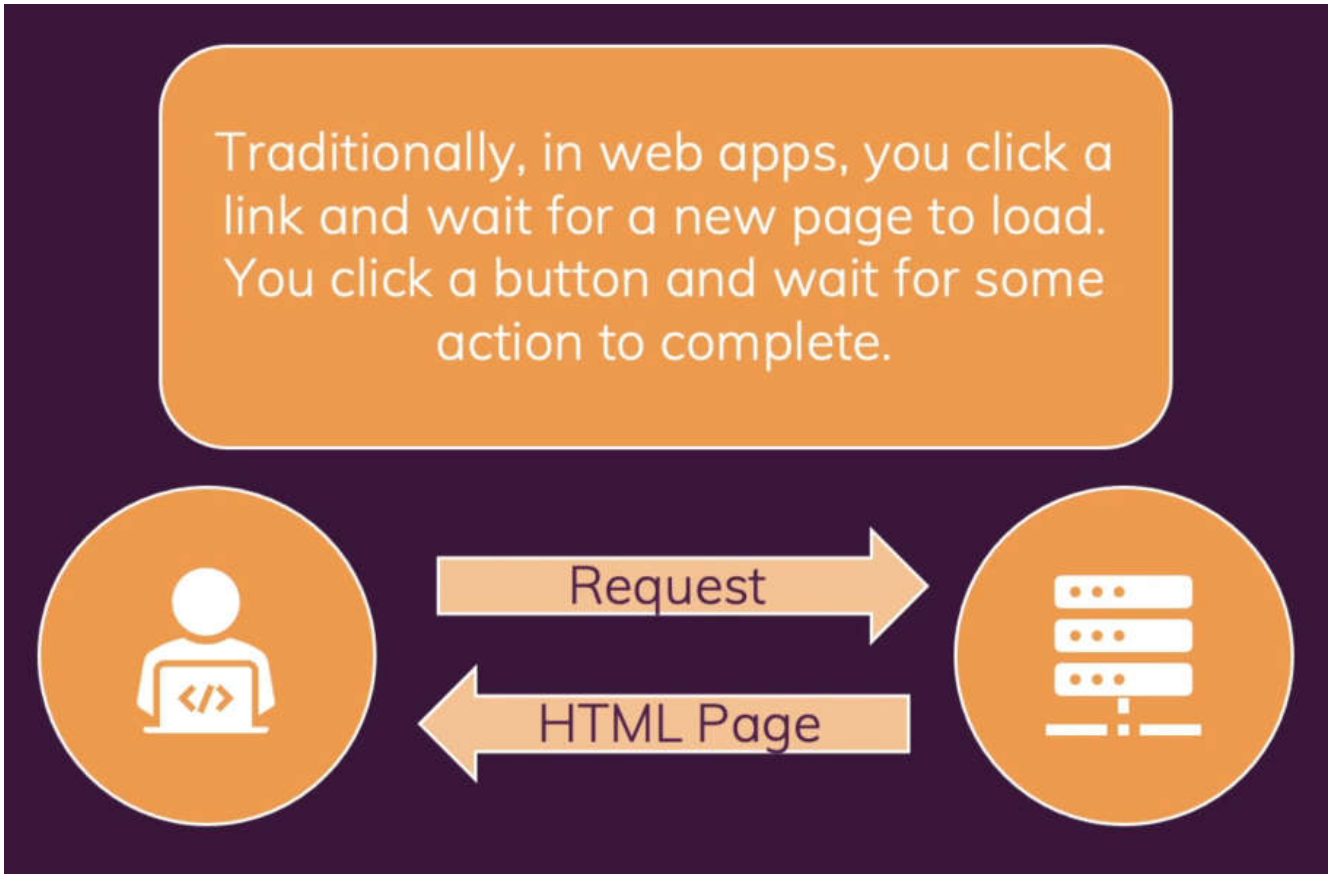
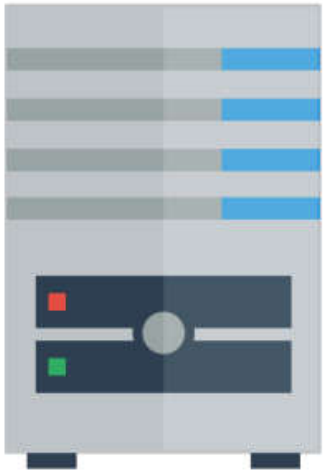
Lecture 1

