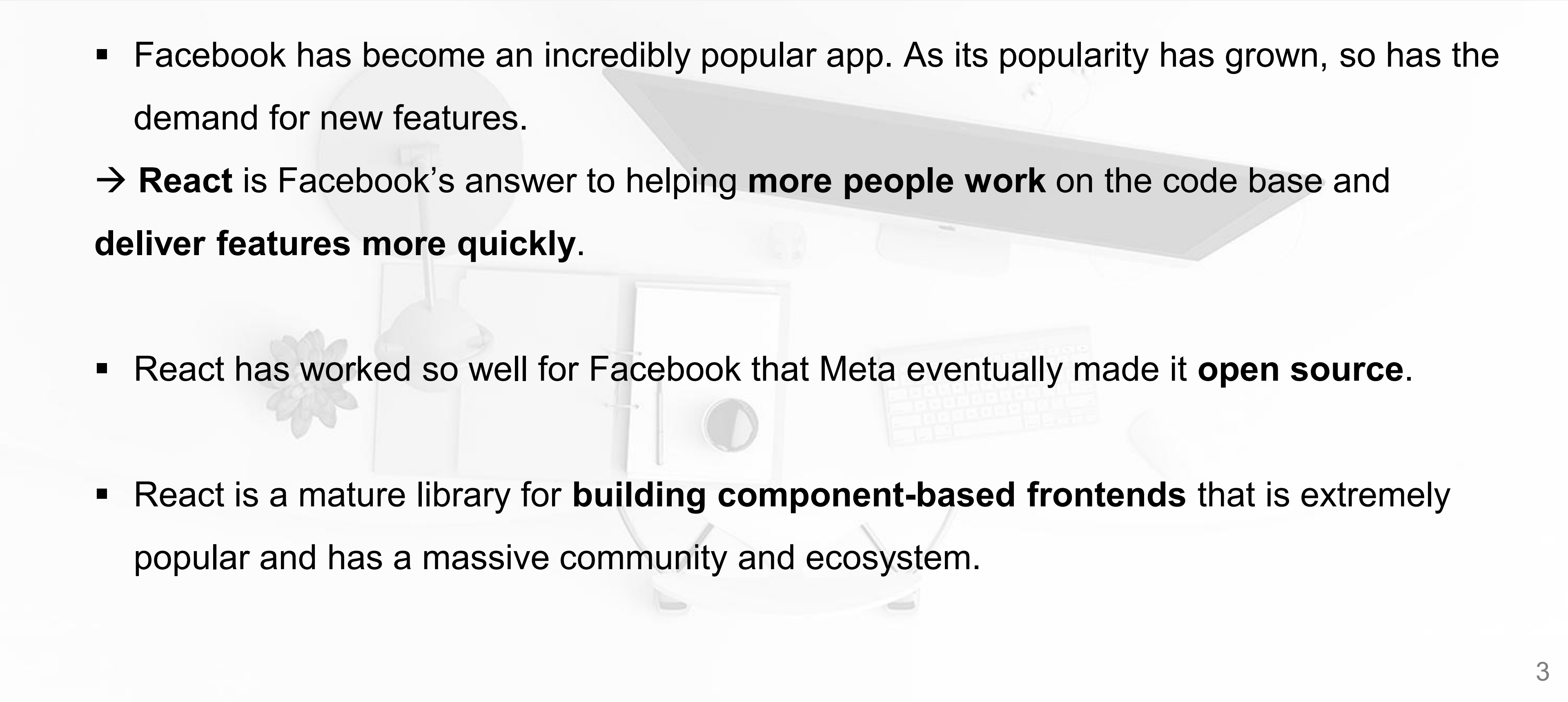
Advanced Web Programming

Ung Văn Giàu
Email: giau.ung@eiu.edu.vn

Introduction

Introducing React



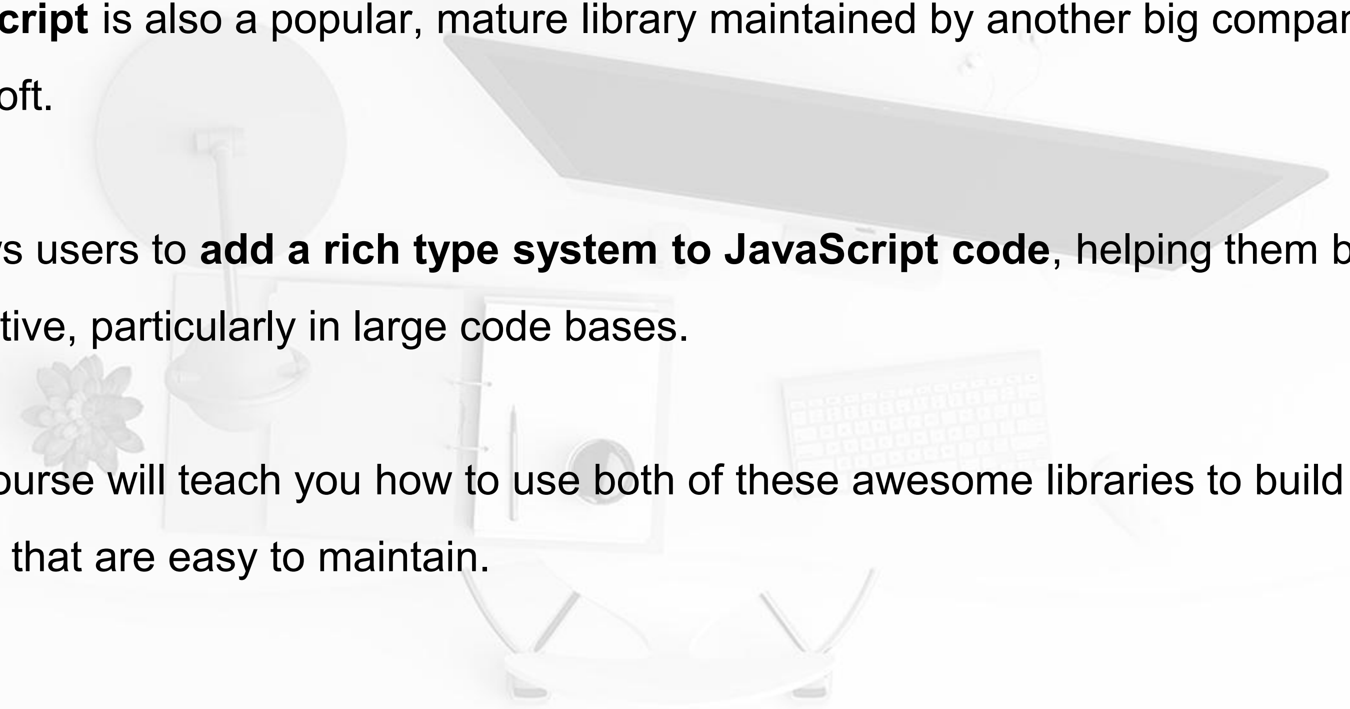
- Facebook has become an incredibly popular app. As its popularity has grown, so has the demand for new features.

→ **React** is Facebook's answer to helping **more people work** on the code base and **deliver features more quickly**.

- React has worked so well for Facebook that Meta eventually made it **open source**.

- React is a mature library for **building component-based frontends** that is extremely popular and has a massive community and ecosystem.

Introducing TypeScript



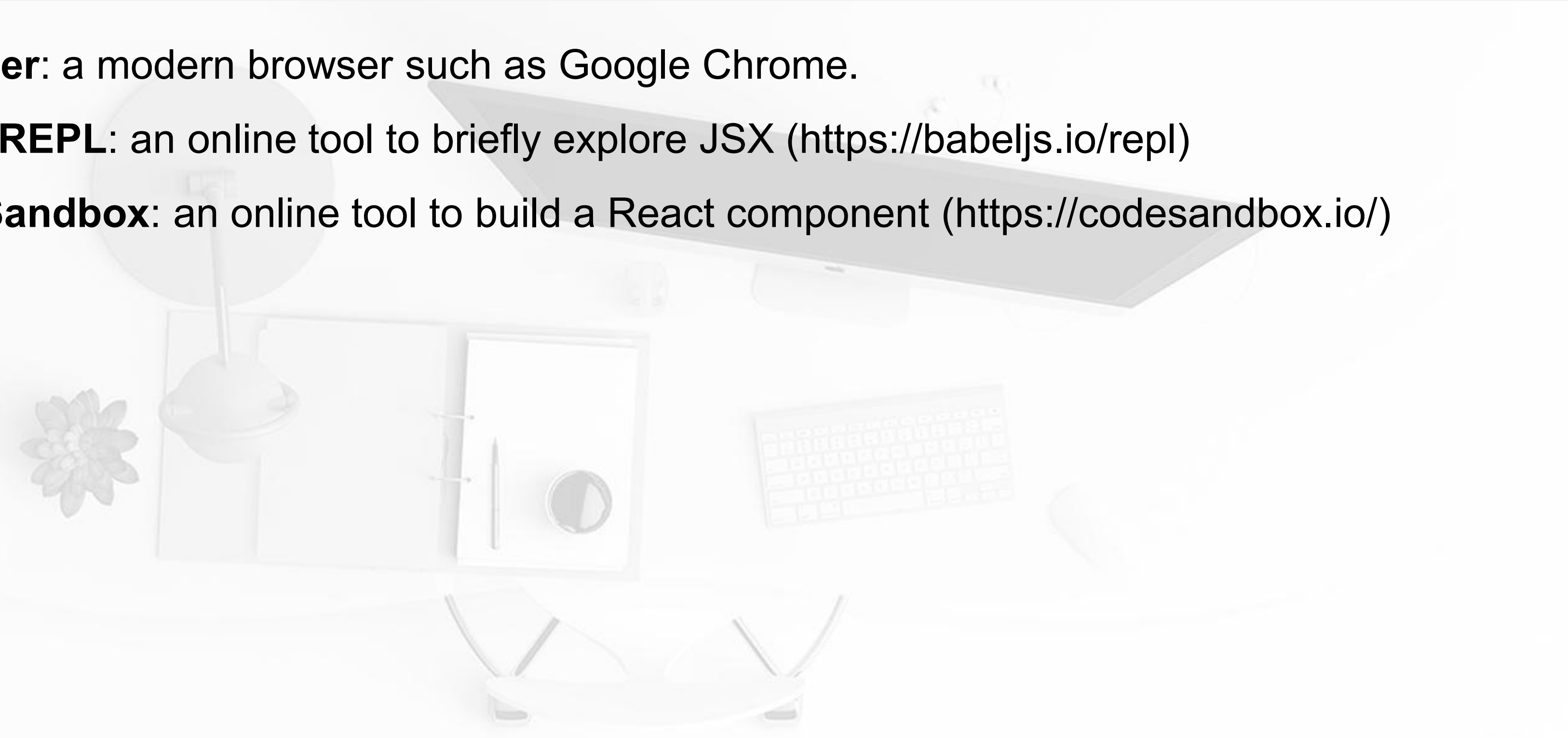
- **TypeScript** is also a popular, mature library maintained by another big company, Microsoft.

- It allows users to **add a rich type system to JavaScript code**, helping them be more productive, particularly in large code bases.

→ This course will teach you how to use both of these awesome libraries to build robust frontends that are easy to maintain.

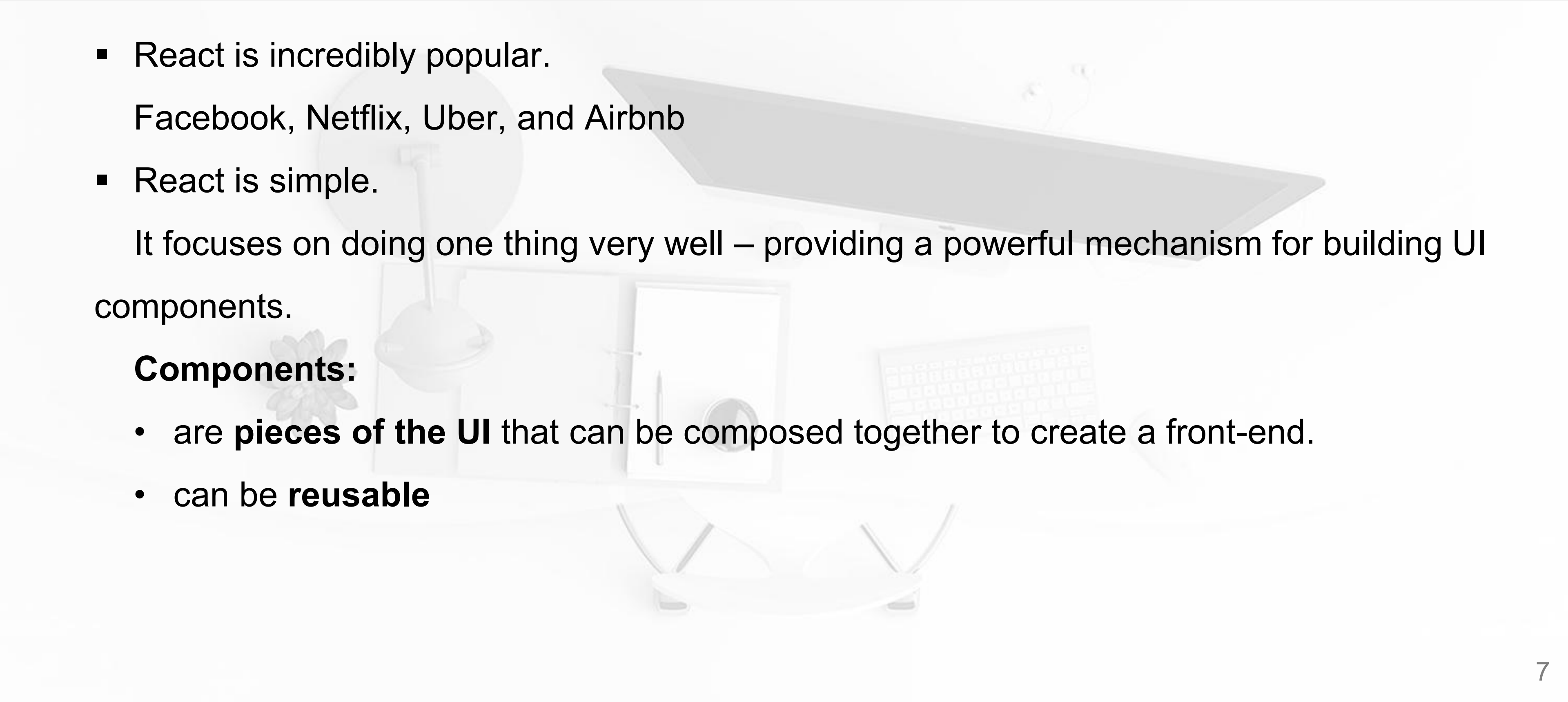
Contents

Technical requirements



- **Browser**: a modern browser such as Google Chrome.
- **Babel REPL**: an online tool to briefly explore JSX (https://babeljs.io/repl)
- **CodeSandbox**: an online tool to build a React component (https://codesandbox.io/)

1. Understanding the benefits of React



- React is incredibly popular.

