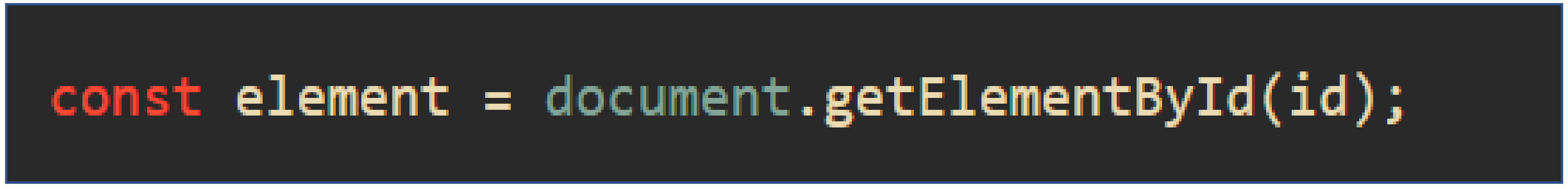
Web Technologies

CAP 756

Presenter: Dr. Shuja Mirza

JavaScript **getElementById()** method

- The **document.getElementById()** method returns an Element object that represents an HTML element with an id that matches a specified string.
- The **document.getElementById()** method returns an Element object that represents an HTML element with an id that matches a specified string.
- If the document has no element with the specified id, the **document.getElementById()** returns null.
- The following shows the syntax of the getElementById() method:



```
const element = document.getElementById(id);
```

- JavaScript **getElementById()** method example:
- Suppose you have the following HTML document:

```
<html>

    <head>

        <title>JavaScript getElementById() Method</title>

    </head>

    <body>

        <p id="message">A paragraph</p>

    </body>

</html>
```

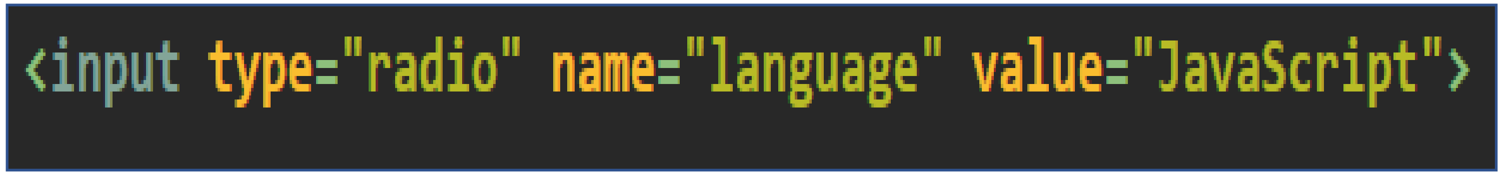
```
const p = document.getElementById('message');
console.log(p);
```

Output

```
<p id="message">A paragraph</p>
```

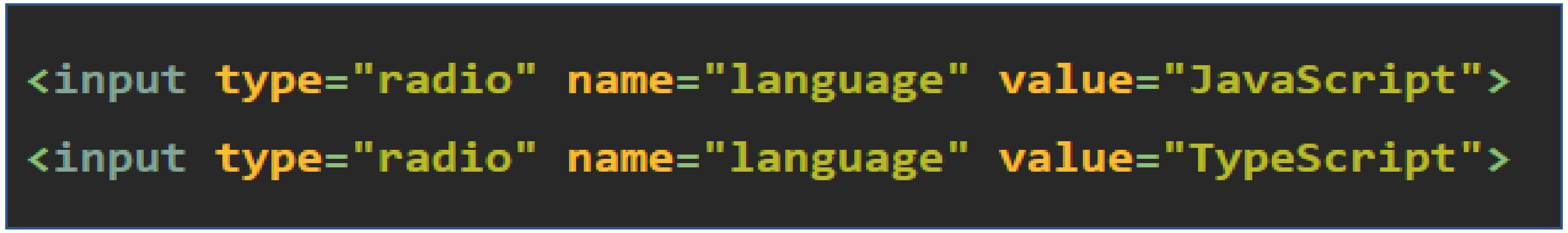
JavaScript getElementsByName() method

- Every element on an HTML document may have a name attribute:



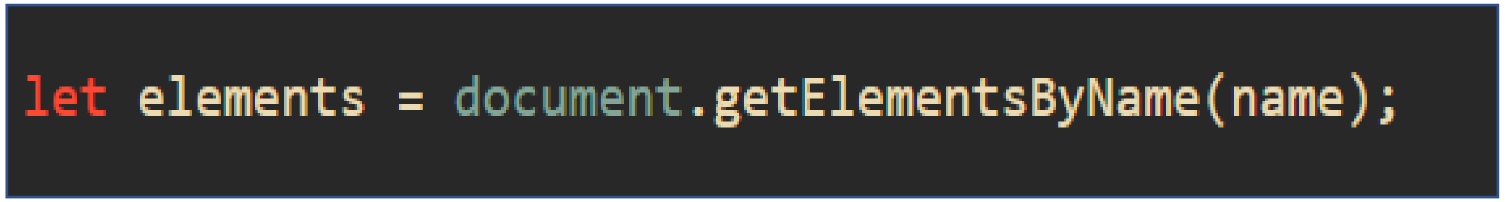
```html
<input type="radio" name="language" value="JavaScript">
```