Getting Started with JavaScript ES6 and React Fundamentals

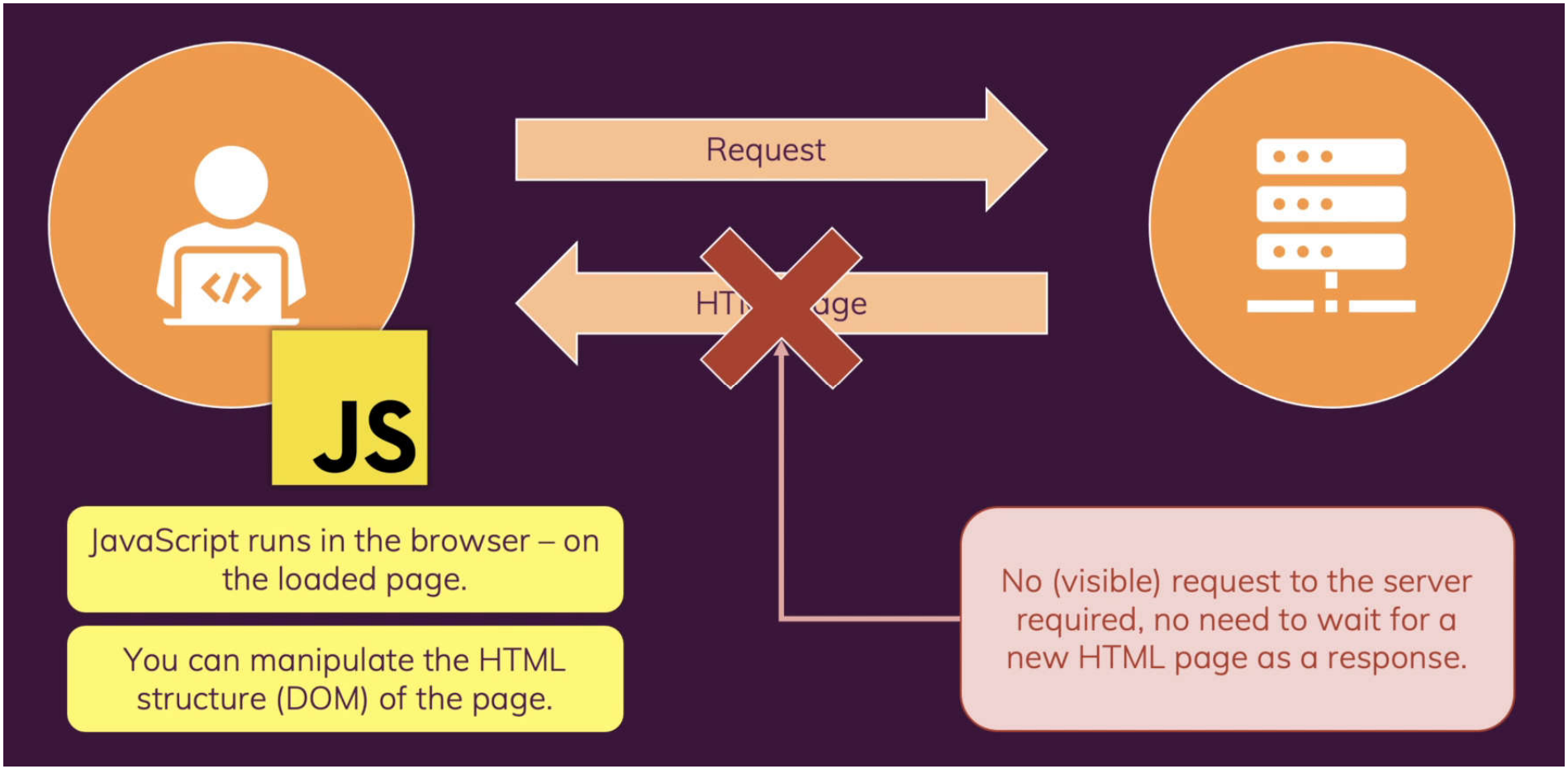
Traditional web application



Web server

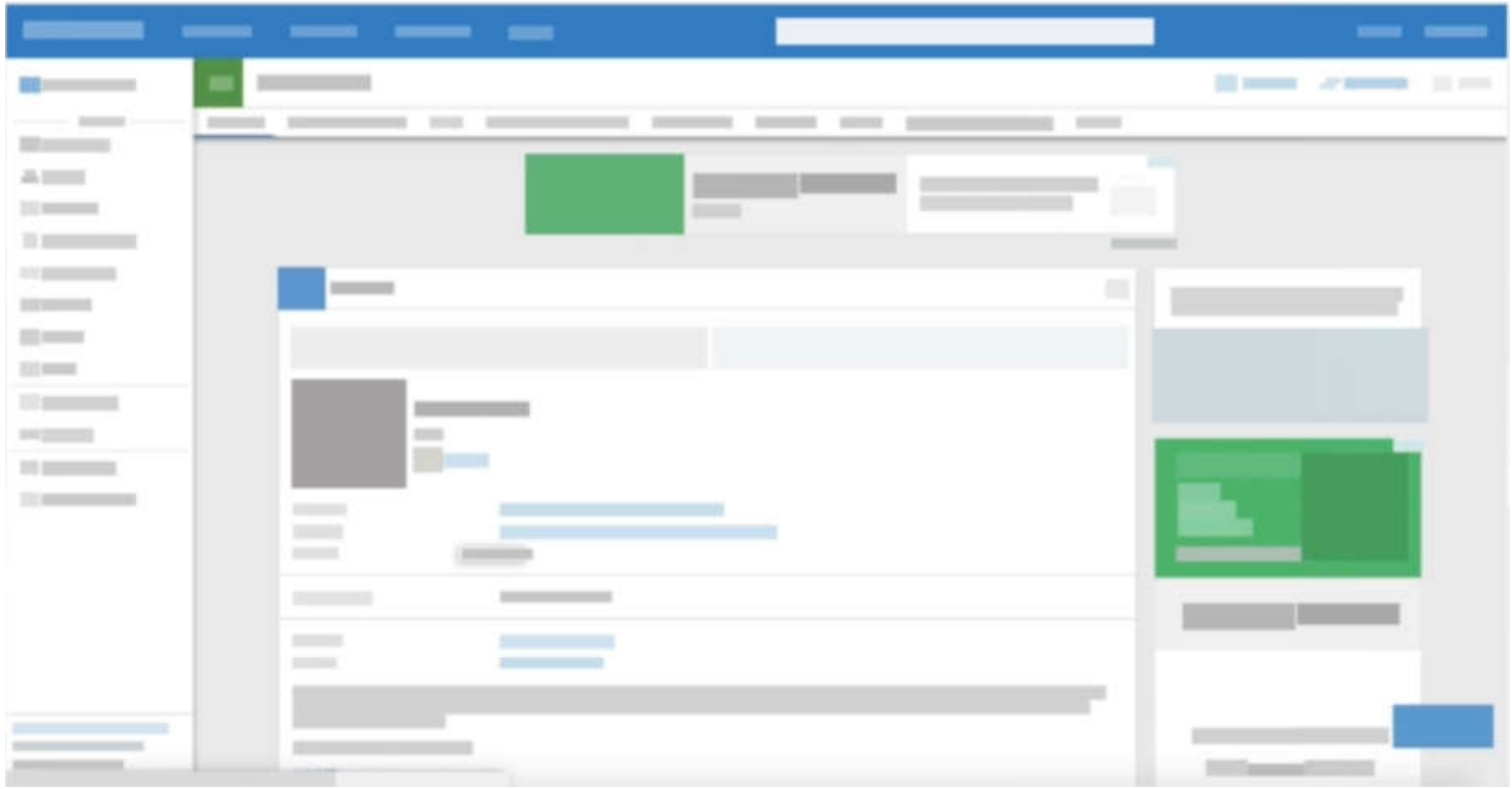


JavaScript can make websites feel more ‘reactive’