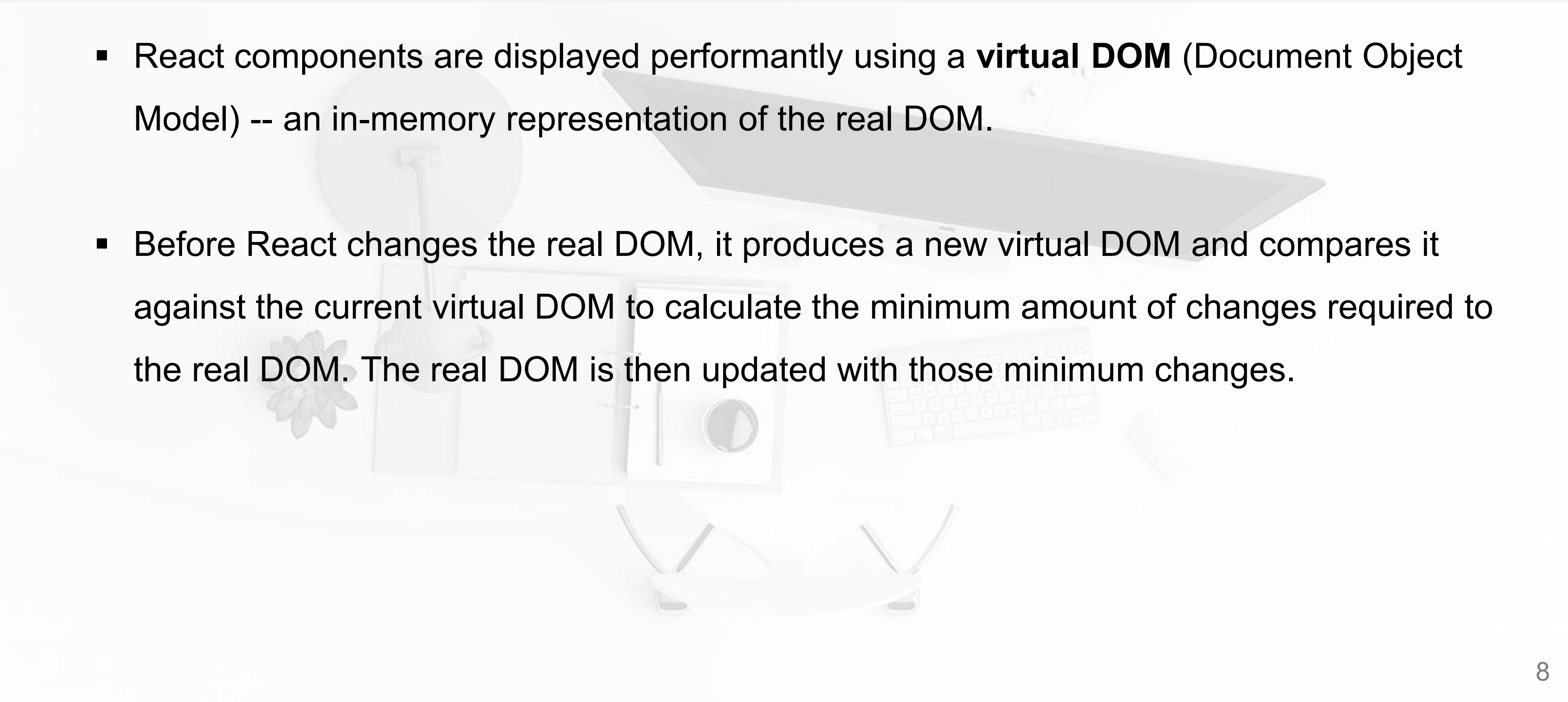
  Facebook, Netflix, Uber, and Airbnb
- React is simple.

  It focuses on doing one thing very well – providing a powerful mechanism for building UI components.

  Components:

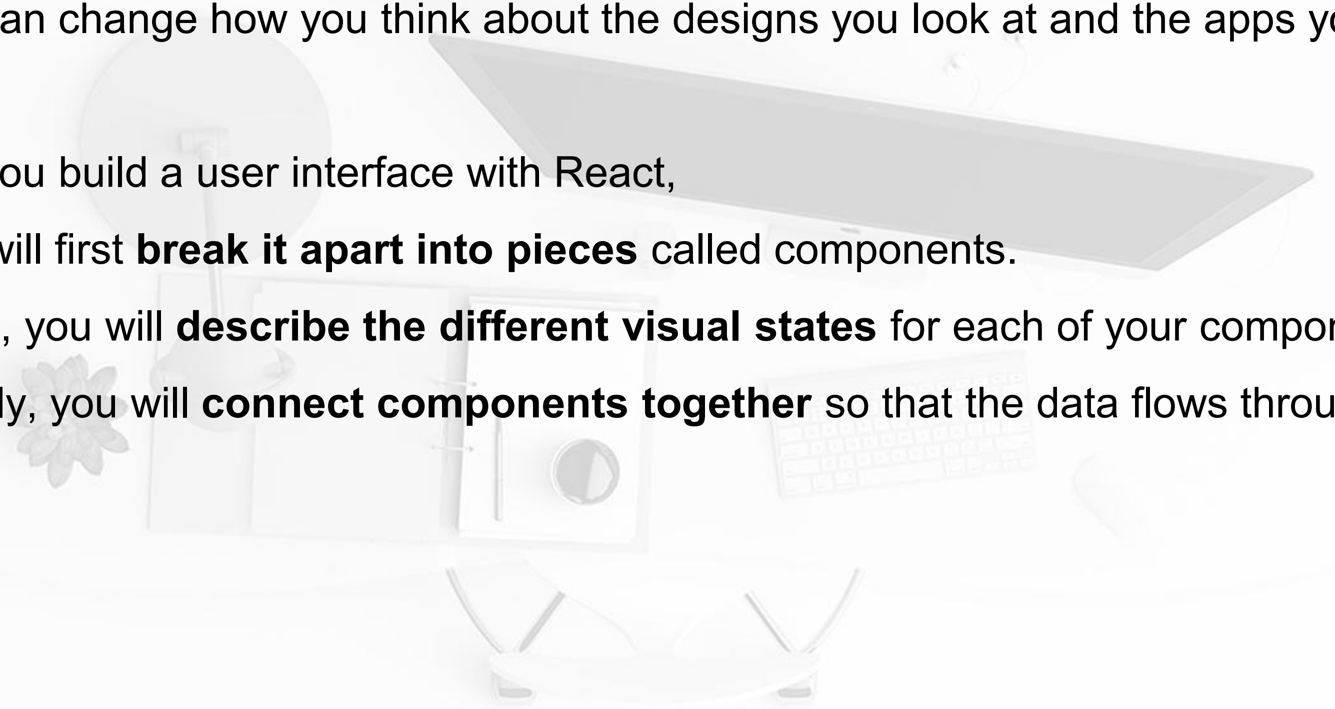
  - are **pieces of the UI** that can be composed together to create a front-end.
  - can be **reusable**

1. Understanding the benefits of React



- React components are displayed performantly using a **virtual DOM** (Document Object Model) -- an in-memory representation of the real DOM.

- Before React changes the real DOM, it produces a new virtual DOM and compares it against the current virtual DOM to calculate the minimum amount of changes required to the real DOM. The real DOM is then updated with those minimum changes.

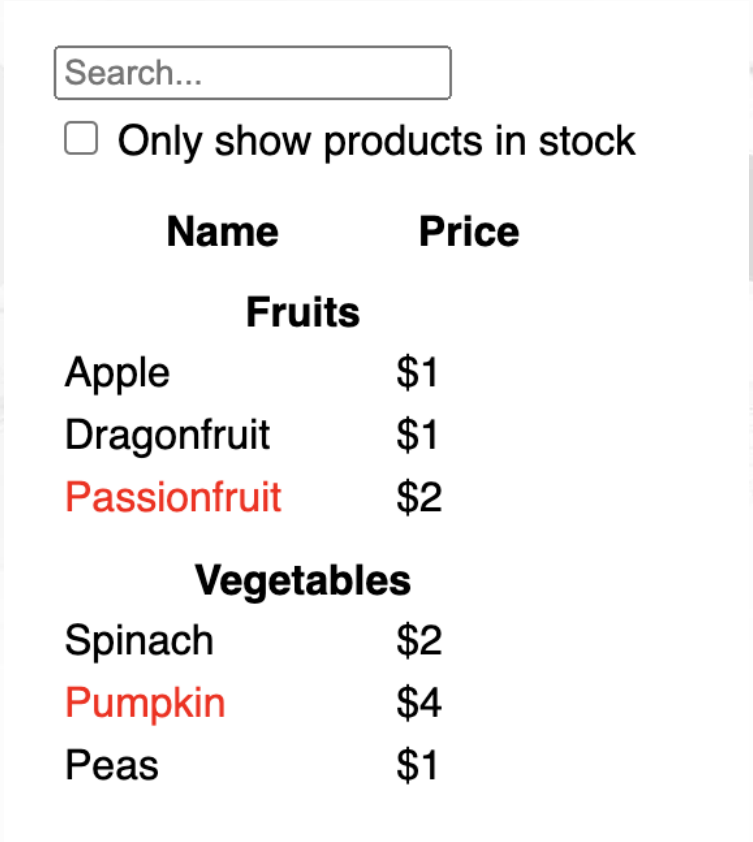


2. Thinking in React

- React can change how you think about the designs you look at and the apps you build.

- When you build a user interface with React,
  - you will first **break it apart into pieces** called components.
  - Then, you will **describe the different visual states** for each of your components.
  - Finally, you will **connect components together** so that the data flows through them.

2. Thinking in React

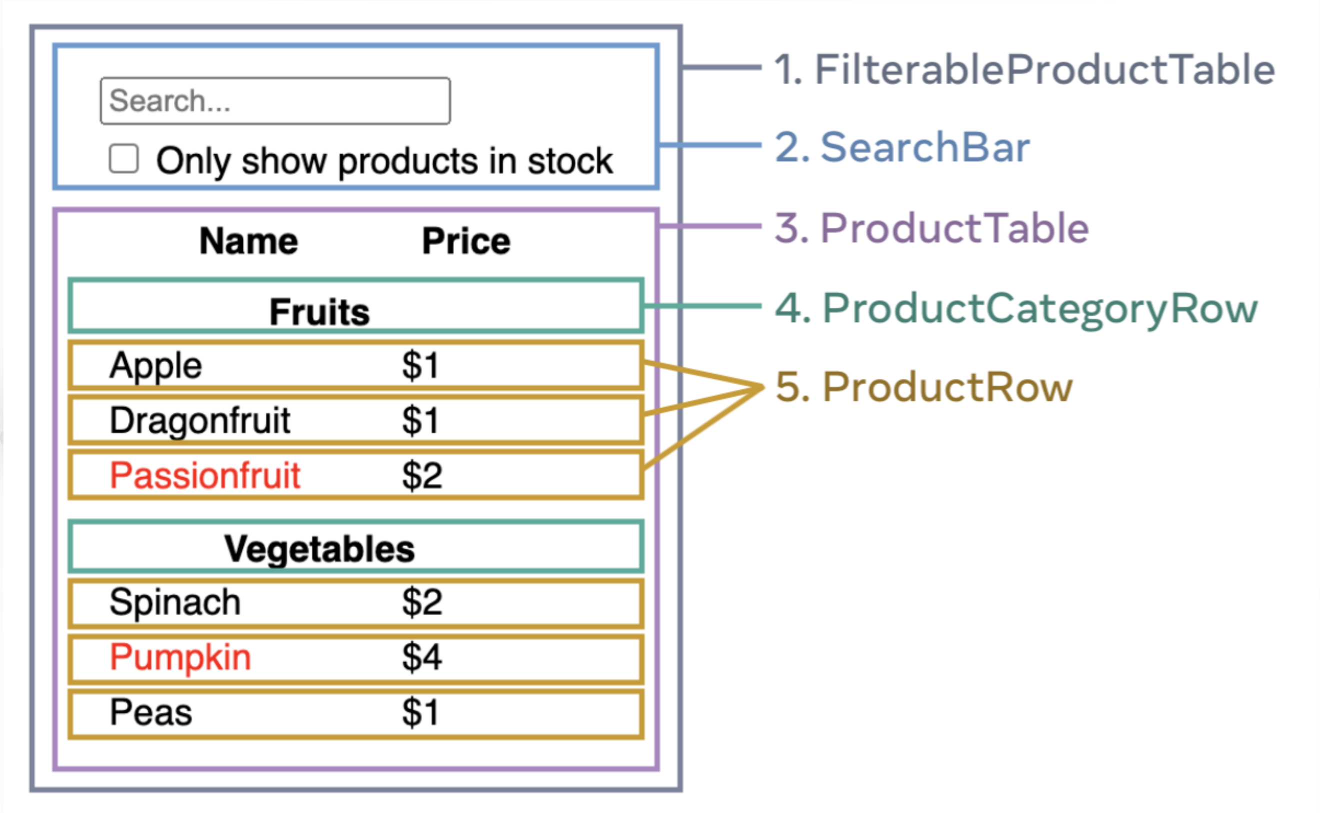


Search...

☐ Only show products in stock

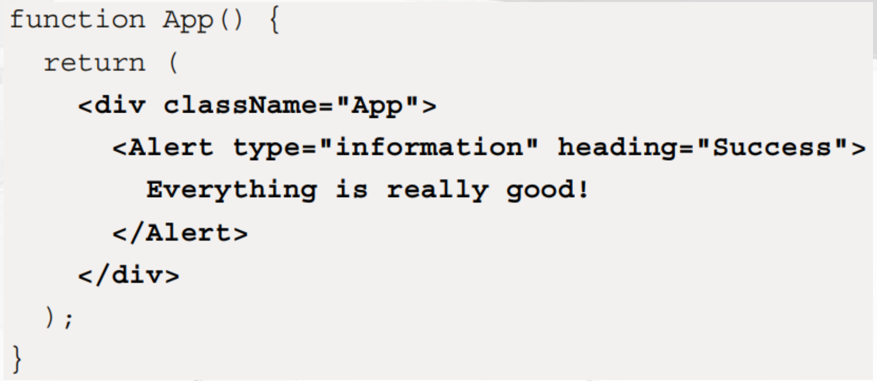
Name	Price
Fruits	
Apple	$1
Dragonfruit	$1
Passionfruit	$2
Vegetables	
Spinach	$2
Pumpkin	$4
Peas	$1

2. Thinking in React



3. Understanding JSX

- JSX is the **syntax used in a React component** to define what the component should display.
- JSX stands for JavaScript XML.
- The following code snippet is a React component with its JSX highlighted:

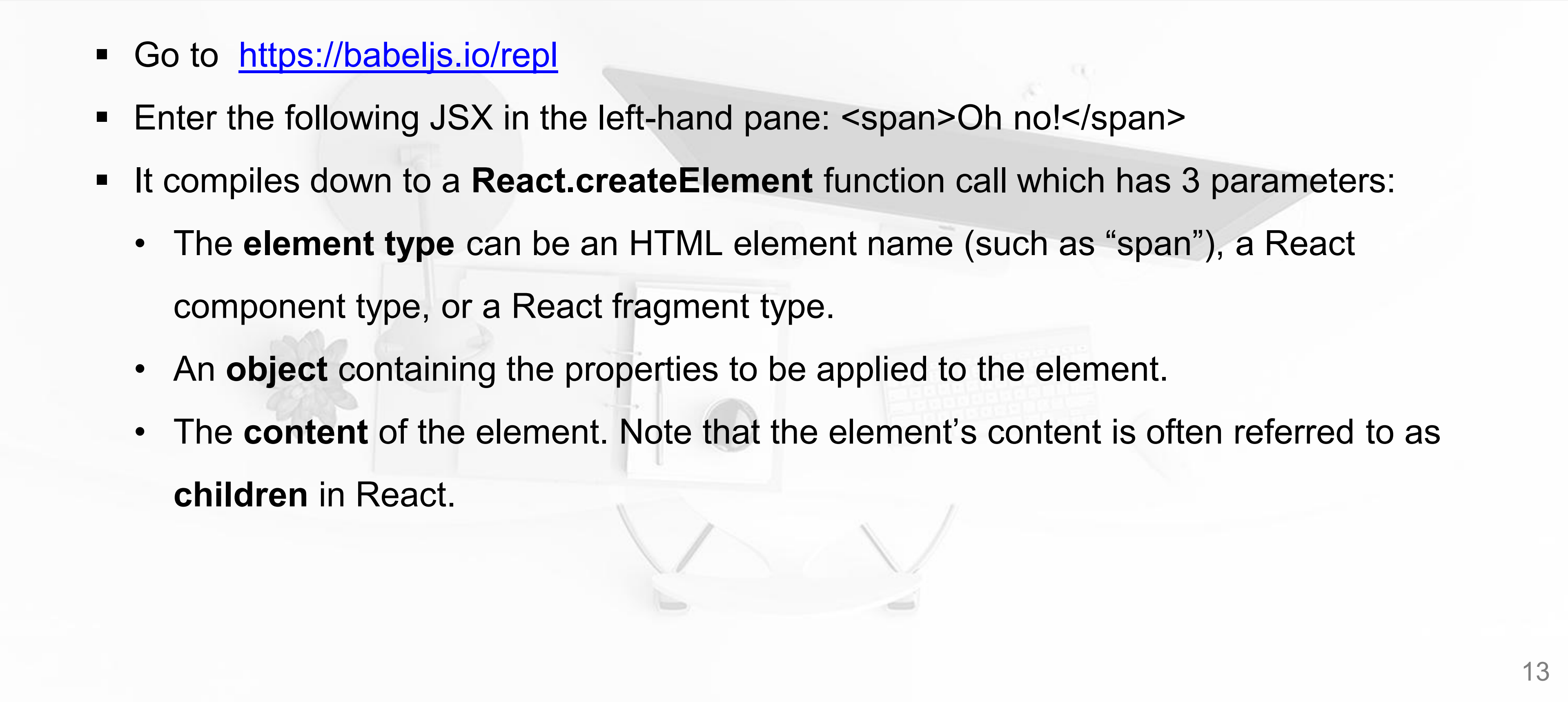


```
function App() {
  return (
    <div className="App">
      <Alert type="information" heading="Success">
        Everything is really good!
      </Alert>
    </div>
  );
}
```

- JSX is **a JavaScript syntax extension** → it needs to be transpiled to JavaScript first.
- Babel is a popular tool used to transpile JSX.