- Unlike the id attribute, multiple HTML elements can share the same value of the name attribute like this:



```html
<input type="radio" name="language" value="JavaScript">
<input type="radio" name="language" value="TypeScript">
```

- To get all elements with a specified name, you use the **getElementsByName()** method of the document object:

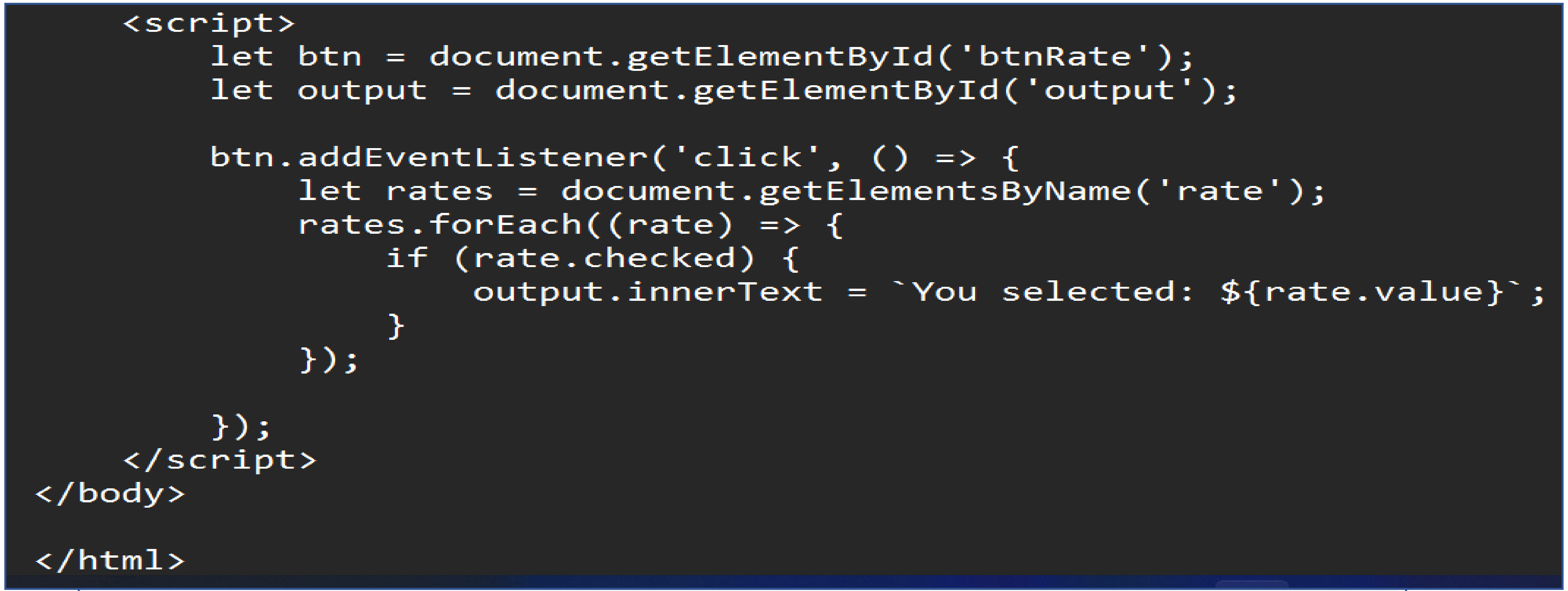


```javascript
let elements = document.getElementsByName(name);
```