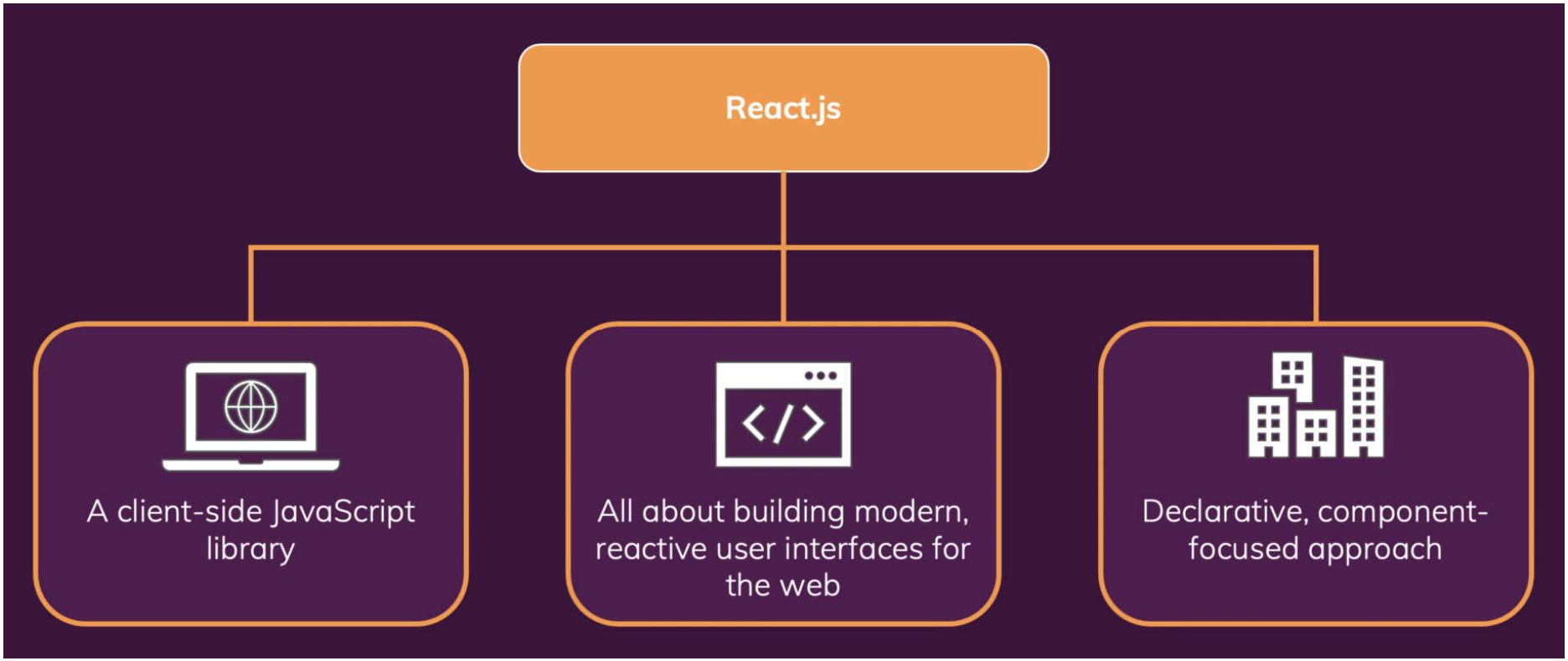
The problem of building complex user interfaces

- Things get more & more complicated
- Harder & harder to debug the code & understand how parts affect the others

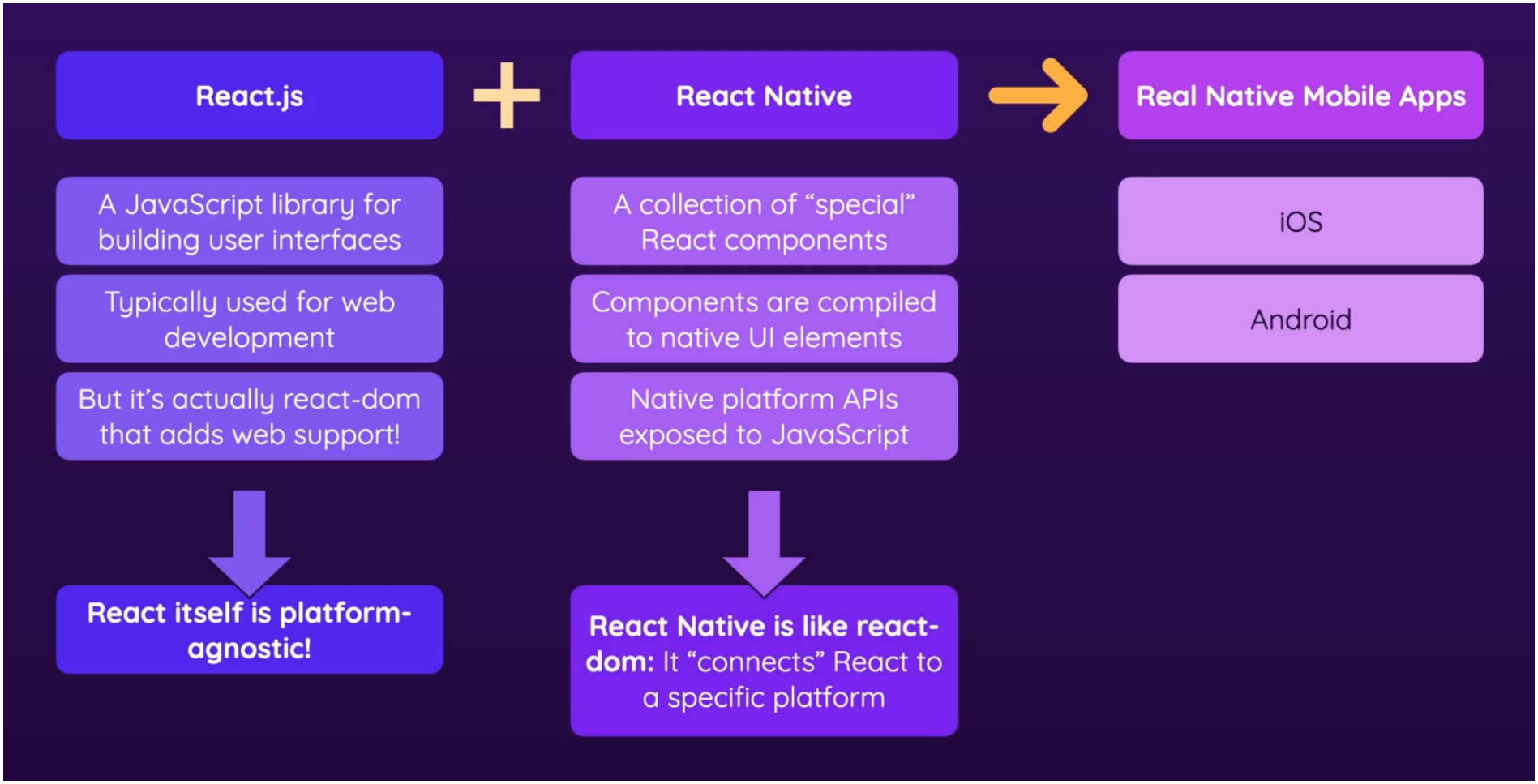


What is React?