3. Understanding JSX

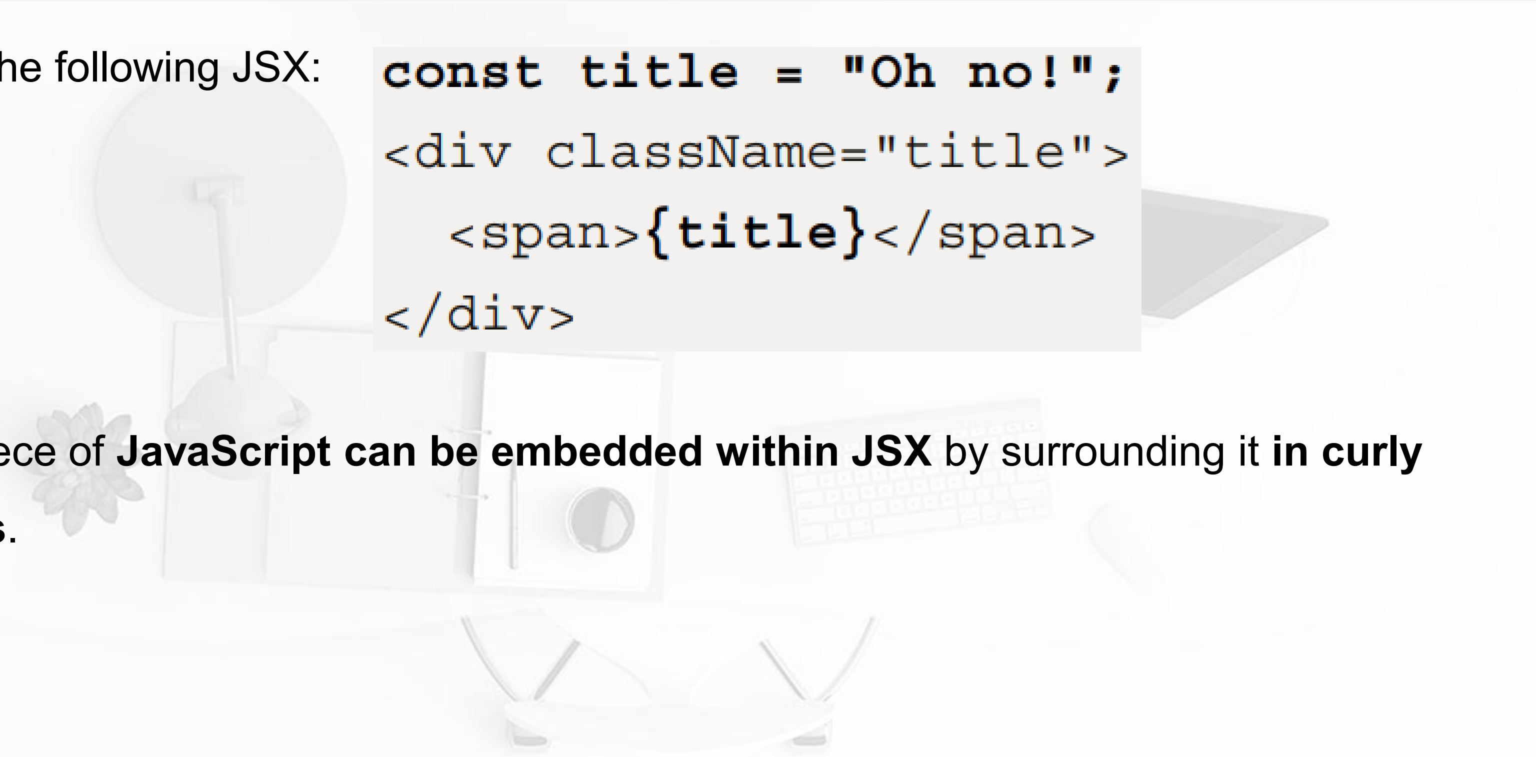
Example



- Go to https://babeljs.io/repl
- Enter the following JSX in the left-hand pane: <span>Oh no!</span>
- It compiles down to a **React.createElement** function call which has 3 parameters:
  - The **element type** can be an HTML element name (such as “span”), a React component type, or a React fragment type.
  - An **object** containing the properties to be applied to the element.
  - The **content** of the element. Note that the element’s content is often referred to as **children** in React.

3. Understanding JSX

Example

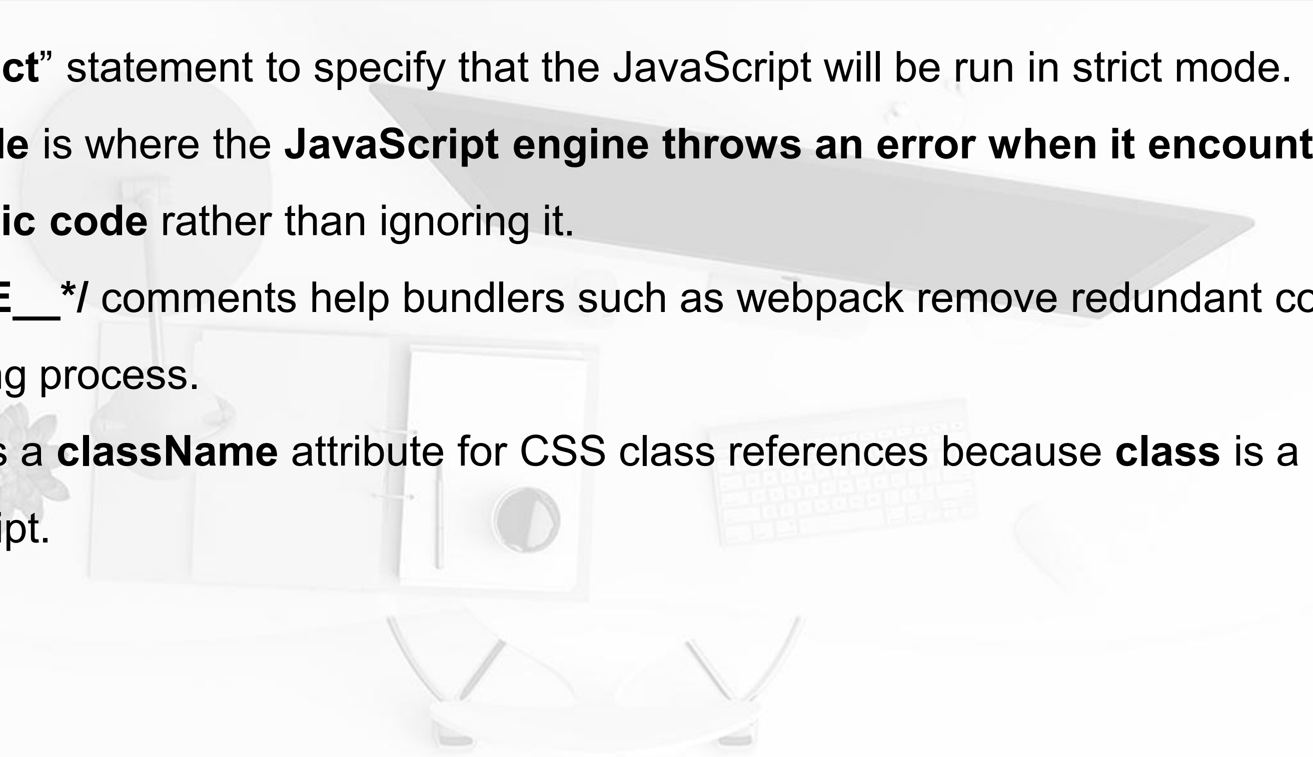


- Enter the following JSX:

```
const title = "Oh no!";
<div className="title">
  <span>{title}</span>
</div>
```

- Any piece of **JavaScript can be embedded within JSX** by surrounding it **in curly braces**.

3. Understanding JSX



- A “**use strict**” statement to specify that the JavaScript will be run in strict mode. **Strict mode** is where the **JavaScript engine throws an error when it encounters problematic code** rather than ignoring it.
- **/*#__PURE__*/** comments help bundlers such as webpack remove redundant code in the bundling process.
- React uses a **className** attribute for CSS class references because **class** is a keyword in JavaScript.

3. Understanding JSX

Example

- Add the following ternary expression:

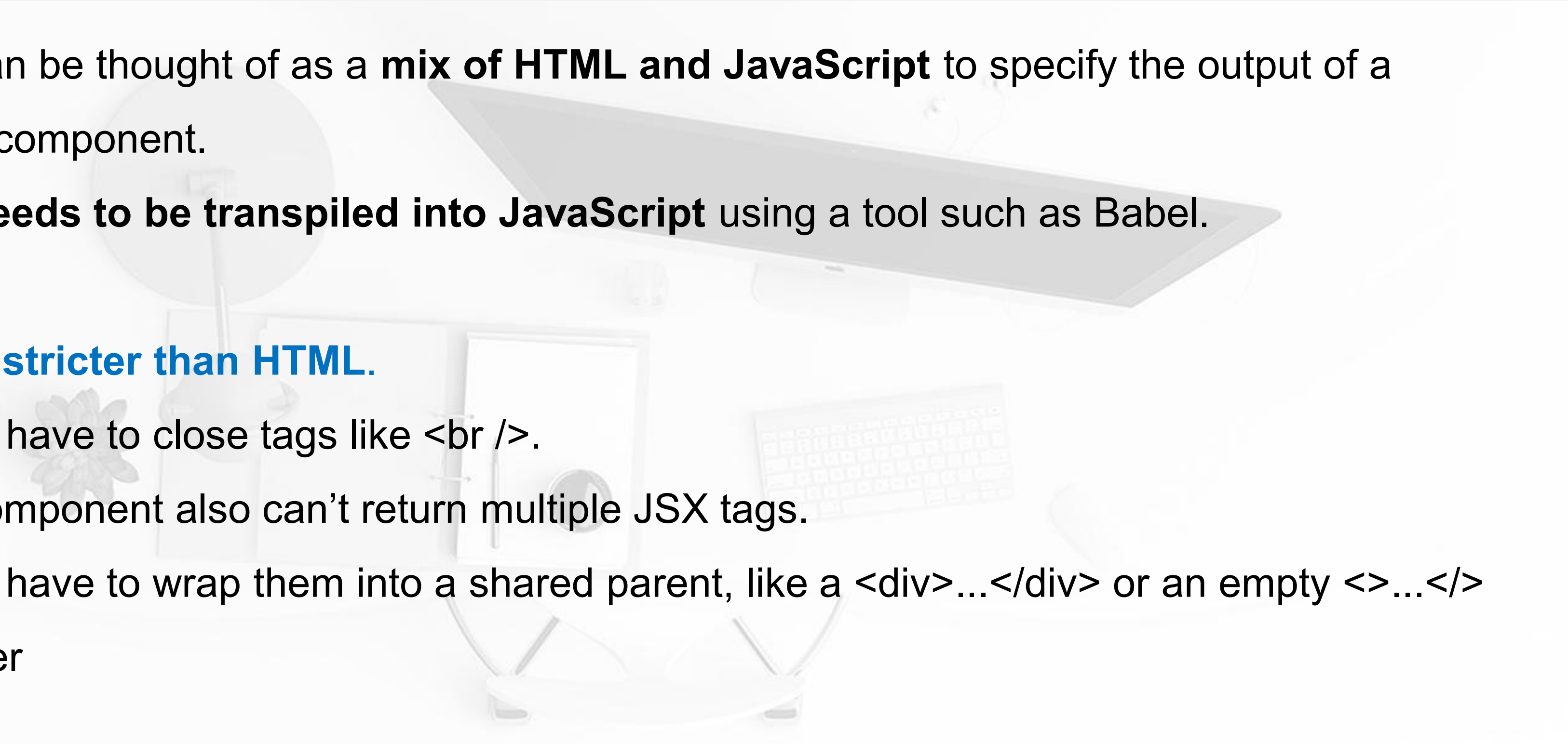
```
const title = "Oh no!";
<div className="title">
  <span>{title ? title : "Something important"}</span>
</div>
```

- A ternary expression is an inline conditional statement in JavaScript.

  It starts with the **condition followed by ?**, then **what returns** when the condition is **true** followed by **:**, and finally, **what returns** when the condition is **false**.

3. Understanding JSX

Recap



- JSX can be thought of as a **mix of HTML and JavaScript** to specify the output of a React component.
- **JSX needs to be transpiled into JavaScript** using a tool such as Babel.

- JSX is **stricter than HTML**.
  - You have to close tags like <br />.
  - A component also can't return multiple JSX tags.

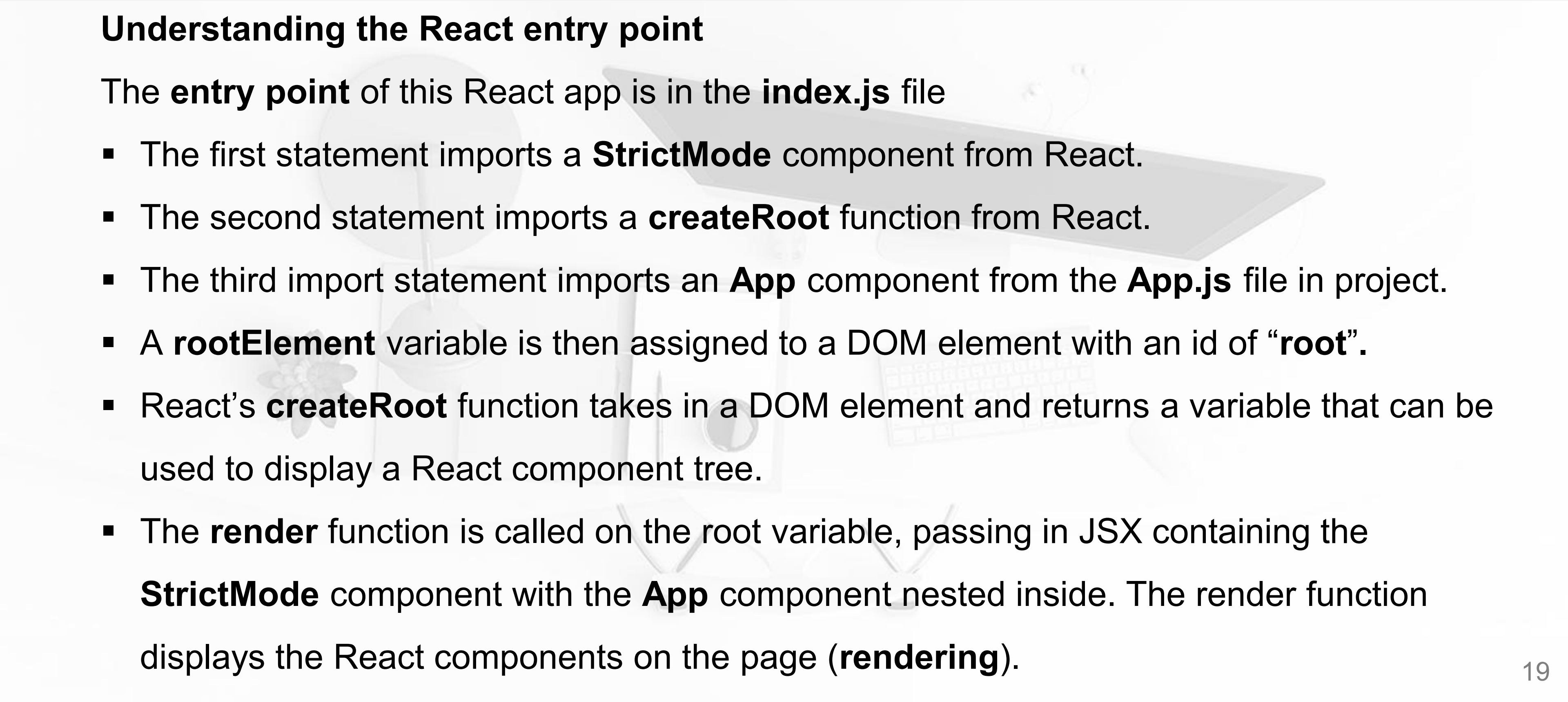
    You have to wrap them into a shared parent, like a <div>...</div> or an empty <>...</> wrapper

4. Creating a component

Creating a CodeSandbox project

- CodeSandbox help us create a React project at the click in a web browser and then focus on how to create a React component.
- Go to https://codesandbox.io/ > click the **Create Sandbox** button > Click the **React** template.
- There are three main panels in the CodeSandbox editor:
  - The **Files** panel: contains all the files in the project.
  - The **Code editor** panel: contains the code.
  - The **Browser** panel: displays a preview of the running app.