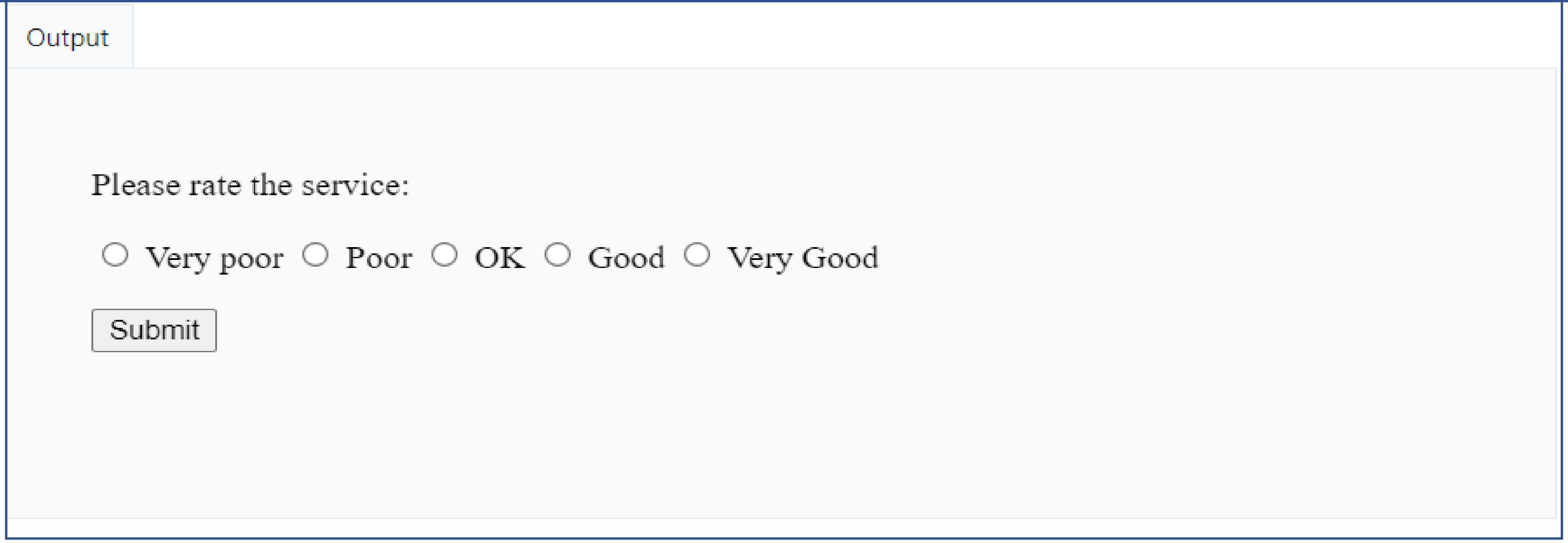
JavaScript getElementsByName() example

- The following example shows a radio group that consists of radio buttons that have the same name (rate).

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8"><title>JavaScript getElementsByName Demo</title>
</head>
<body>
    <p>Please rate the service:</p>
    <p>
        <label for="very-poor">
            <input type="radio" name="rate" value="Very poor" id="very-poor"> Very poor
        </label>
        <label for="poor">
            <input type="radio" name="rate" value="Poor" id="poor"> Poor
        </label>
        <label for="ok">
            <input type="radio" name="rate" value="OK" id="ok"> OK
        </label>
        <label for="good">
            <input type="radio" name="rate" value="Good"> Good
        </label>
        <label for="very-good">
            <input type="radio" name="rate" value="Very Good" id="very-good"> Very Good
        </label>
    </p>
    <p>
        <button id="btnRate">Submit</button>
    </p>
    <p id="output"></p>
```



```
    <script>
        let btn = document.getElementById('btnRate');
        let output = document.getElementById('output');

        btn.addEventListener('click', () => {
            let rates = document.getElementsByName('rate');
            rates.forEach((rate) => {
                if (rate.checked) {
                    output.innerText = `You selected: ${rate.value}`;
                }
            });

        });
    </script>
</body>

</html>
```

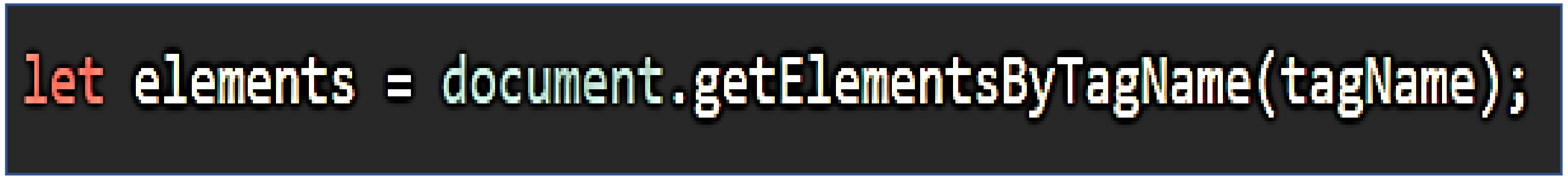


Output

Please rate the service:

○ Very poor ○ Poor ○ OK ○ Good ○ Very Good

- JavaScript **getElementsByTagName**
- The **getElementsByTagName()** is a method of the document object or a specific DOM element.
- The **getElementsByTagName()** method accepts a tag name and returns a live HTMLCollection of elements with the matching tag name in the order which they appear in the document.
- The following illustrates the syntax of the **getElementsByTagName():**