- A JavaScript library for building user interfaces
  - Created by Facebook



What is React Native?



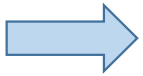
Modern JavaScript features

- let & const instead of var
- Arrow functions
- Modules pattern (import, export)
- Classes, properties, methods
- Some new syntax

Arrow functions

```
function myFunc() {

}
```



```
const myFunc = () => {

}
```

```
const printMyName = (name) => {
   console.log(name);
}
printMyName('John');
```

Exports & Imports (Modules)

person.js

```
const person = {
   name: 'Mary',
   age: 12
}
export default person;
```

common.js

```
export const clean = () => {
   console.log('Cleaning...');
}
export const dbName = 'myDB';
```

app.js

```
import person from './person.js';
import prs from './person.js';

import { dbName } from './common.js';
import { clean, dbName } from './common.js';

import * as common from './common.js';
console.log(common.dbName);
```

default export
you choose the name

named export
name defined by export

JS Classes

```
class Person {
    name = 'Dave';
    greet = () => {
        console.log('Hi there!')
    };
}
```

```
let p1 = new Person();
p1.greet();
```

```
// inheritance
class Person extends Master {
    constructor() {
        super();
    }
}
```

```
class Person {
    name = 'default';
    age = 0;
    constructor(n, a) {
        this.name = n;
        this.age = a;
    }
}
```

Methods in ES6 Classes

- You can add methods in a class.
- Example, to create a method named `present`:

```
class Car {
    constructor(name) {
        this.brand = name;
    }

    present() {
        return 'I have a ' + this.brand;
    }
}

const mycar = new Car("Ford");
mycar.present();
```

JS Class Properties & Methods

Properties are like "variables attached to classes/ objects"

ES6

```
constructor () {
        this.myProperty = ' value'
}
```

ES7

```
myProperty = ' value'
```

Methods are like "functions attached to classes/ objects"

ES6

```
myMethod () { ... }
```

ES7

```
myMethod = () => { ... }
```

Spread & Rest Operator

...

Spread

Used to split up array elements OR object properties

```
const newArray = [...oldArray, 1, 2]
const newObject = { ...oldObject, newProp: 5 }
```

Rest

Used to merge a list of function arguments into an array

```
function sortArgs(...args) {
    return args.sort()
}
```

ES6 Spread Operator for Arrays

- The spread operator (`...`) allows us to quickly copy all or part of an existing array or object into another array or object.

```
const numbersOne = [1, 2, 3];
const numbersTwo = [4, 5, 6];
const numbersCombined = [...numbersOne, ...numbersTwo];
console.log(numbersCombined);
```

- Result: `[ 1, 2, 3, 4, 5, 6 ]`

- Assign the 2 items from array `numbers` to variables and put the rest in an array:

```
const numbers = [1, 2, 3, 4, 5, 6];
const [one, two, ...rest] = numbers;
console.log(rest);
```

- Result: `[ 3, 4, 5, 6 ]`

ES6 Spread Operator or Objects

- We can use the spread operator with objects, too.
- Example of combining two objects:

```
const obj1 = {
    brand: 'Ford',
    model: 'Mustang',
    color: 'red'
}
const obj2 = {
    year: 2021, color: 'yellow'
}
const obj3 = { ...obj1, ...obj2 }
console.log(obj3);
```

- Result: `{ brand: 'Ford', model: 'Mustang', color: 'yellow', year: 2021 }`
- **Note:** the properties that match (i.e. `color`) are overwritten by the later object.

Destructuring

Extract array elements or object properties and store them in variables.

Array Destructuring

```
[a, b] = ['Hello', 'Max']
console.log(a) // Hello
console.log(b) // Max
```

Object Destructuring

```
{name} = {name: 'Max', age: 28}
console.log(name) // Max
console.log(age) // undefined
```

JS Reference Types

Beware that there are reference types in JavaScript.

e.g. objects, arrays

```
const person = {
   name: 'Alex'
}
const person2 = person;
person.name = 'Alesis';
// Alex or Alesis?
console.log(person2.name);
```

JS array method: map()

```
const numbers = [1, 2, 3];
const doubleNumbers = numbers.map(n => n * 2);
console.log(doubleNumbers);
```

This is very similar to Java's functional programming style.

ES6 `Array.map()`

- The `map()` method transforms an array's items with a function and returns the transformed array.

```
const fruits = ['apple', 'banana', 'orange'];

const fruitList = fruits.map(
    fruit => "<li>" + fruit + "</li>"
);

console.log(fruitList);
```