4. Creating a component

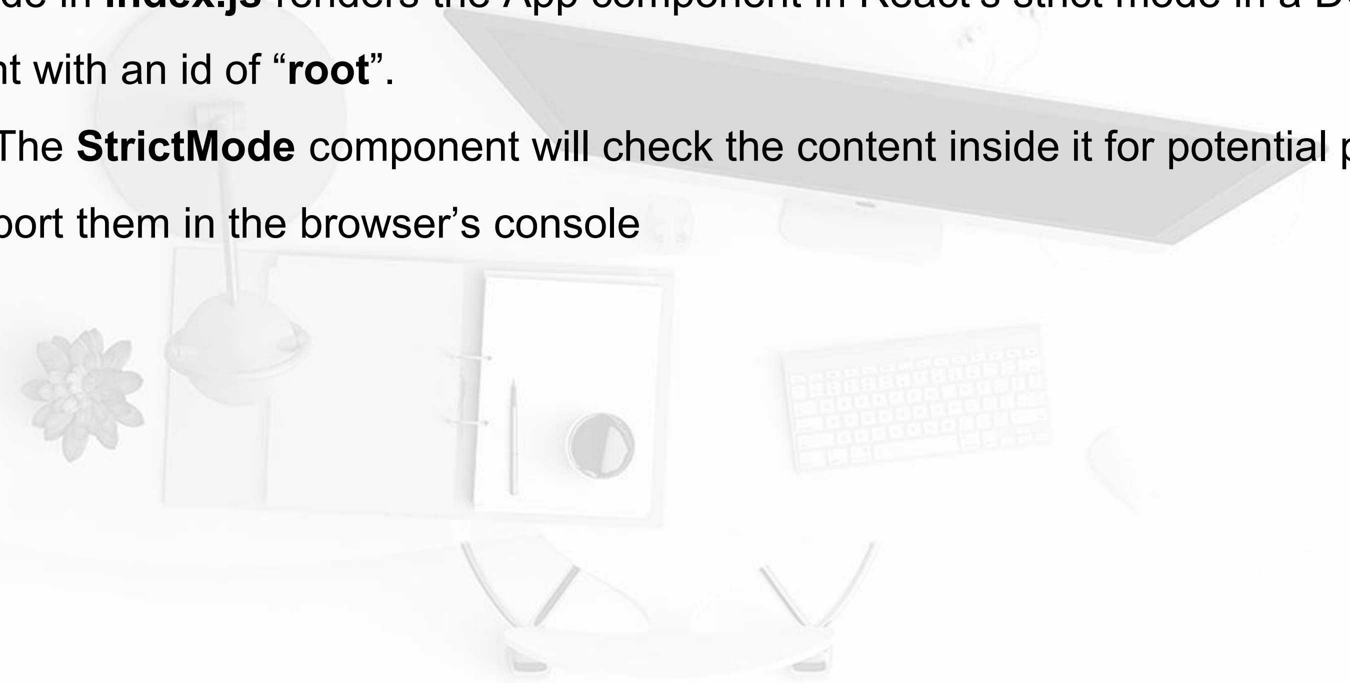


Understanding the React entry point

The **entry point** of this React app is in the **index.js** file

- The first statement imports a **StrictMode** component from React.
- The second statement imports a **createRoot** function from React.
- The third import statement imports an **App** component from the **App.js** file in project.
- A **rootElement** variable is then assigned to a DOM element with an id of "**root**".
- React's **createRoot** function takes in a DOM element and returns a variable that can be used to display a React component tree.
- The **render** function is called on the root variable, passing in JSX containing the **StrictMode** component with the **App** component nested inside. The render function displays the React components on the page (**rendering**).

4. Creating a component



- The code in **index.js** renders the App component in React’s strict mode in a DOM element with an id of “**root**”.
- **Note:** The **StrictMode** component will check the content inside it for potential problems and report them in the browser’s console

4. Creating a component

Understanding the React component tree

- A React app is structured in a tree of components and DOM elements.
- The root component is the component at the top of the tree.
- React components can be nested inside another React component.
- React components can reference one or more other components, and even DOM elements

```
StrictMode
└── App
    └── div
        └── h1
            └── h2
```

4. Creating a component

Creating a basic alert component

Description: A component displays an alert, which we will simply call Alert. It will consist of an icon, a heading, and a message.

Note:

- A **React component name must start with a CAPITAL letter**.
- The **filename** for component files isn't important, and usually uses the **same name as the component**, either in **Pascal** (ProductList) or **snake case** (product_list).
- The file extension must be **.js** or **.jsx** for React transpilers to recognize these as React components.

4. Creating a component

Example

In the Files panel, right-click on the **src** folder and choose **Create File** > enter the component filename (**Alert.js**) > Enter the following code.

```
function Alert() {
  return (
    <div>
      <div>
        <span role="img" aria-label="Warning">⚠</span>
        <span>Oh no!</span>
      </div>
      <div>Something isn't quite right ...</div>
    </div>
  );
}
```

4. Creating a component

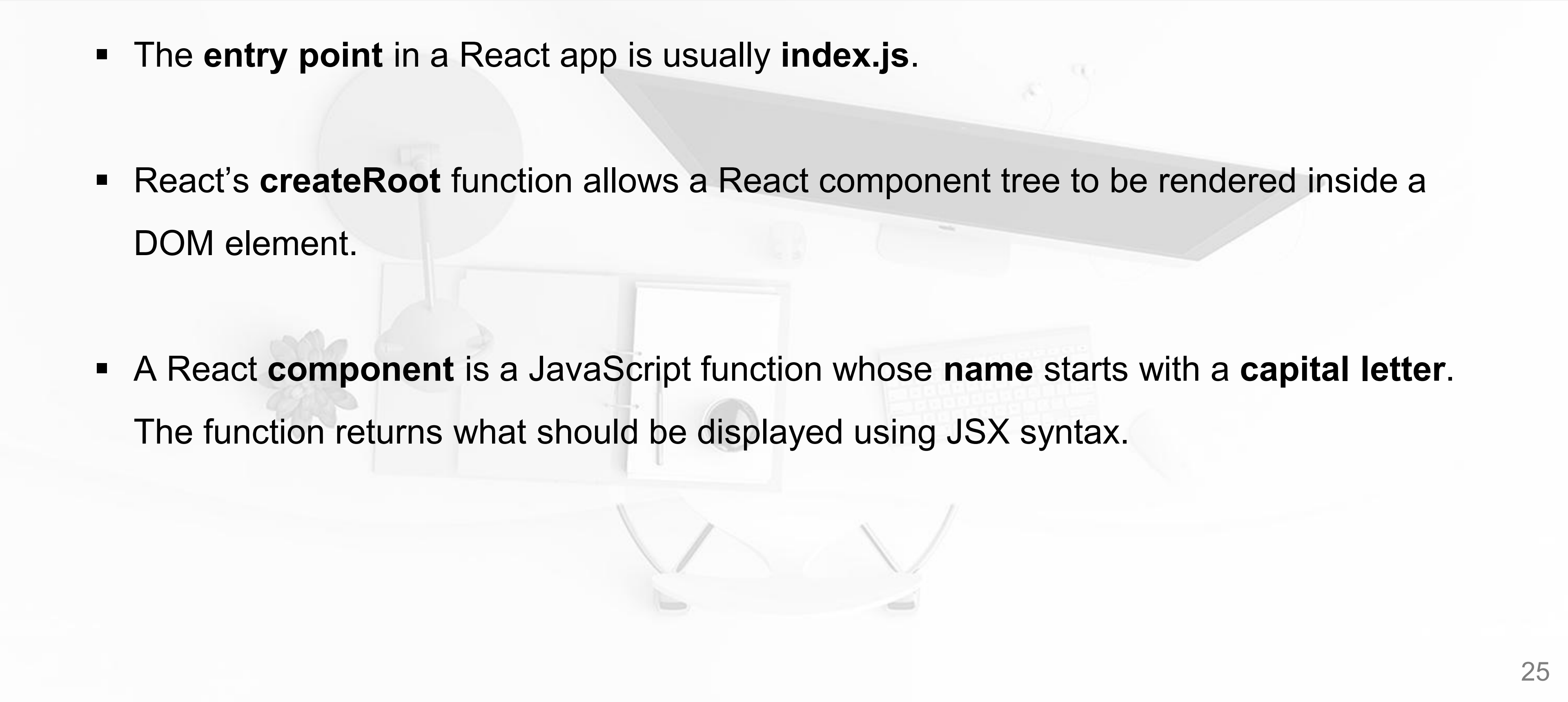
Example

A React component can be implemented using **arrow function syntax**.

```
const Alert = () => {
  return (
    <div>
      <div>
        <span role="img" aria-label="Warning">
          ⚠
        </span>
        <span>Oh no!</span>
      </div>
      <div>Something isn't quite right ...</div>
    </div>
  );
};
```

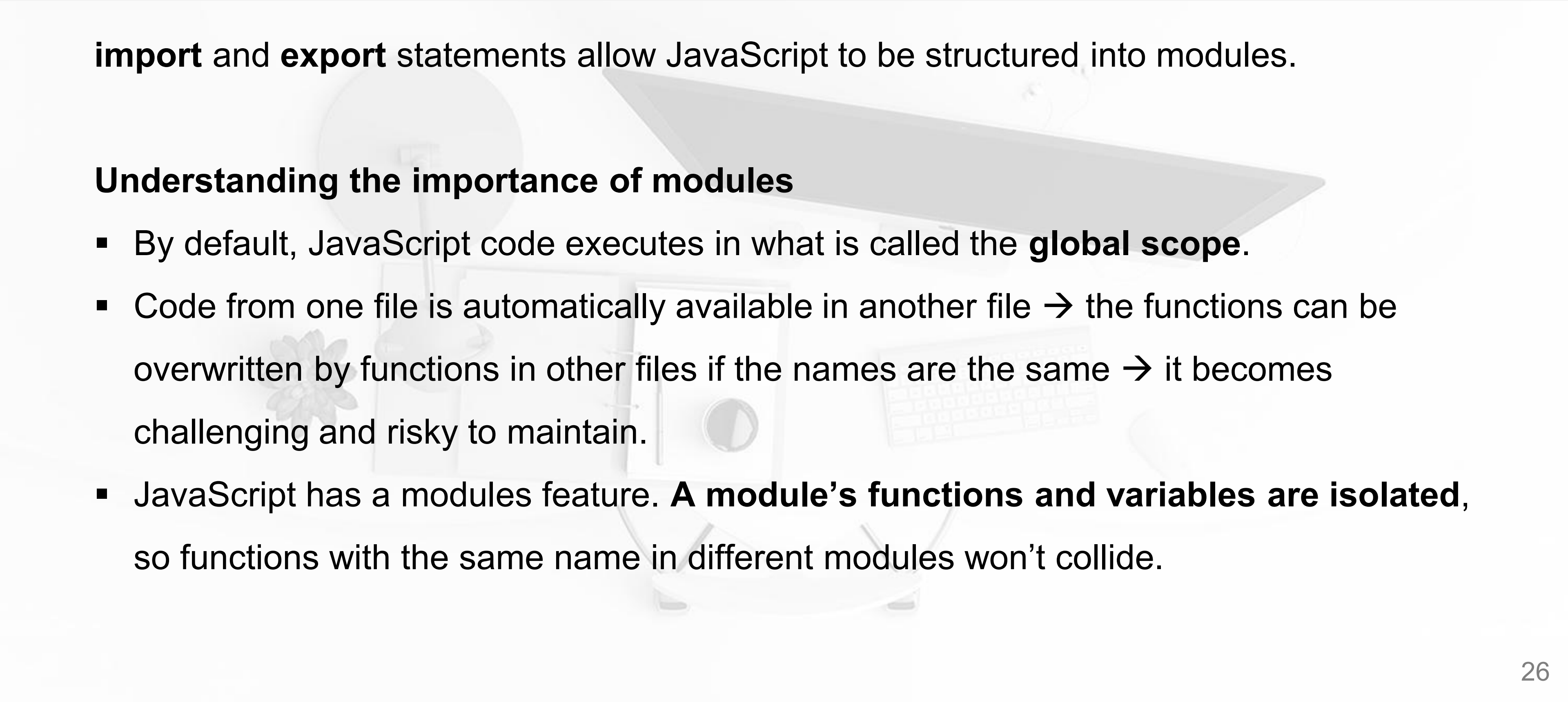
4. Creating a component

Recap



- The **entry point** in a React app is usually **index.js**.
- React’s **createRoot** function allows a React component tree to be rendered inside a DOM element.
- A React **component** is a JavaScript function whose **name** starts with a **capital letter**. The function returns what should be displayed using JSX syntax.

5. Understanding imports and exports

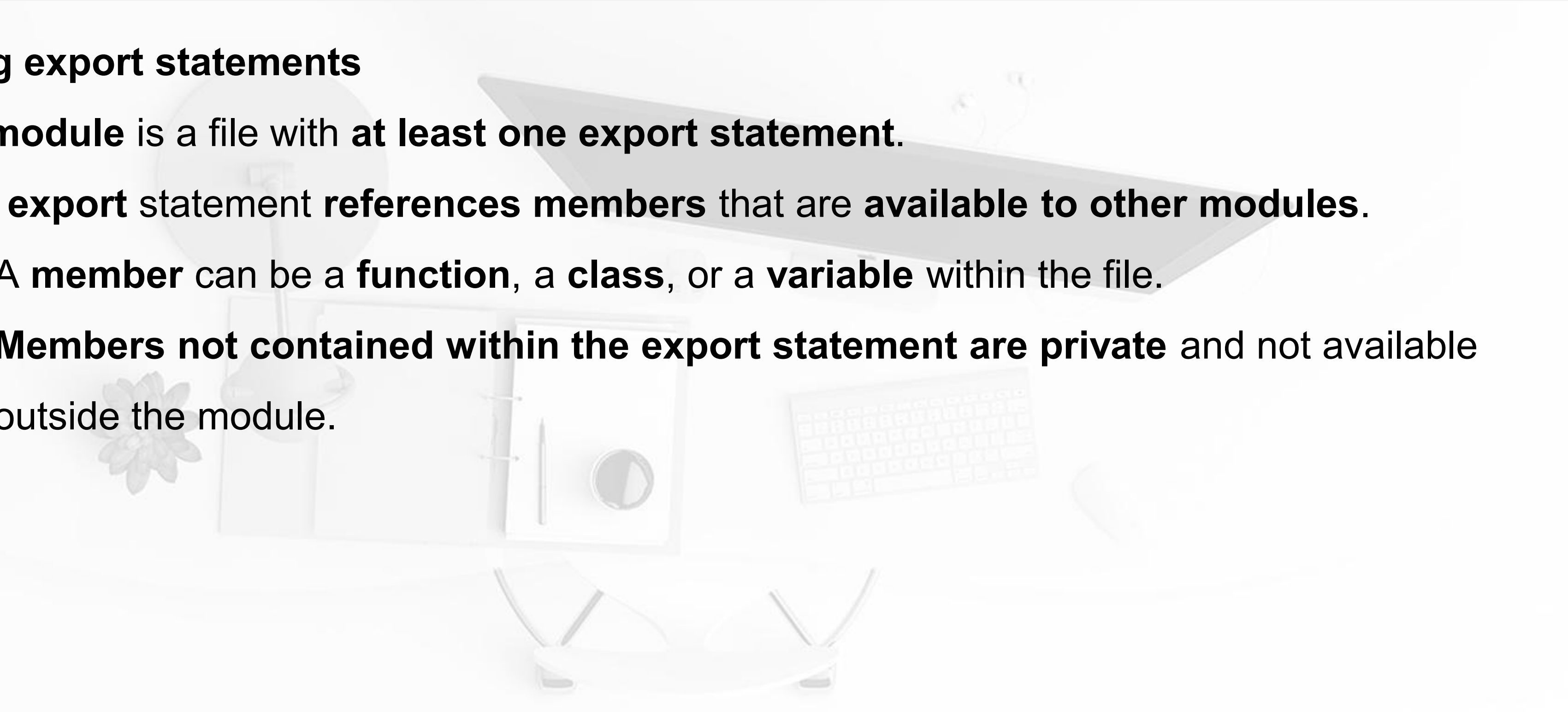


import and **export** statements allow JavaScript to be structured into modules.