```
let elements = document.getElementsByTagName(tagName);
```

- JavaScript **getElementsByClassName**
- The **getElementsByClassName()** method returns an array-like of objects of the child elements with a specified class name. The **getElementsByClassName()** method is available on the document element or any other elements.
- When calling the method on the document element, it searches the entire document and returns the child elements of the document:

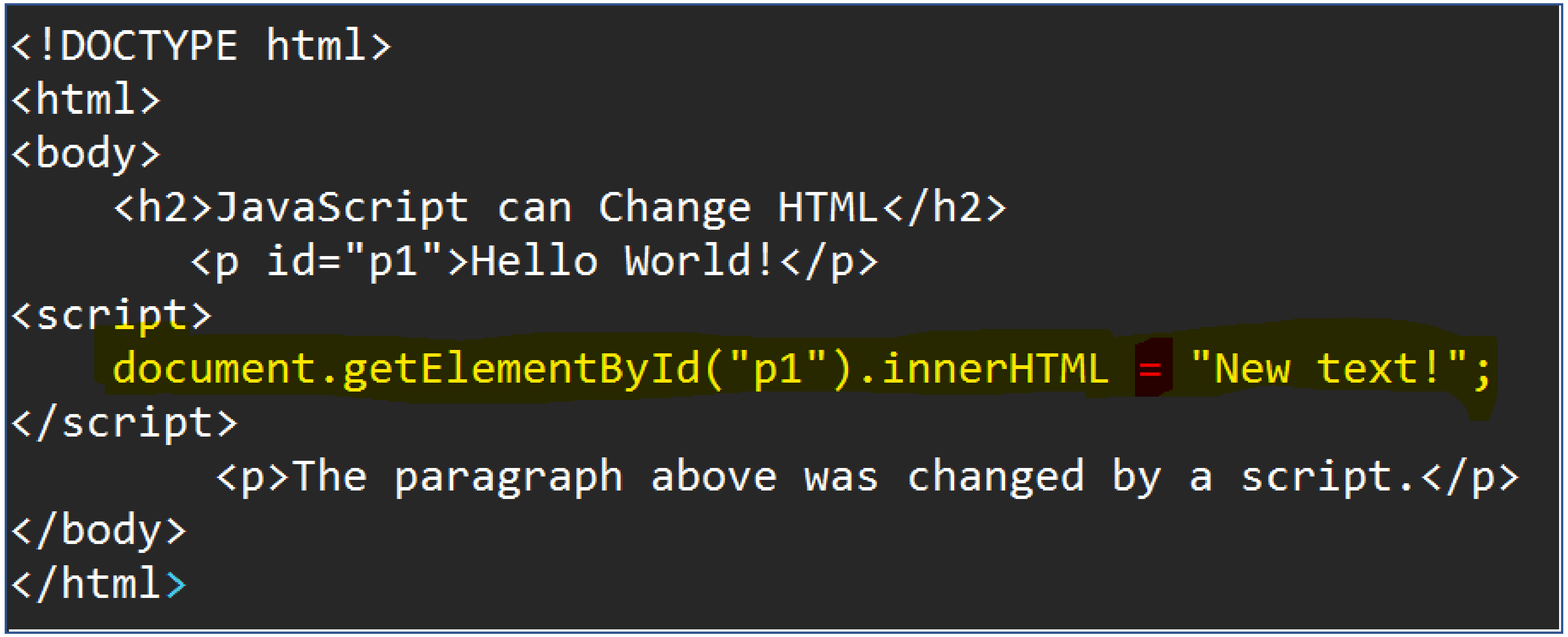
```
let elements = document.getElementsByClassName(names);
```

- However, when calling the method on a specific element, it returns the descendants of that specific element with the given class name:

```
let elements = rootElement.getElementsByClassName(names);
```

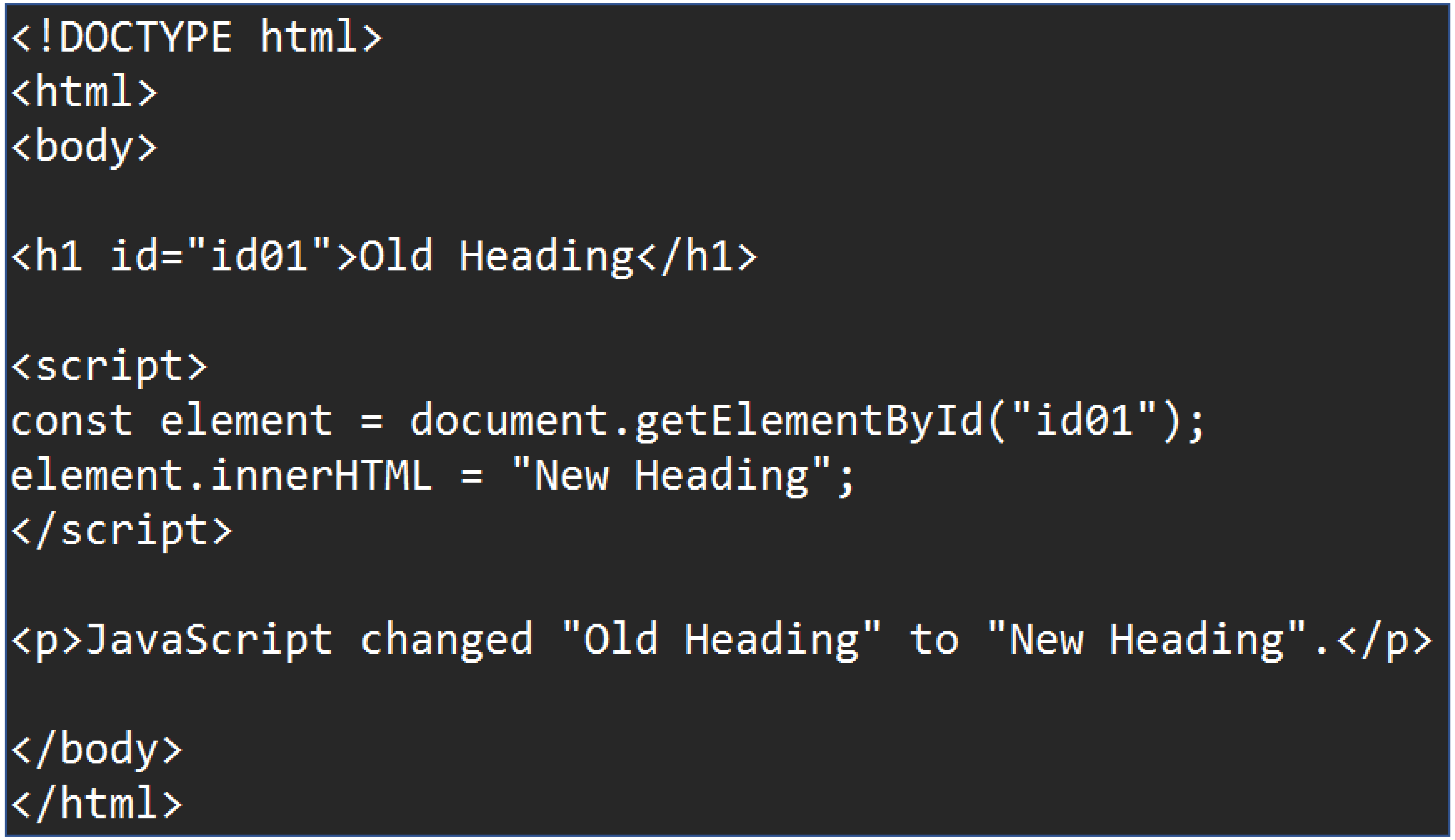
JavaScript HTML DOM - Changing HTML

- The **innerHTML** property can be used to write the dynamic html on the html document.
- It is used mostly in the web pages to generate the dynamic html such as registration form, comment form, links etc.
- To change the content of an HTML element, use this syntax:
- document.getElementById(*id*).innerHTML = *new HTML*