- Result:

```
['<li>apple</li>', '<li>banana</li>', '<li>orange</li>']
```

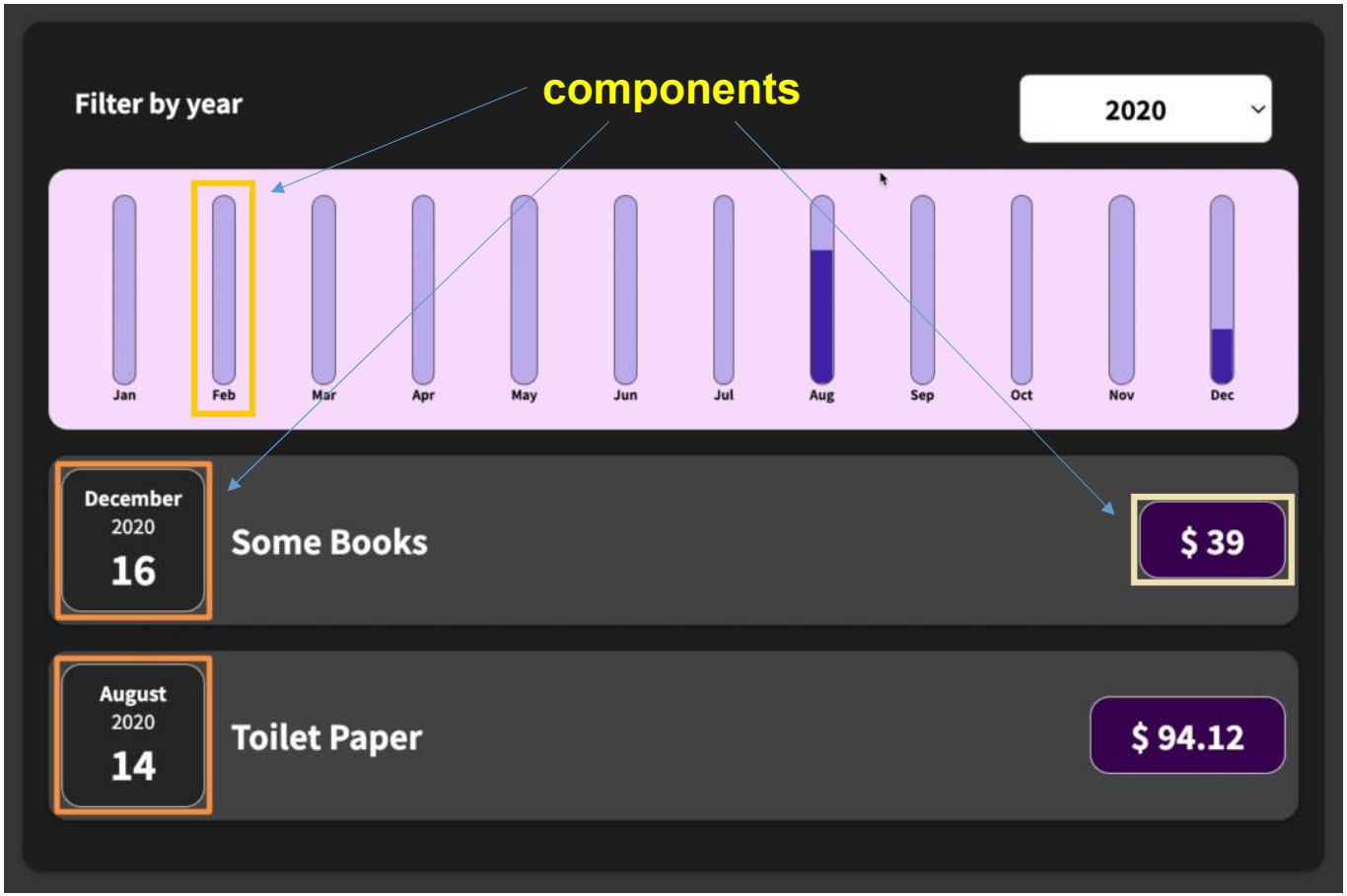
Exercise: array method map()

```
function transformToObjects(numberArray) {
   // Todo: Add your logic
   // should return an array of objects
}
```

For the provided input [1, 2, 3] the transformToObjects() function should return [{val: 1}, {val: 2}, {val: 3}].

ReactJS is all about Components

- Re-usable building blocks in the user interface
  - Composed of HTML, CSS (for styling) and JavaScript for logic



React's Key Concepts

1. Don’t touch the DOM. I’ll do it!

- Libraries & frameworks before:
  - Listen to user events
  - Directly change individual parts of the web page
- Problems:
  - Hard to see relationship between events & DOM changes
  - DOM manipulation takes long time ==> slow
- React solution:
  - User events affect the app's State. State controls what the page looks like
  - Manipulating a Virtual DOM before finally rendering the actual DOM => faster

React's Key Concepts

2. Build website like LEGO blocks

- Reusable components
  - e.g. Button, List, Product, Footer...
- Small components put together ==> bigger component!
- Can move components to other projects

React's Key Concepts

3. Unidirectional data flow

- Data only flow from the top-level component to child components
  - Data never move up
  - All the changes can only flow down from parent component to child components
- Anytime we want to change the webpage, we change the state

React's Key Concepts

4. React builds UI only (the rest is up to you)

- Unlike AngularJS, which is a MVC framework, React is just a UI **library**
  - React only provides the "view" part of the web application (front-end).
  - So we need some kind of back-end (can be Node.js back-end, can be others)
- React everywhere principle:
  - React project can build cross-platform UI
  - Web, Mobile, Desktop, VR...
  - `react-dom`: specific library to build for the web platform