Understanding the importance of modules

- By default, JavaScript code executes in what is called the **global scope**.
- Code from one file is automatically available in another file → the functions can be overwritten by functions in other files if the names are the same → it becomes challenging and risky to maintain.
- JavaScript has a modules feature. **A module's functions and variables are isolated**, so functions with the same name in different modules won't collide.

5. Understanding imports and exports



Using export statements

- A **module** is a file with **at least one export statement**.
- An **export** statement **references members** that are **available to other modules**.
  - A **member** can be a **function**, a **class**, or a **variable** within the file.
  - **Members not contained within the export statement are private** and not available outside the module.

5. Understanding imports and exports

Using export statements

- Example of a **named export statement**

```
function myFunc1() {
    ...
}
function myFunc2() {
    ...
}
function myFunc3() {
    ...
}
export { myFunc1, myFunc3 };
```

5. Understanding imports and exports

Using export statements

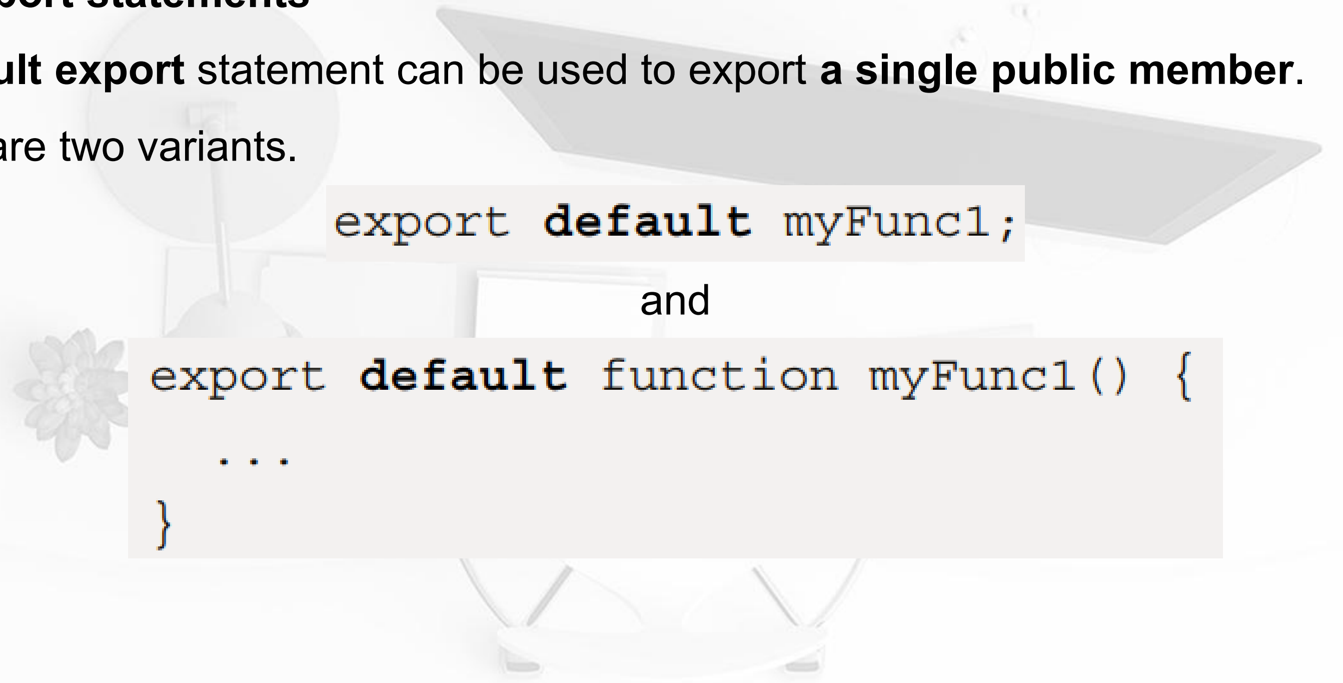
- The **export keyword can be added before the function keyword** on the public functions.

```
export function myFunc1() {
  ...
}

function myFunc2() {
  ...
}

export function myFunc3() {
  ...
}
```

5. Understanding imports and exports



Using export statements

- A **default export** statement can be used to export **a single public member**.
- There are two variants.

```
export default myFunc1;
```

and

```
export default function myFunc1() {
   ...
}
```

5. Understanding imports and exports

Using import statements

- Using an **import** statement **allows public members from a module to be used**.
- There are named and default import statements.
- Example of a **default import statement**:

```
import myFunc1 from './myModule';
```

- Example of a **named import statement**:

```
import { myFunc1, myFunc3 } from './myModule';
```

- **Note:**

  Unlike default imports, the names of imported members must match the exported members.

5. Understanding imports and exports

Adding Alert to the App component

- Open **Alert.js** and add the **export** keyword before the **Alert** function
- Open **App.js** and add the **import** statement at the top of the file

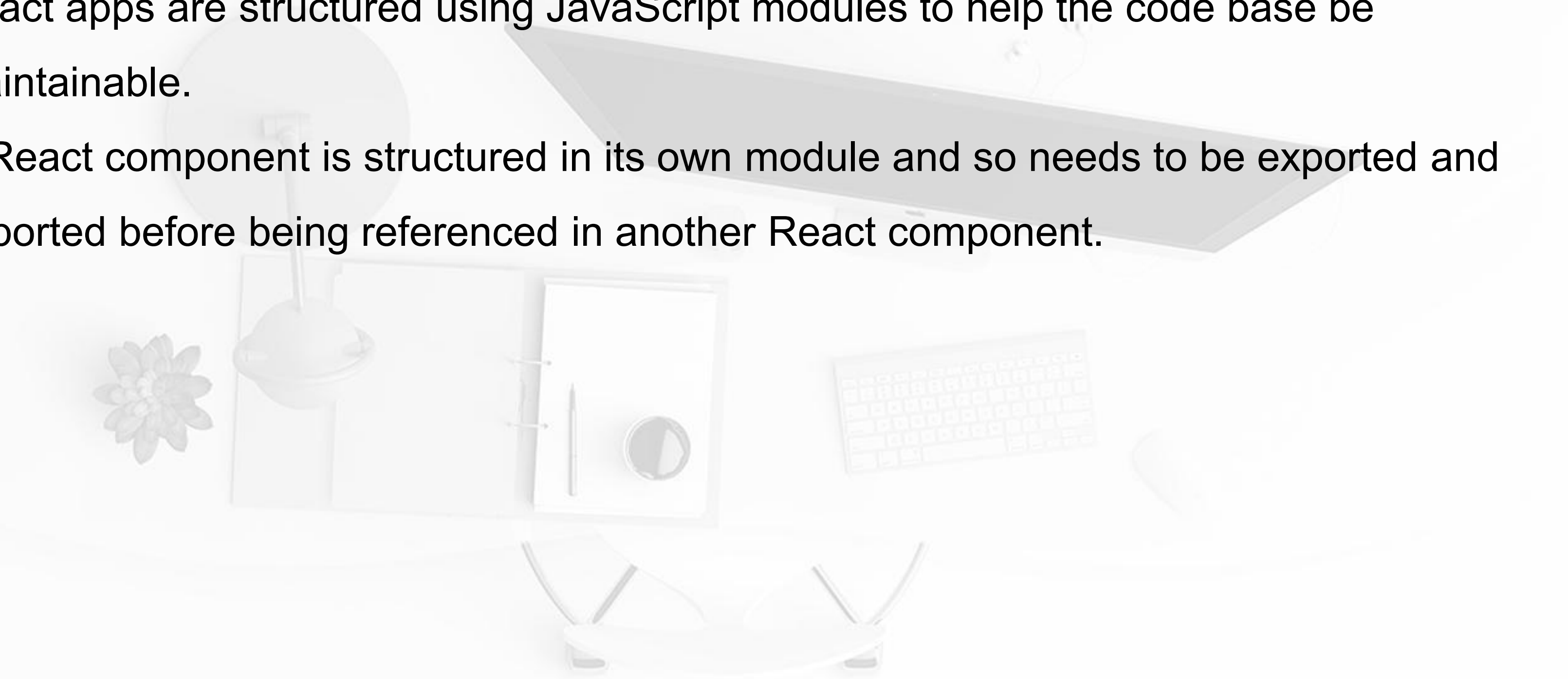
  import { Alert } from './Alert';

- Reference **Alert** in the **App** component's JSX

```
export default function App() {
  return (
    <div className="App">
      <Alert />
    </div>
  );
}
```

5. Understanding imports and exports

Recap



- React apps are structured using JavaScript modules to help the code base be maintainable.
- A React component is structured in its own module and so needs to be exported and imported before being referenced in another React component.

6. Using props

Props is short for ***properties.***

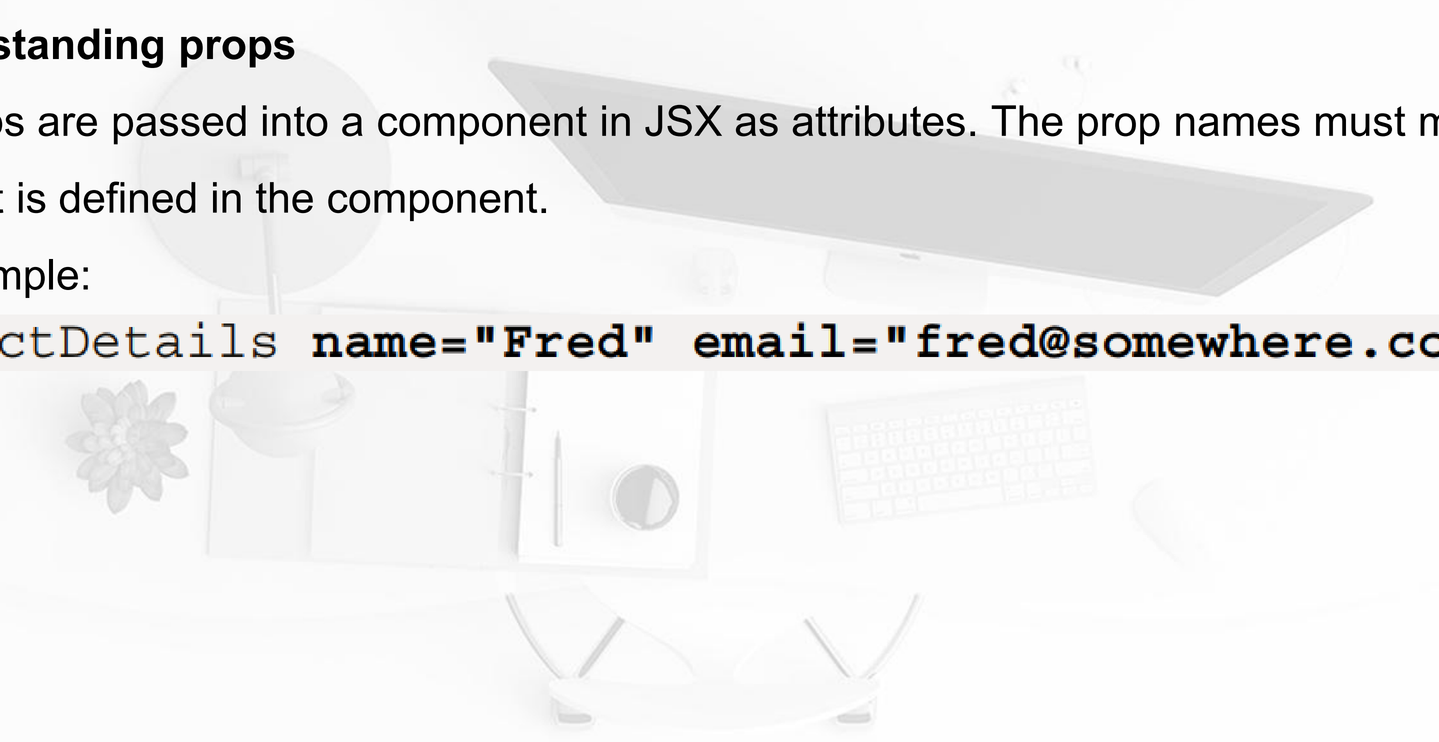
Understanding props

- **props** is an **optional parameter** that is **passed into a React component**. This parameter is **an object** containing the properties.
- The following example shows a props parameter in a ContactDetails component:

```
function ContactDetails(props) {
  console.log(props.name);
  console.log(props.email);
  ...
}
```

- **Note:** The parameter doesn’t have to be named props, but it is common practice.

6. Using props

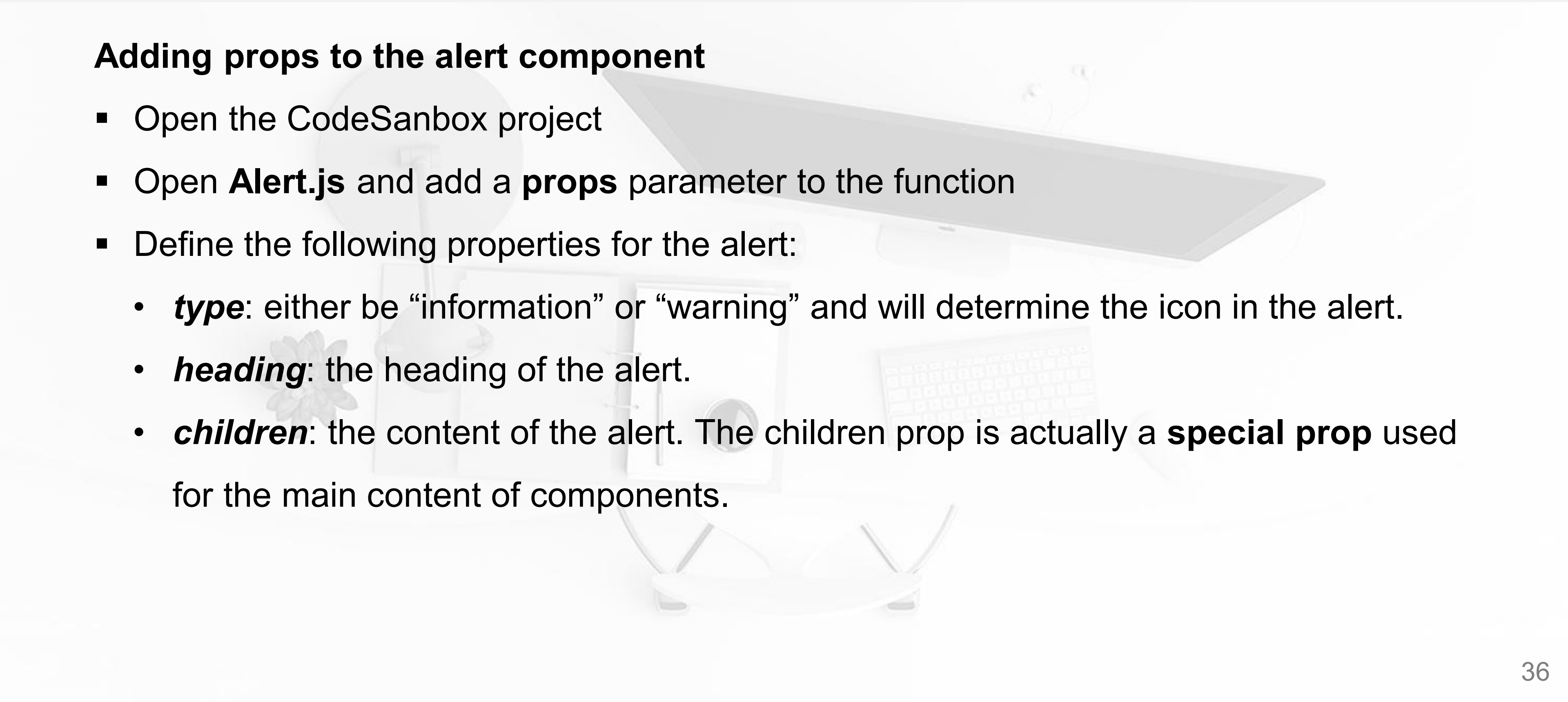


Understanding props

- Props are passed into a component in JSX as attributes. The prop names must match what is defined in the component.
- Example:

```
<ContactDetails name="Fred" email="fred@somewhere.com" />
```

6. Using props



Adding props to the alert component

- Open the CodeSanbox project
- Open **Alert.js** and add a **props** parameter to the function
- Define the following properties for the alert:
  - ***type***: either be “information” or “warning” and will determine the icon in the alert.
  - ***heading***: the heading of the alert.
  - ***children***: the content of the alert. The children prop is actually a **special prop** used for the main content of components.