```
<!DOCTYPE html>
<html>
<body>
    <h2>JavaScript can Change HTML</h2>
       <p id="p1">Hello World!</p>
<script>
    document.getElementById("p1").innerHTML = "New text!";
</script>
        <p>The paragraph above was changed by a script.</p>
</body>
</html>
```

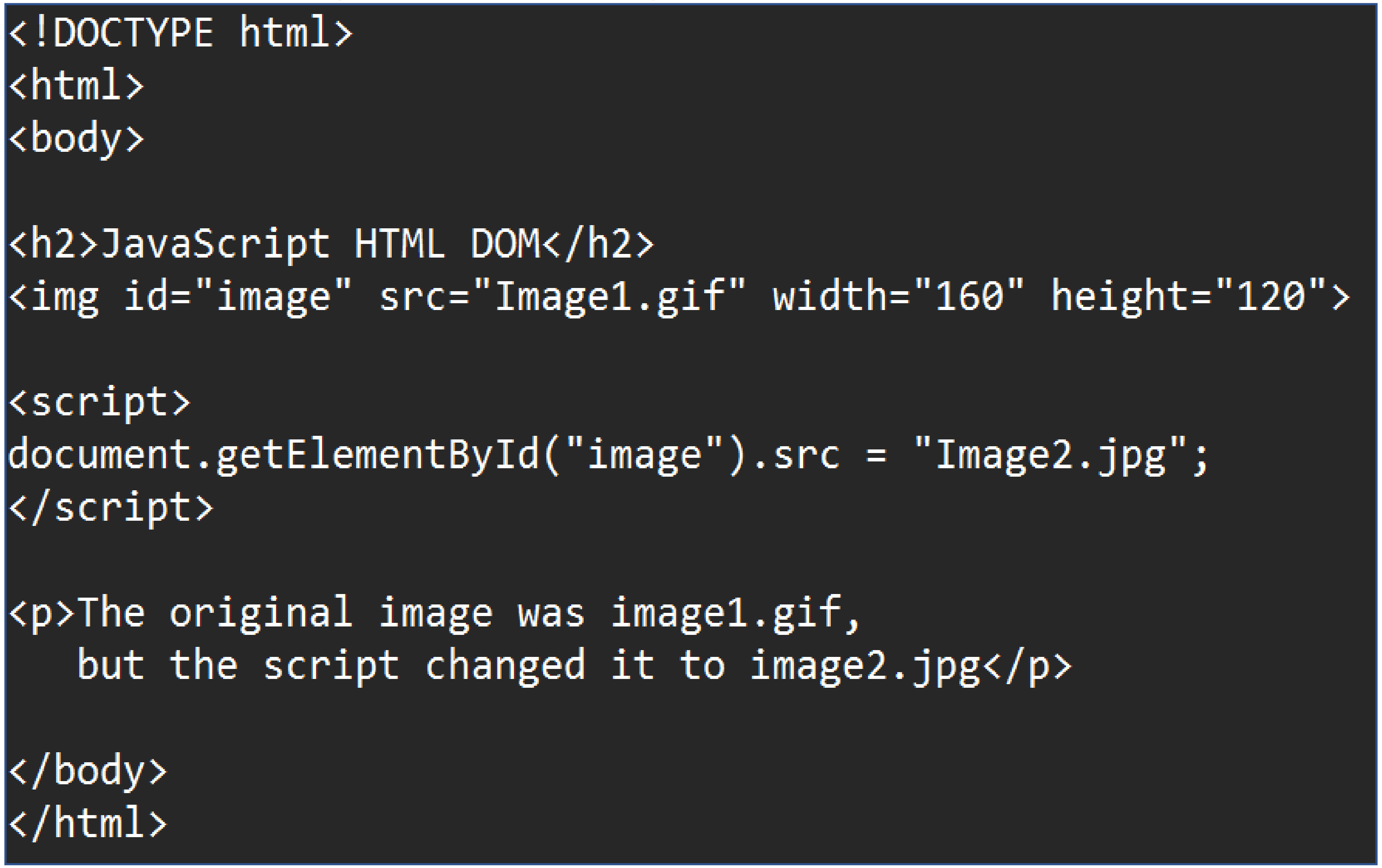
- This example changes the content of an <h1> element:



```html
<!DOCTYPE html>
<html>
<body>

<h1 id="id01">Old Heading</h1>

<script>
const element = document.getElementById("id01");
element.innerHTML = "New Heading";
</script>

<p>JavaScript changed "Old Heading" to "New Heading".</p>

</body>
</html>
```

• Changing the Value of an Attribute

- To change the value of an HTML attribute, use this syntax:
- document.getElementById(*id*).*attribute = new value*



```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>
<img id="image" src="Image1.gif" width="160" height="120">

<script>
document.getElementById("image").src = "Image2.jpg";
</script>

<p>The original image was image1.gif,
   but the script changed it to image2.jpg</p>