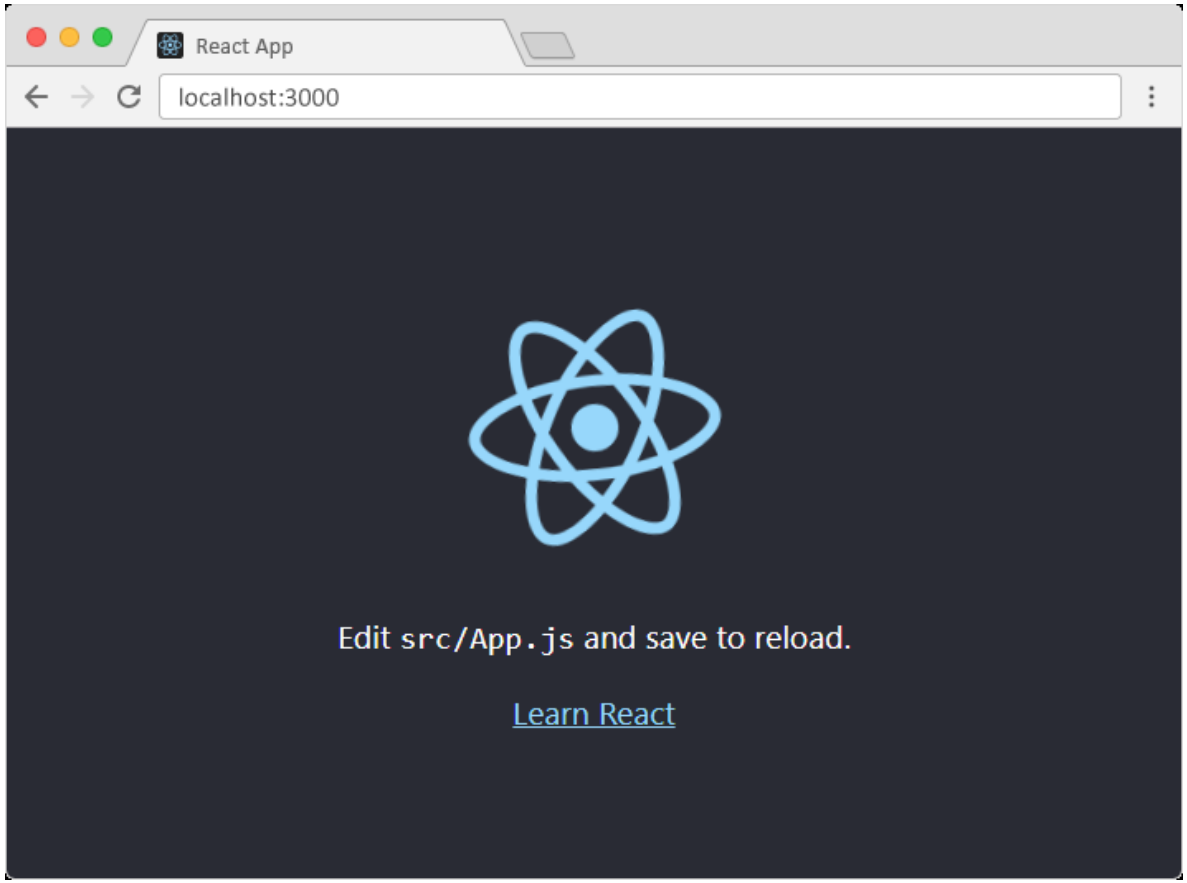
Why use Components?

- Reusability
  - Dont’ repeat yourself
- Separation of concerns
  - Small pieces of code which focus on single purposes

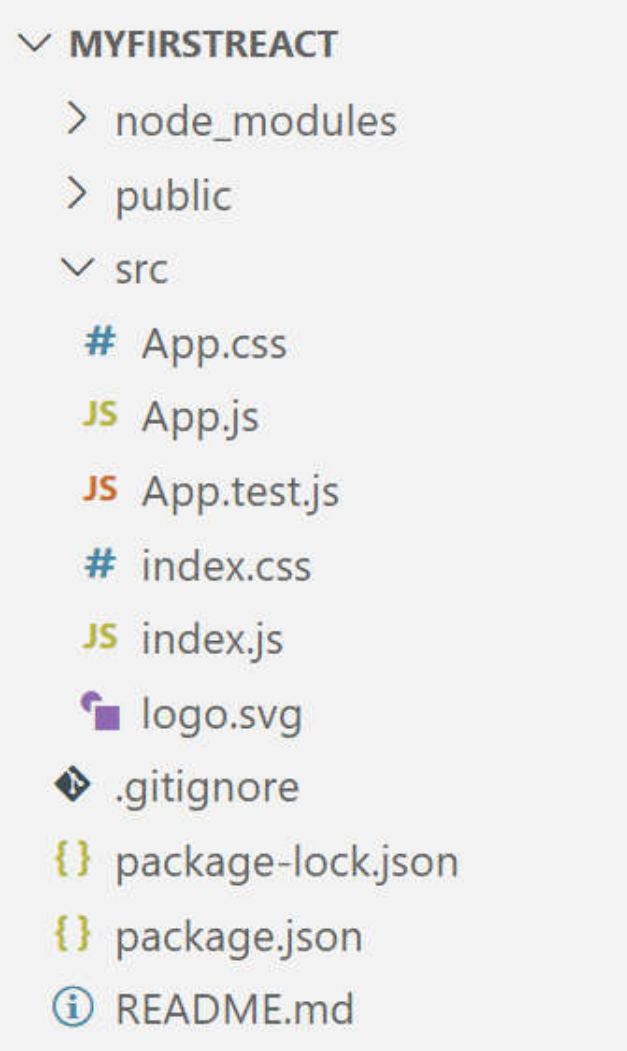
Create a new React project

- Install Node.js (LTS version recommended)
- Install create-react-app
  - npm install -g create-react-app
- Create a project
  - npx create-react-app my-react-app
- Start the application
  - cd my-react-app
  - npm start

React Development Server



React project structure



- A React project is essentially a Node.js project
  - But we're not going to run the `.js` files using `node` command
- `public` folder contains static assets which can be accessed from `http://localhost:PORT/`
  - React actually uses `express` as a web server and serve static files in this `public` folder
- `src` folder contains React source files
- After creating the project, go to the project folder:

```
cd myfirstreact
```

- Start the application:

```
npm start
```

- A browser tap will be opened automatically.

index.js (application entry point)

```
import ReactDOM from 'react-dom/client';

import './index.css';
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

(1) **index.js** is embedded in the **index.html** file under **public** folder.
(more on this in a later slide)

(2) <App /> is a component.

index.css