6. Using props

Adding props to the alert component

- Update the alert component’s JSX to use the props

```
export function Alert(props) {
  return (
    <div>
      <div>
        <span
          role="img"
          aria-label={
            props.type === "warning"
              ? "Warning"
              : "Information"
          }
        >
          {props.type === "warning" ? "⚠" : "i"}
        </span>
        <span>{props.heading}</span>
      </div>
      <div>{props.children}</div>
    </div>
  );
}
```

6. Using props

Adding props to the alert component

- Open **App.js** and update the **Alert** component in the JSX to pass in props

```
<Alert type="information" heading="Success">
  Everything is really good!
</Alert>
```

6. Using props

Destructuring is a JavaScript feature **that allows properties to be unpacked from an object**.

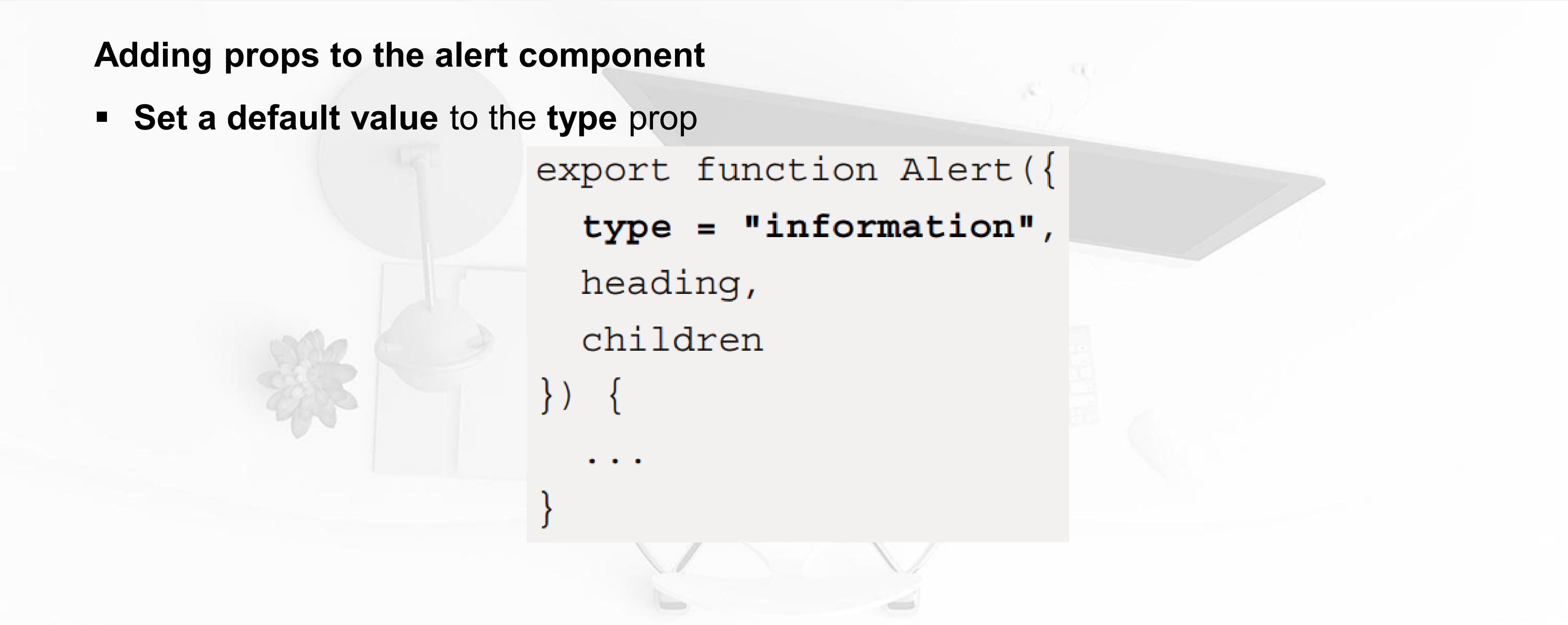
Adding props to the alert component

- Clean up the alert component code by destructuring the props parameter

```
export function Alert({ type, heading, children }) {
```

```
<div>
  <span
    role="img"
    aria-label={
      type === "warning" ? "Warning" :
        "Information"
    }
  >
    {type === "warning" ? "⚠" : "i"}
  </span>
  <span>{heading}</span>
</div>
<div>{children}</div>
```

6. Using props



Adding props to the alert component

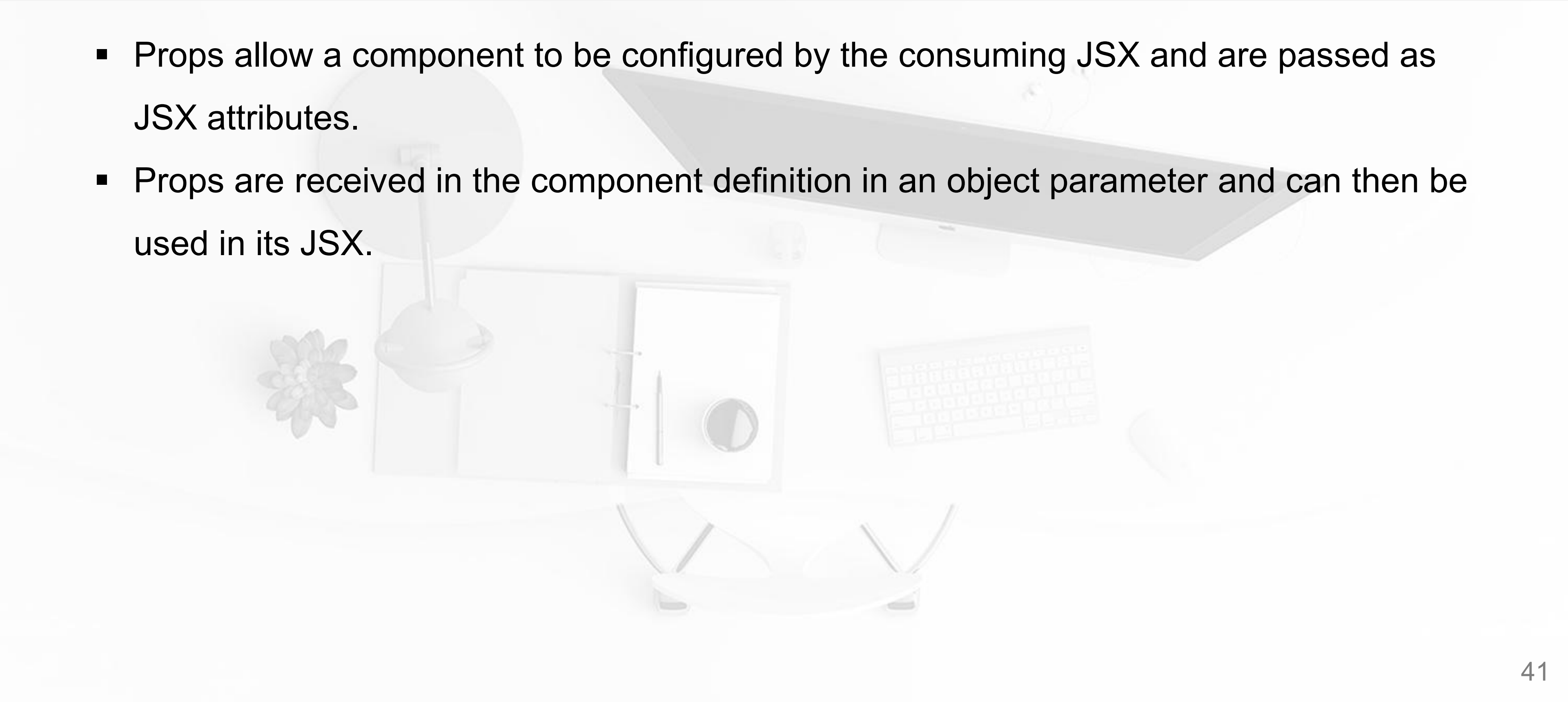
- **Set a default value** to the **type** prop

```
export function Alert({
  type = "information",
  heading,
  children
}) {
  ...
}
```

6. Using props

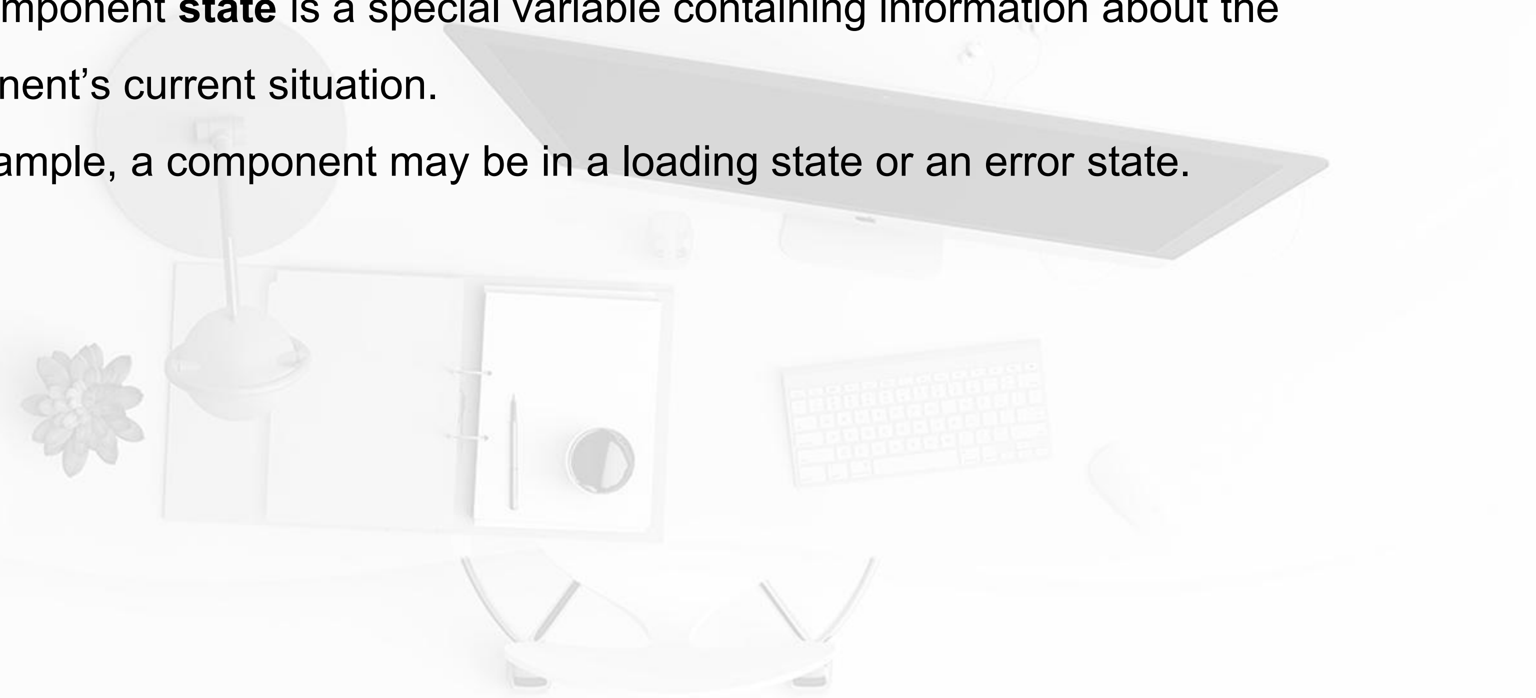
Recap

- Props allow a component to be configured by the consuming JSX and are passed as JSX attributes.
- Props are received in the component definition in an object parameter and can then be used in its JSX.



7. Using state

- The component **state** is a special variable containing information about the component’s current situation.
- For example, a component may be in a loading state or an error state.

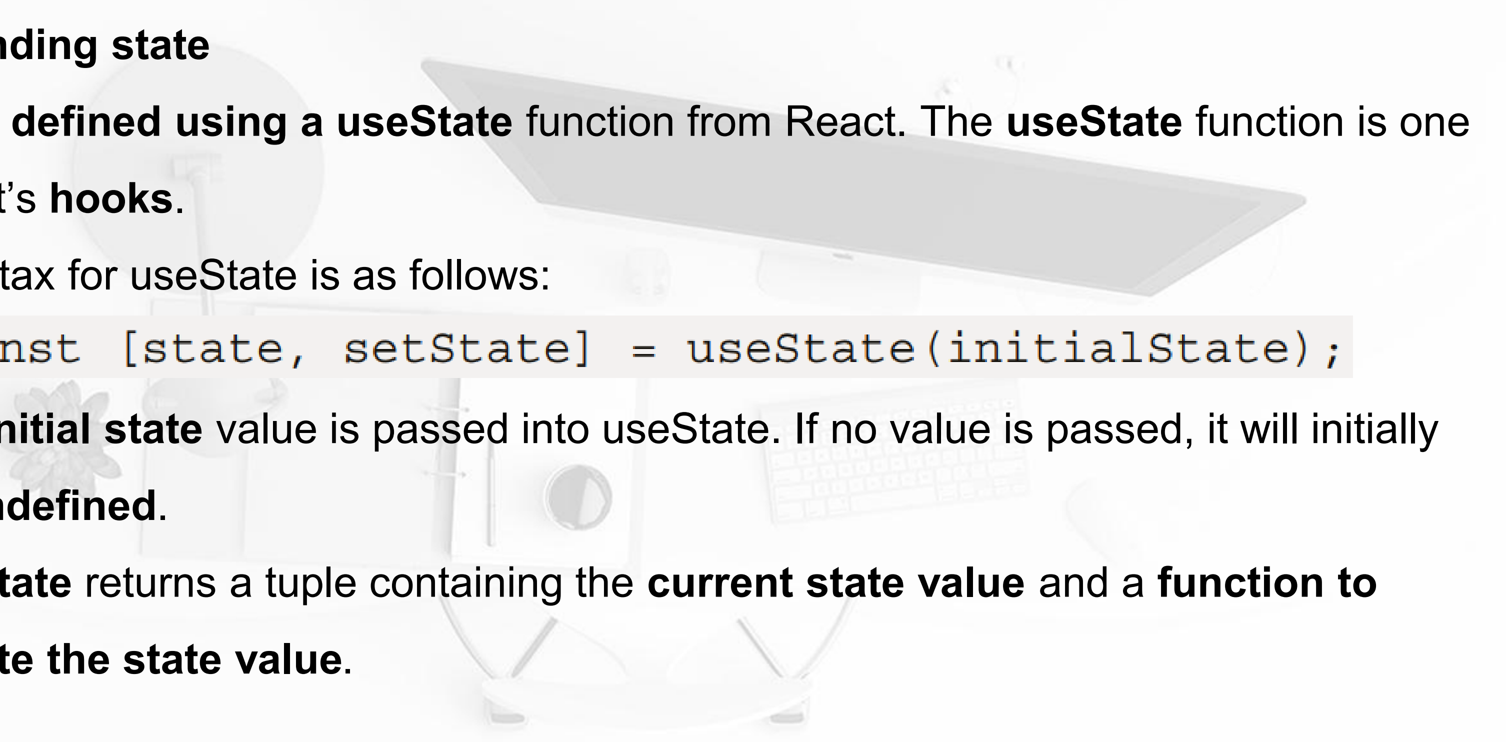


7. Using state

Understanding state

- There isn't a predefined list of states. We define what's appropriate for a given component. Some components won't even need any state.
- **State** is a **key part of making a component interactive**.
  When a user interacts with a component, the component's output may need to change.
  A change to a component state causes the component to refresh, more often referred to as **re-rendering**.