</body>
</html>
```

JavaScript Form Validation

- HTML form validation can be done by JavaScript.
- If a form field (fname) is empty, this function alerts a message, and returns false, to prevent the form from being submitted:
- JavaScript Example

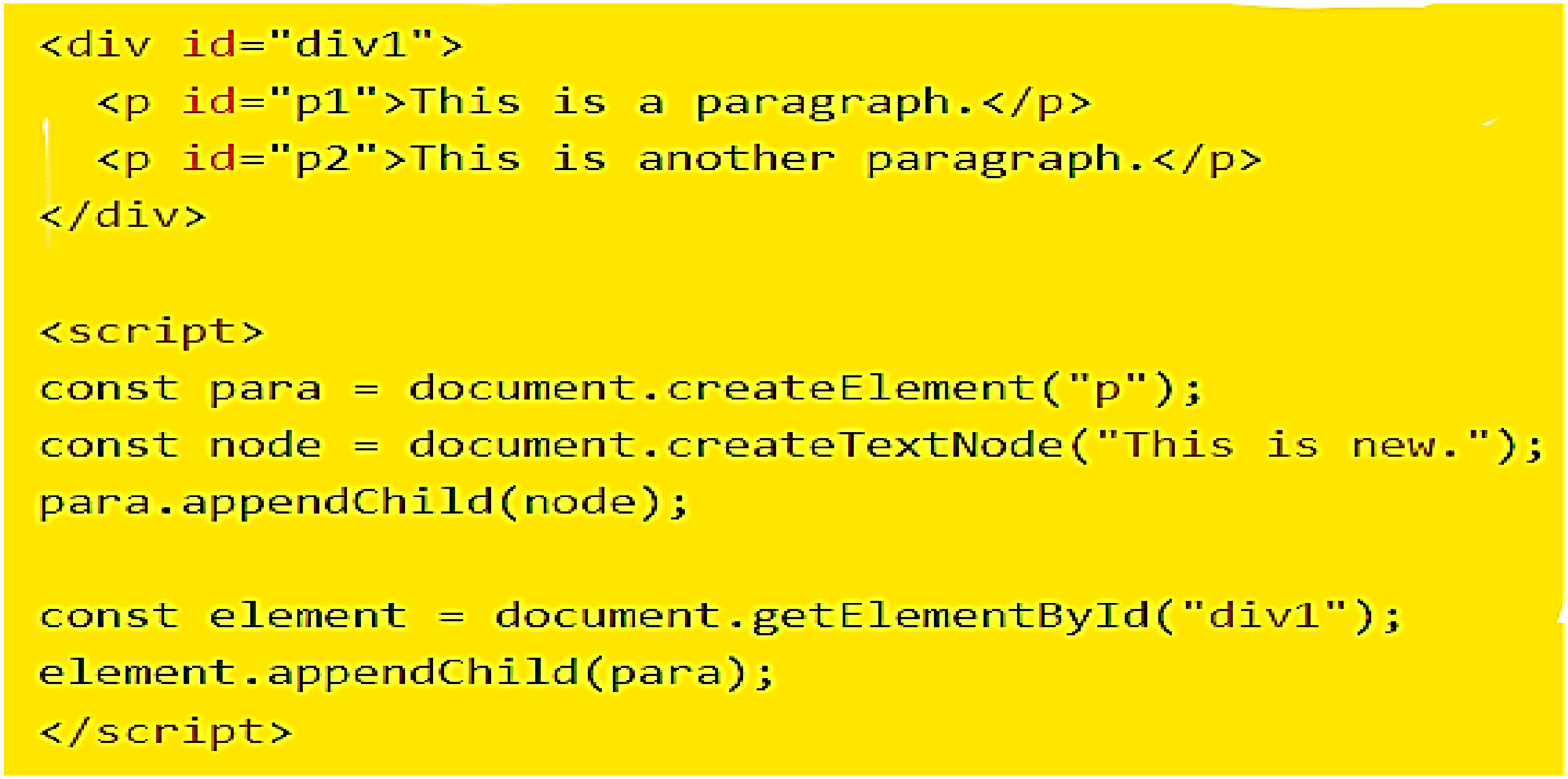
```
function validateForm()
{
  let x = document.forms["myForm"]["fname"].value;
  if (x == "")
{
    alert("Name must be filled out");
    return false;
  }
}
```

- HTML Form Example

```html
<script>
function validateForm() {
  let x = document.forms["myForm"]["fname"].value;
  if (x == "") {
    alert("Name must be filled out");
    return false;
  }
}
</script>
</head>
<body>

<h2>JavaScript Validation</h2>

<form name="myForm" action="/action_page.php" onsubmit="return validateForm()"
method="post">
  Name: <input type="text" name="fname">
  <input type="submit" value="Submit">
</form>