```
* {
  box-sizing: border-box;
}

html {
  font-family: sans-serif;
}

body {
  margin: 0;
  background-color: #3f3f3f;
}
```

Just some regular CSS which will be injected into the index page.

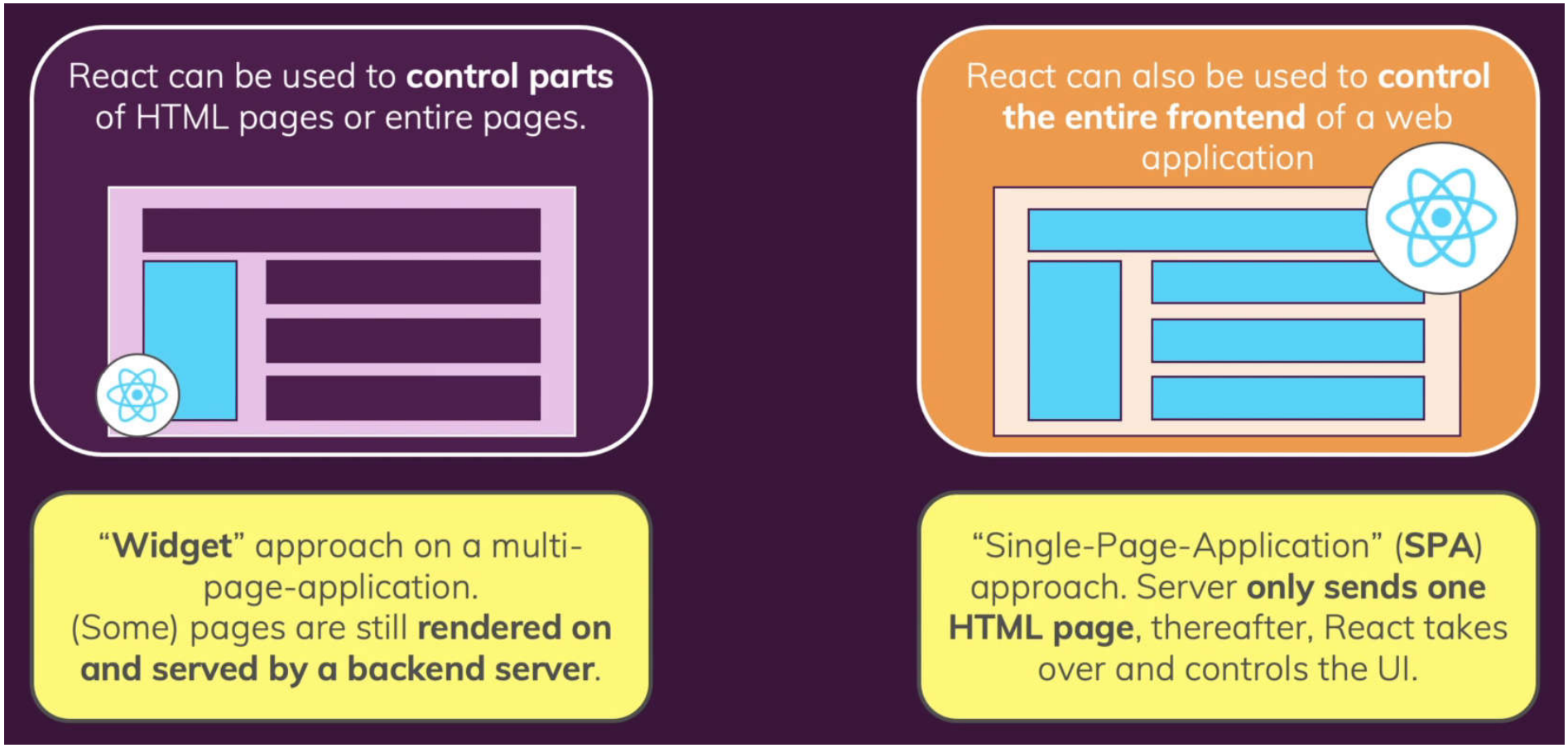
App.js

```
function App() {
  return (
    <div>
      <h2>Let's get started!</h2>
    </div>
  );
}

export default App;
```

The HTML code inside JS code is called JSX, which will be compiled by React into actual JavaScript.

Single-Page-Applications (SPAs)



JSX

What is JSX?

- Stands for **JavaScript XML**
- Used to write HTML in JavaScript
- Easier to write & add HTML in React
- Shortcut for `React.createElement()`
  - Recall: `document.createElement()`

```
const div = document.createElement('div');
outer.classList.add('App');
const h1 = document.createElement('h1');
h1.textContent = 'Hello World!';

div.appendChild(h1);
```

```
function App() {
  return (
    <div className="App">
      <h1>Hello World!</h1>
    </div>
  );
}

export default App;
```

What is JSX?

- JSX allow to write HTML elements in JavaScript and place them in the DOM
  - without any `createElement()` and/or `appendChild()` methods
- JSX converts HTML tags into react elements.

```
const div = document.createElement('div');
outer.classList.add('App');
const h1 = document.createElement('h1');
h1.textContent = 'Hello World!';

div.appendChild(h1);
```

```
function App() {
  return (
    <div className="App">
      <h1>Hello World!</h1>
    </div>
  );
}

export default App;
```

JSX - Expression support

- Embed JS expressions in curly braces `{ }`

```
const intro = (
    <div>
        I'm {thisYear - user.bYear} years old.
    </div>
);
```

JSX - Attribute values

- Do not add quotes for expressions used as attribute value.

```
// right
const sqr = <Square value={i} />;

// wrong
const sqr = <Square value="{i}"
/>;
```

JSX - Single-line and multi-line elements

- Parentheses are NOT needed in multi-line elements
  - But recommended

```
// single-line
const myElement = <h1>I Love JSX!</h1>;

// multi-line
const listElement = (
    <ul>
        <li>PR1</li>
        <li>WPR</li>
        <li>MPR</li>
    </ul>
);
```

JSX's rule - Only <u>one</u> top-level element

```
// the problem
const myElement = (
    <h1>I Love JSX!</h1>
    <h1>Me too!</h1>
);
```

- Solutions:
  1. Wrap them in a parent `<div>` element
  2. Wrap them in a *fragment*. Like so:

```
// use fragment
const myElement = (
    <>
        <h1>I Love JSX!</h1>
        <h1>Me too!</h1>
    </>
);
```

JSX Elements Must be Closed

- JSX follows XML rules
  - HTML elements MUST be properly closed
- Close empty elements with `/>`

```
const element = <input type="text" />;
```

JSX - The `className` attribute

- Use `className` attribute instead of the `class` attribute in HTML.
- This is an exeption because `class` is a JS reserved keyword.