7. Using state



Understanding state

- **State is defined using a useState** function from React. The **useState** function is one of React’s **hooks**.
- The syntax for useState is as follows:

```
const [state, setState] = useState(initialState);
```

  - The **initial state** value is passed into useState. If no value is passed, it will initially be **undefined**.
  - **useState** returns a tuple containing the **current state value** and a **function to update the state value**.

7. Using state

Implementing a visible state in the alert component

- **Requirement**: Implement a feature in the alert component that allows the user to close it.
- Follow these steps:
  - Open **Alert.js**
  - Add the following **import** statement to import the **useState** hook from React

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

  - Define the **visible** state in the component definition:

```
export function Alert(...) {
  const [visible, setVisible] = useState(true);
  return (
    ...
  );
}
```

7. Using state

- Follow these steps:
  - Add a condition that returns ***null*** if the ***visible*** state is ***false***

```
export function Alert(...) {
    const [visible, setVisible] = useState(true);
    if (!visible) {
        return null;
    }
    return (
        ...
    );
}
```

7. Using state

Adding a close button to Alert

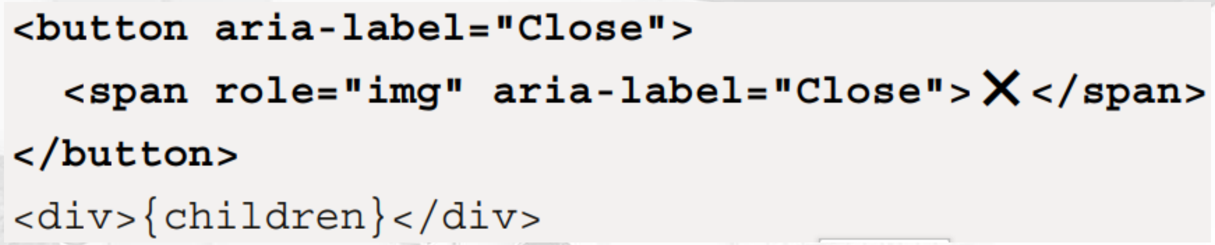
- **Requirement:** Add a **close button** to the alert component to allow the user to close it.
- Follow these steps:
  - Open **Alert.js** and add a **closable** prop

```
export function Alert({
  type = "information",
  heading,
  children,
  closable
}) {
  ...
}
```

7. Using state

Adding a close button to Alert

- Follow these steps:
  - Add a close button

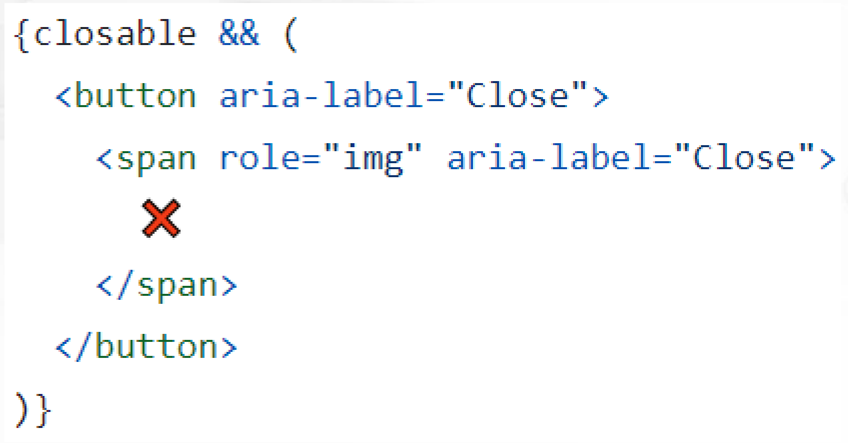


```
<button aria-label="Close">
  <span role="img" aria-label="Close">✕</span>
</button>
<div>{children}</div>
```

7. Using state

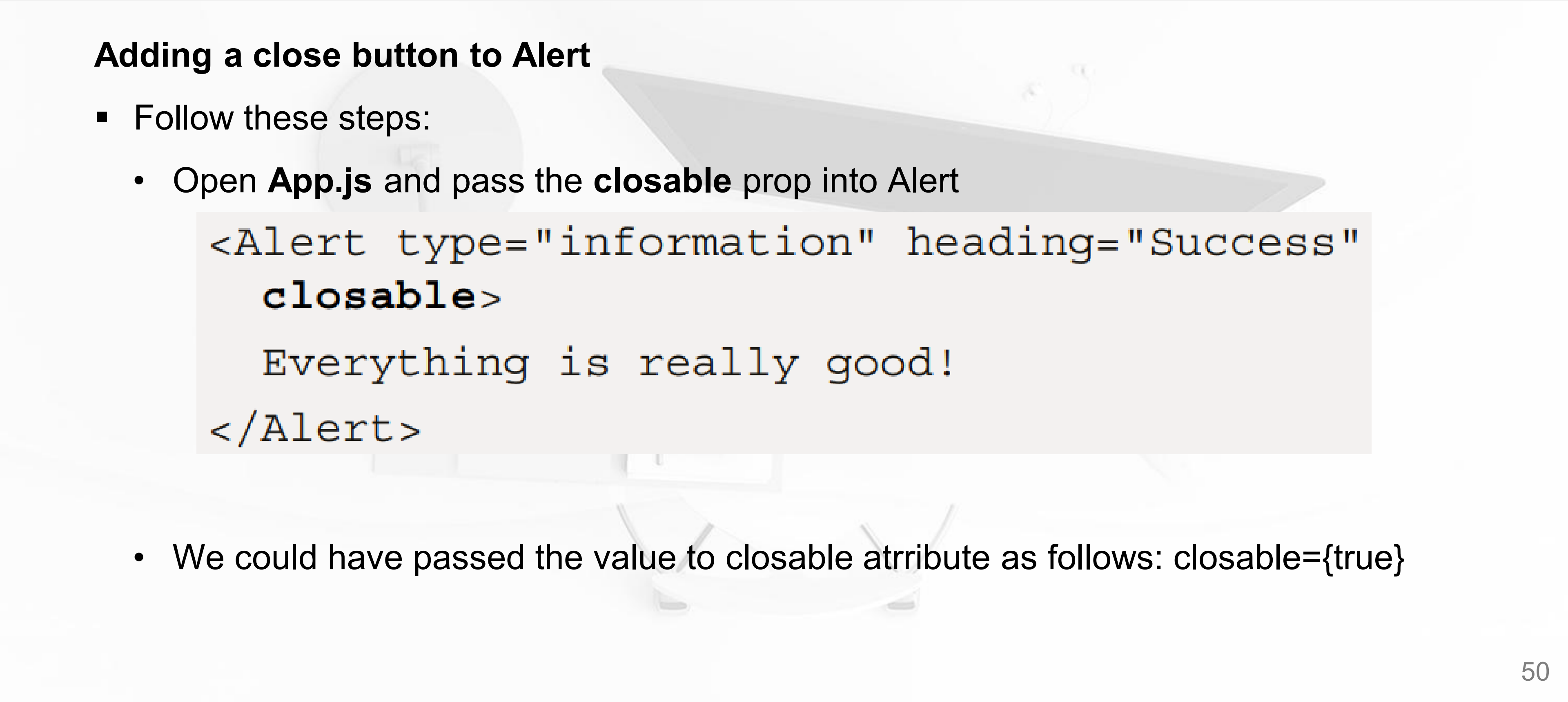
Adding a close button to Alert

- Follow these steps:
  - Use a JavaScript logical AND short circuit expression **(&&)** to render the close button conditionally



```
{closable && (
  <button aria-label="Close">
    <span role="img" aria-label="Close">
      ❌
    </span>
  </button>
)}
```

7. Using state



Adding a close button to Alert

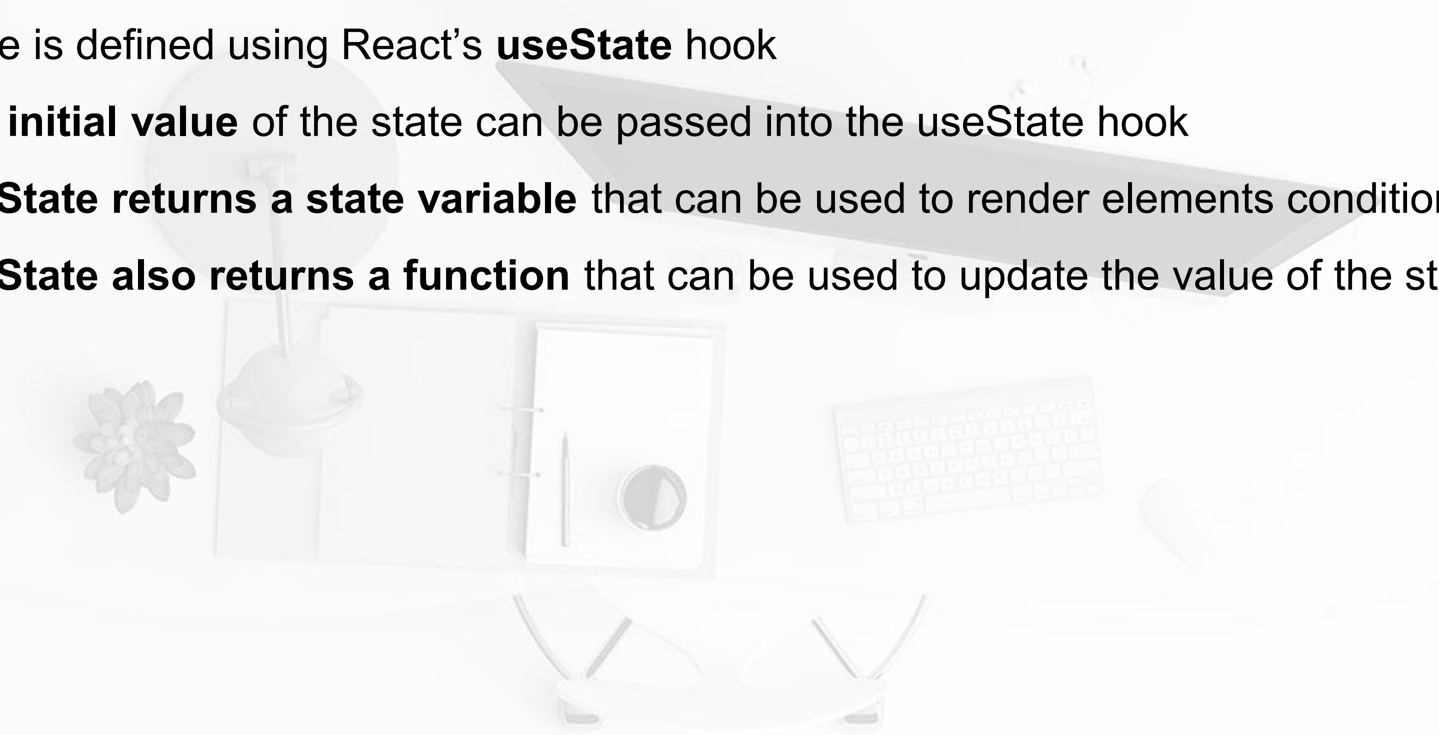
- Follow these steps:
  - Open **App.js** and pass the **closable** prop into Alert

```
<Alert type="information" heading="Success"
  closable>
  Everything is really good!
</Alert>
```

  - We could have passed the value to closable atrribute as follows: closable={true}

7. Using state

Recapitulation



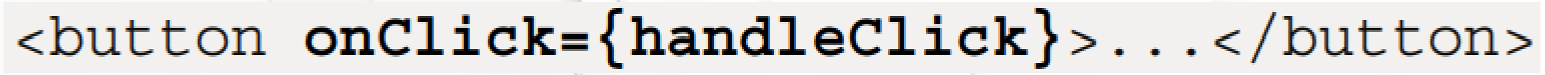
- State is defined using React’s **useState** hook
- The **initial value** of the state can be passed into the useState hook
- **useState returns a state variable** that can be used to render elements conditionally
- **useState also returns a function** that can be used to update the value of the state

8. Using events

Events are another key part of allowing a component to be interactive.

Understanding events

- Browser events happen as the user interacts with DOM elements.
- Logic can be executed when an event is raised.
- A function called an **event handler** (**event listener**) can be registered for an element event that contains the logic to execute when that event happens.
- Event handlers are generally registered to an element in JSX using an attribute



```
<button onClick={handleClick}>...</button>
```

8. Using events

Implementing a close button click handler in the alert

- Open **Alert.js** and register a **click** handler on the close button

```
<button aria-label="Close" onClick={handleCloseClick}>
```

- Implement the **handleCloseClick** function (**above the return** statement) in the component

```
if (!visible) {
  return null;
}
function handleCloseClick() {}
return (
```

8. Using events

Implementing a close button click handler in the alert

- Implement the **handleCloseClick** function in the component
  - Arrow function syntax can be used for event handlers if preferred

```
const handleCloseClick = () => {}
```

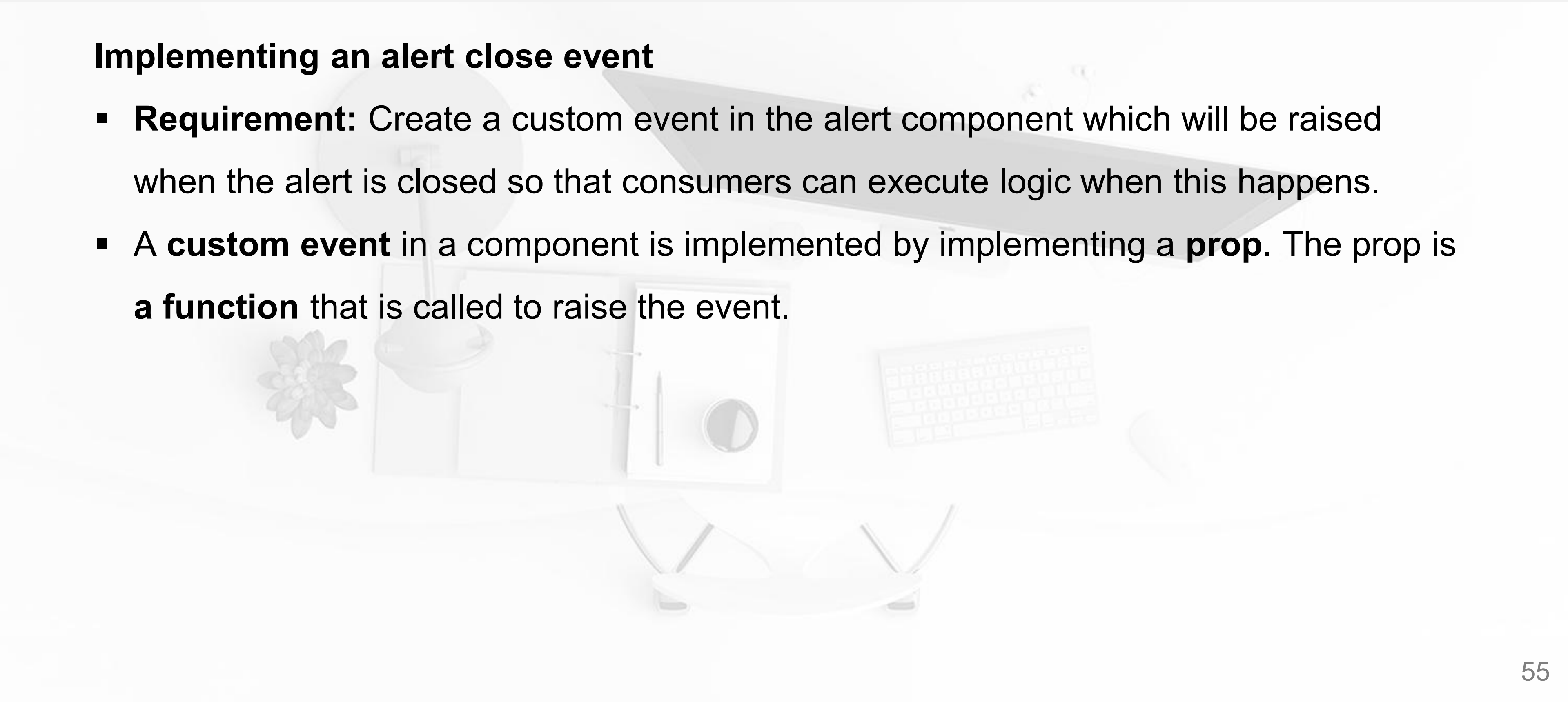
  - Event handlers can also be added directly to the element in JSX

```
<button aria-label="Close" onClick={() => {}}>
```

- Use the **visible** state setter function to make the visible state **false** in the event handler

```
function handleCloseClick() {
  setVisible(false);
}
```

8. Using events



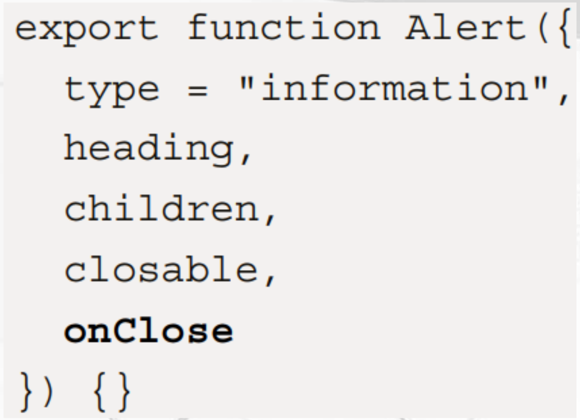
Implementing an alert close event

- **Requirement:** Create a custom event in the alert component which will be raised when the alert is closed so that consumers can execute logic when this happens.
- A **custom event** in a component is implemented by implementing a **prop**. The prop is **a function** that is called to raise the event.

8. Using events

Implementing an alert close event

- Follow these steps:
  - Open **Alert.js** and add a **prop** for the event



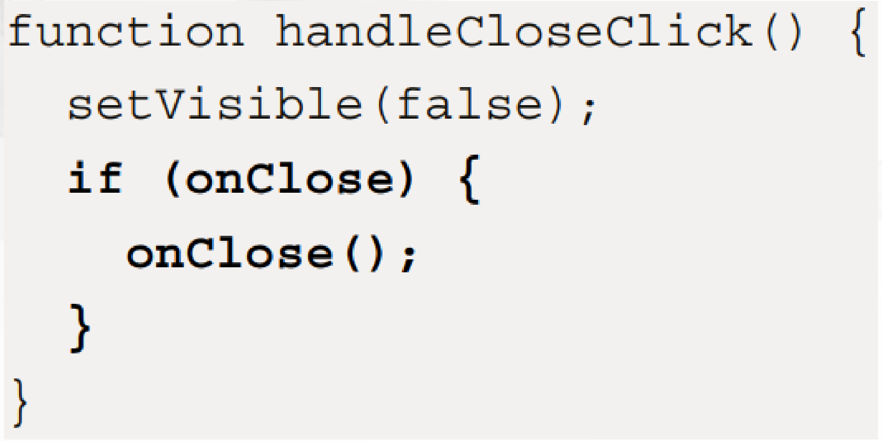
```
export function Alert({
  type = "information",
  heading,
  children,
  closable,
  onClose
}) {}
```

Note: It is common practice to **start an event prop name** with **on**.

8. Using events

Implementing an alert close event

- Follow these steps:
  - In the **handleCloseClick** event handler, **raise the close event** after the **visible** state is set to **false**



```
function handleCloseClick() {
  setVisible(false);
  if (onClose) {
    onClose();
  }
}
```

8. Using events

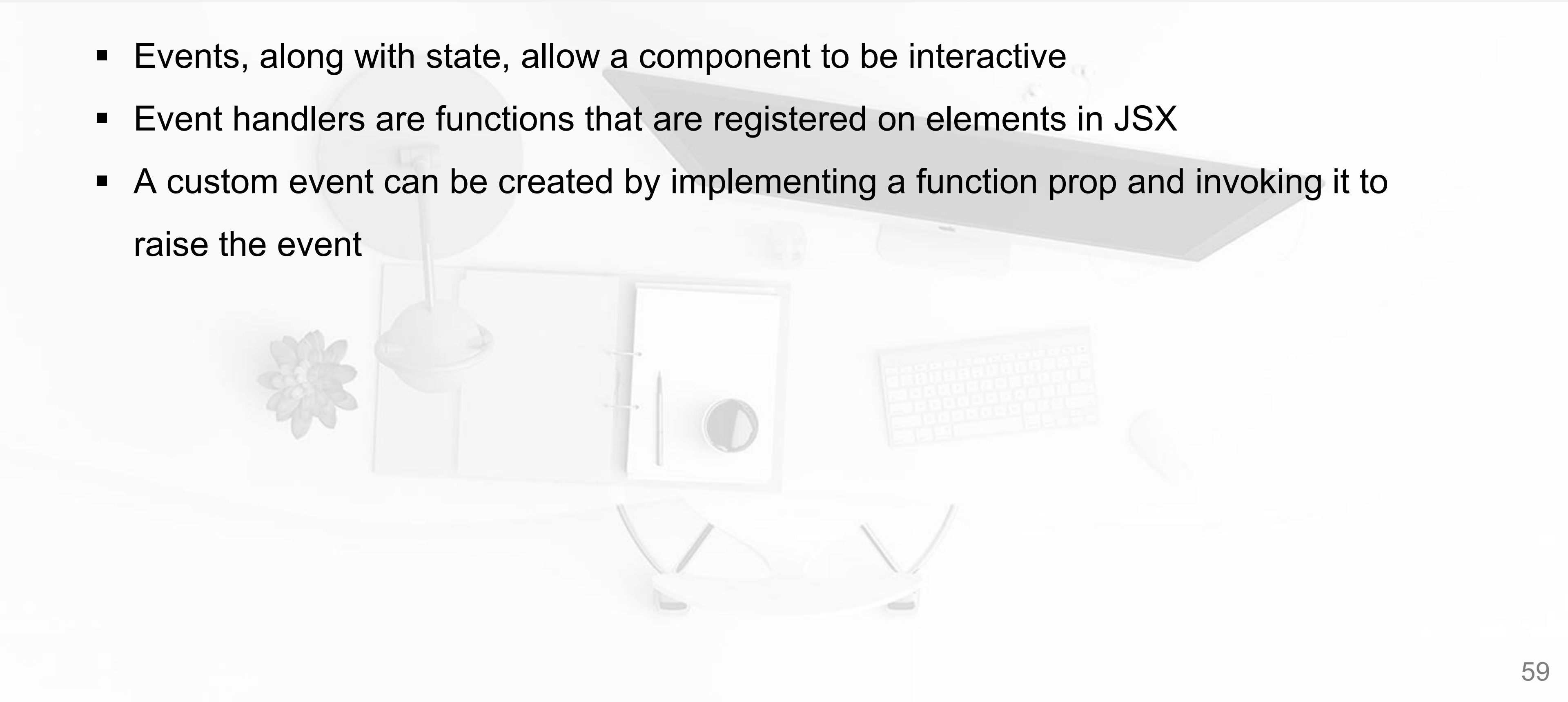
Implementing an alert close event

- Follow these steps:
  - Open **App.js** and add the following event handler to **Alert** in the JSX

```
<Alert
  type="information"
  heading="Success"
  closable
  onClose={() => console.log("closed")}
>
```

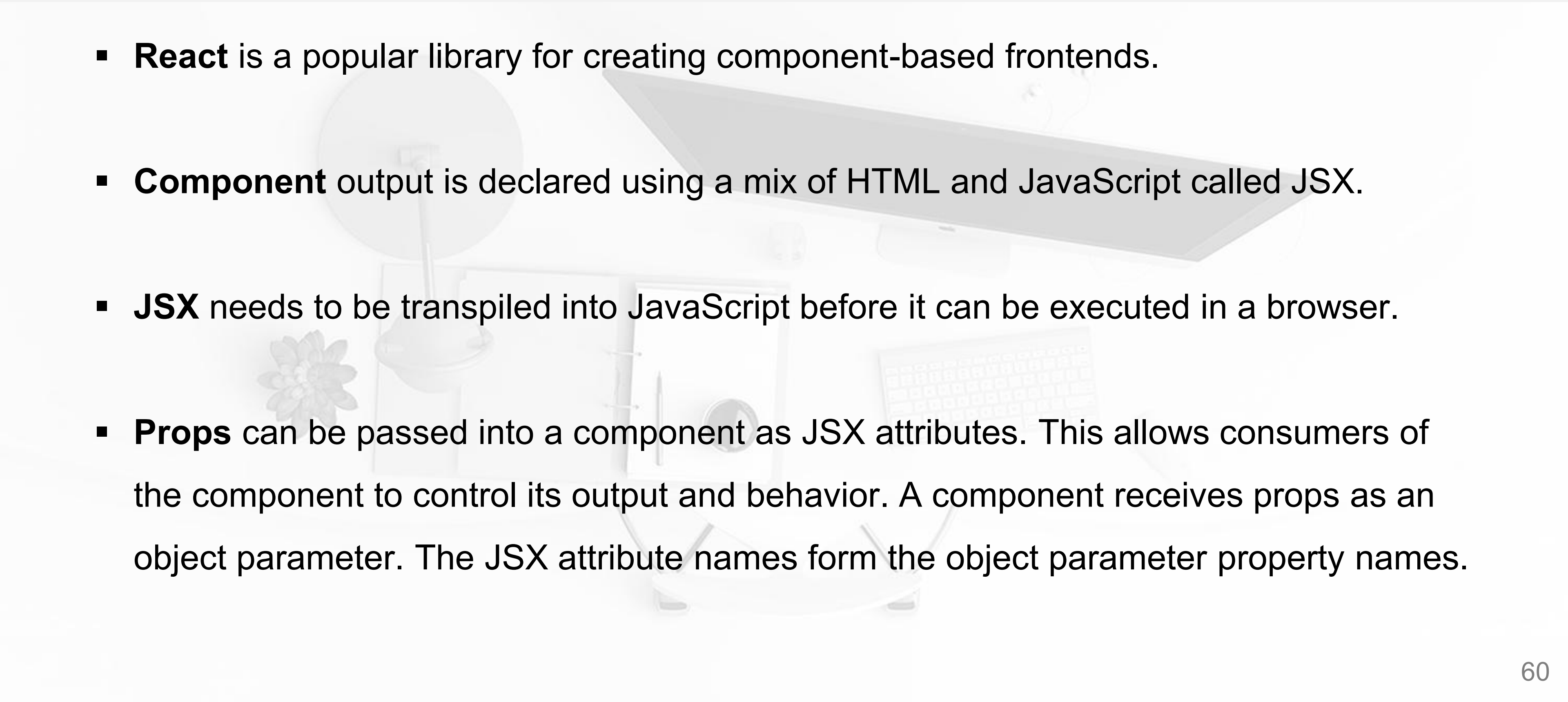
8. Using events

Recap



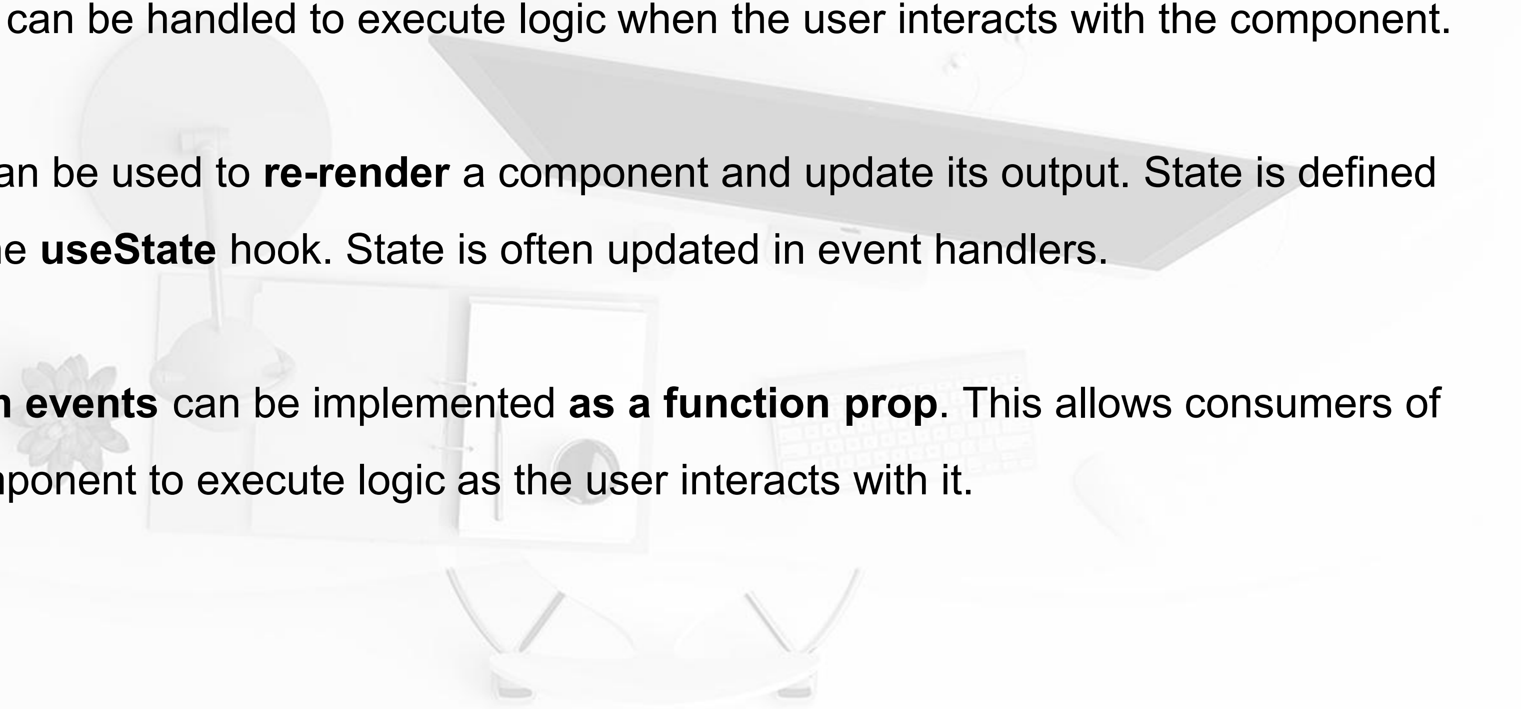
- Events, along with state, allow a component to be interactive
- Event handlers are functions that are registered on elements in JSX
- A custom event can be created by implementing a function prop and invoking it to raise the event

Summary



- **React** is a popular library for creating component-based frontends.
- **Component** output is declared using a mix of HTML and JavaScript called JSX.
- **JSX** needs to be transpiled into JavaScript before it can be executed in a browser.
- **Props** can be passed into a component as JSX attributes. This allows consumers of the component to control its output and behavior. A component receives props as an object parameter. The JSX attribute names form the object parameter property names.

Summary



- **Events** can be handled to execute logic when the user interacts with the component.
- **State** can be used to **re-render** a component and update its output. State is defined using the **useState** hook. State is often updated in event handlers.
- **Custom events** can be implemented **as a function prop**. This allows consumers of the component to execute logic as the user interacts with it.

Q&A