```
const myElement = <button className="square"></button>;
```

JSX - `if` statement support

- Cannot put `if` statement inside `{ }`
  - An `if` statement is not an *expression* anyway.
- → Workaround: use *ternary expression*

```
<h1>Good {h < 12 ? "morning" : "afternoon"}!</h1>
```

JSX - loop/collection support

- Cannot put `for` loops inside `{ }`
- → Workaround: prepare a collection of components in advance

```
const cars = [
    <Car brand="Toyota" />,
    <Car brand="Vinfast" />,
    <Car brand="Mercedes" />
];

return <ul>{cars}</ul>;
```

```
const cars = [
    'Toyota', 'Vinfast', 'Mercedes'
];

return (<ul>
    {cars.map(e => <Car brand={e} />)}
</ul>);
```

React Components

React components

- Independent and reusable bits of code
  - Components are what appear on the UI
  - Components return HTML
- Two types:
  - **Class component:** extends `React.Component` and has the `render()` method
  - **Function component:** a function which returns JSX, shorter code, behaves mostly like a Class component
- Difference:
  - You can add properties, methods, etc. to a Class component

Class components

- Constructor
  - Constructor inherited from `React.Component` receive HTML attributes as an argument (usually) called `props`
  - Can be overriden but not overloaded (no overloading in JS)
- State
  - `state` attribute and `setState()` method inherited from `React.Component`
  - React monitors the `state` object
  - When the `state` object changes, the component re-renders.

Creating a Class Component

- Create a component by extending `React.Component`. A component's properties should be kept in an object called `state`.
  - The `state` property is a special property.
  - A component re-renders if its `state` changes.

```
import React from 'react';
class Car extends React.Component {
    constructor() {
        super();
        this.state = { color: "red" };
    }
    render() {
        return <h2>I am a {this.state.color} Car!</h2>;
    }
}
```

React Component `props`

- Use an attribute to pass a color to a component, and use it in the JSX of that component.

```
import React from 'react';
class Car extends React.Component {
    render() {
        return <h2>I am a {this.props.color} Car!</h2>;
    }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car color="red" />);
```

`props` in the Constructor

- If a component has a constructor function, the `props` should always be passed to the constructor and also to the `React.Component` via the `super()` method.

```
import React from 'react';
class Car extends React.Component {
    constructor(props) {
        super(props);
    }
    render() {
        return <h2>I am a {this.props.model}!</h2>;
    }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car model="Mustang" />);
```

React Function Component

- Here is the same component from previous examples, but created using a *Function component* instead.

```
function Car(props) {
    return <h2>I am a {props.color} Car!</h2>;
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car color="red" />);
```

Nesting Components

- We can refer to components inside other components:

```
function Car() {
    return <h2>I am a Car!</h2>;
}
function Garage() {
    return (
        <>
            <h1>Who lives in my Garage?</h1>
            <Car />
        </>
    );
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```

Components in Files

- React is all about re-using code, and it is recommended to split your components into separate files.
- To do that, create a new `.js` file and put the component's code inside.
  - Note that the filename must start with an uppercase character.

```
function Car() {
    return <h2>Hi, I am a Car!</h2>;
}

export default Car;
```

React Styling

React CSS styling

- Three ways:
  - Inline styling
  - CSS stylesheets
  - CSS Modules

- Reference: https://www.w3schools.com/REACT/react_css_styling.asp

Inline CSS styling

- Assign a JS object to the style attribute:

```
<h1 style={{color: "red"}}>Hello Style!</h1>
```

- You see double curly braces because there is an object inside the JSX expression.
- CSS properties are in `camelCase`
  - Similar to CSS properties in JavaScript
  - See https://www.w3schools.com/jsref/dom_obj_style.asp

React CSS stylesheets

- You can `import` an external `.css` file

```
import './App.css';
```

- ...then style the web page based on tag names, `className` and `id` attributes of the tags/components.

React CSS modules

- Create the CSS module with the `.module.css` extension.
  - Example: `my-style.module.css`

```
.bigblue {
    color: DodgerBlue;
    padding: 40px;
    font-family: Sans-Serif;
    text-align: center;
}
```

- ...then import it into the `.js` file of your component:

```
import styles from './my-style.module.css';
```

- ...and use it:

```
const Car = () => {
    return <h1 className={styles.bigblue}>Hello Car!</h1>;
}
```

React events

- React has the same events as HTML
  - onLoad, onClick, onMouseOver, onChange, onSubmit...
  - These events can only be attached to synthetic elements (HTML elements, not components)

- Reference: https://reactjs.org/docs/events.html

Adding Events

- React events are written in `camelCase` syntax
  - `onClick` instead of `onclick`
- React event handlers are written inside curly braces
  - `onClick={shoot}` instead of `onClick="shoot()"`

```
function Football() {
    const shoot = () => {
        alert("Great Shot!");
    }
    return (
        <button onClick={shoot}>Take the shot!</button>
    );
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Football />);
```

React Hooks

What are React Hooks?

- Hooks allow function components to have access to state and other React features
  - Hooks make function components more powerful
  - Function components and Hooks can replace class components
- Hook rules:
  - Hooks can only be called inside React function components
  - Hooks can only be called at the top level of a component
  - Hooks cannot be conditional (*)

Function component update example (not gonna work)

```
function Board() {
  let status = 'Hi!';
  const clickHandler = () => {
    status = 'Updated!';
  }
  return (
    <div className="info">
      <div className="status">
        {status}
      </div>
      <button onClick={clickHandler}>Click me</button>
    </div>
  )
}
```

Function component update with useState hook (works)

```
import React, { useState } from 'react';

function Board() {
  const [status, updateStatus] = useState('Old value');
  const clickHandler = () => {
    updateStatus('Updated!');
  }
  return (
    <div className="info">
      <div className="status">
        {status}
      </div>
      <button onClick={clickHandler}>Click me</button>
    </div>
  )
}
```

More about useState in React

- State is created and managed separatedly for each Component instance.
  - Different instances of the same Component have different states
- Consider this example:

```
const [status, updateStatus] = useState('First value');
```

  - const is used although we plan to update the value later
  - **Reason:** status isn't modified directly (with the = sign)
  - When updateStatus is called, React will eventually re-load the Component (which means re-calling the Component function)

The useEffect Hook

- The useEffect Hook allows you to perform side effects in your components.
  - Examples of side effects: fetching data, directly updating the DOM, and timers...
- useEffect accepts two arguments (the 2nd is optional)

```
useEffect(<function>, <dependency>)
```

useEffect hook **timer** example

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

function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    setTimeout(() => {
      setCount((count) => count + 1);
    }, 1000);
  });

  return <h1>I've been rendered {count} times!</h1>;
}
```

Controlling when useEffect executes

- No dependency passed:

```
useEffect(() => {
   // Runs with every render
});
```

- An empty array:

```
useEffect(() => {
   // Runs only on the first render
}, []);
```

- Props or state variables:

```
useEffect(() => {
   // Runs on the first render
   // And any time any dependency value changes
}, [props.something, someStateVar]);
```