</body>
</html>
```

Adding and Removing Nodes (HTML Elements)

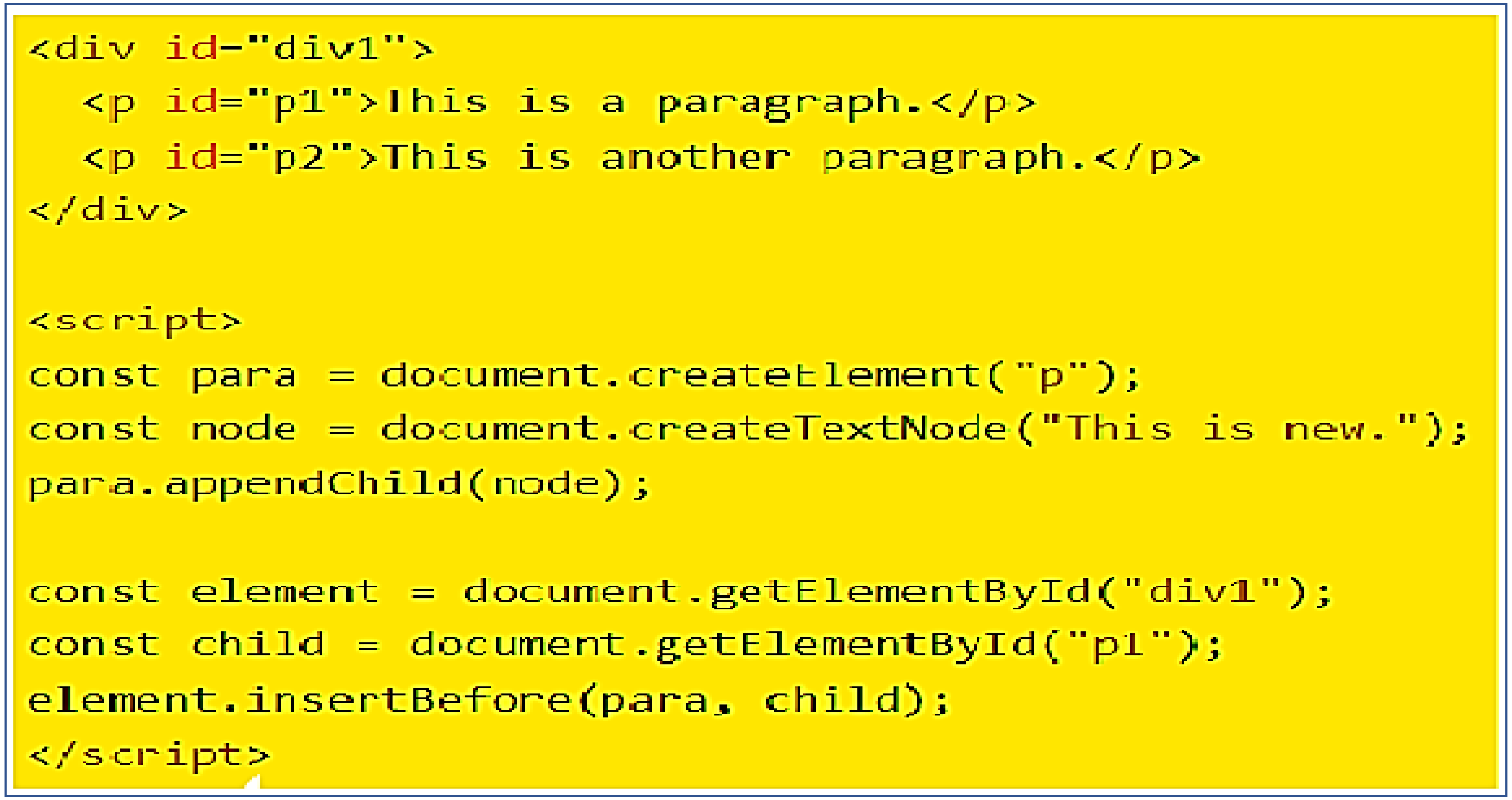
- Creating New HTML Elements (means Nodes)
- To add a new element to the HTML DOM, you must create the element (element node) first, and then append it to an existing element.



```
<div id="div1">
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>

<script>
const para = document.createElement("p");
const node = document.createTextNode("This is new.");
para.appendChild(node);

const element = document.getElementById("div1");
element.appendChild(para);
</script>
```

Creating new HTML Elements - insertBefore()

- The **appendChild()** method in the previous example, appended the new element as the last child of the parent.
- If you don't want that you can use the **insertBefore()** method:



```
<div id="div1">
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>

<script>
const para = document.createElement("p");
const node = document.createTextNode("This is new.");
para.appendChild(node);

const element = document.getElementById("div1");
const child = document.getElementById("p1");
element.insertBefore(para, child);
</script>
```

- Removing Existing HTML Elements
- To remove an HTML element, use the remove() method

-
  ```
  <div>
    <p id="p1">This is a paragraph.</p>
    <p id="p2">This is another paragraph.</p>
  </div>

  <script>
  const elmnt = document.getElementById("p1"); elmnt.remove();
  </script>
  ```

- Replacing HTML Elements
- To replace an element to the HTML DOM, use the replaceChild() method:

-
  ```
<div id="div1">
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>

<script>
const para = document.createElement("p");
const node = document.createTextNode("This is new.");
para.appendChild(node);

const parent = document.getElementById("div1");
const child = document.getElementById("p1");
parent.replaceChild(para, child);
</script>
```

Java Script Events

- JavaScript's interaction with HTML is handled through events that occur when the user or the browser manipulates a page.
- When the page loads, it is called an event. When the user clicks a button, that click too is an event.
- Other examples include events like pressing any key, closing a window, resizing a window, etc.
- Events are a part of the Document Object Model (DOM) and every HTML element contains a set of events which can trigger JavaScript Code.
- Examples of HTML events:

➢When a user clicks the mouse

➢When a web page has loaded

➢When an image has been loaded

➢When the mouse moves over an element

➢When an input field is changed

➢When an HTML form is submitted

➢When a user strokes a key

Assign Events Using the HTML DOM

- The HTML DOM allows you to assign events to HTML elements using JavaScript:

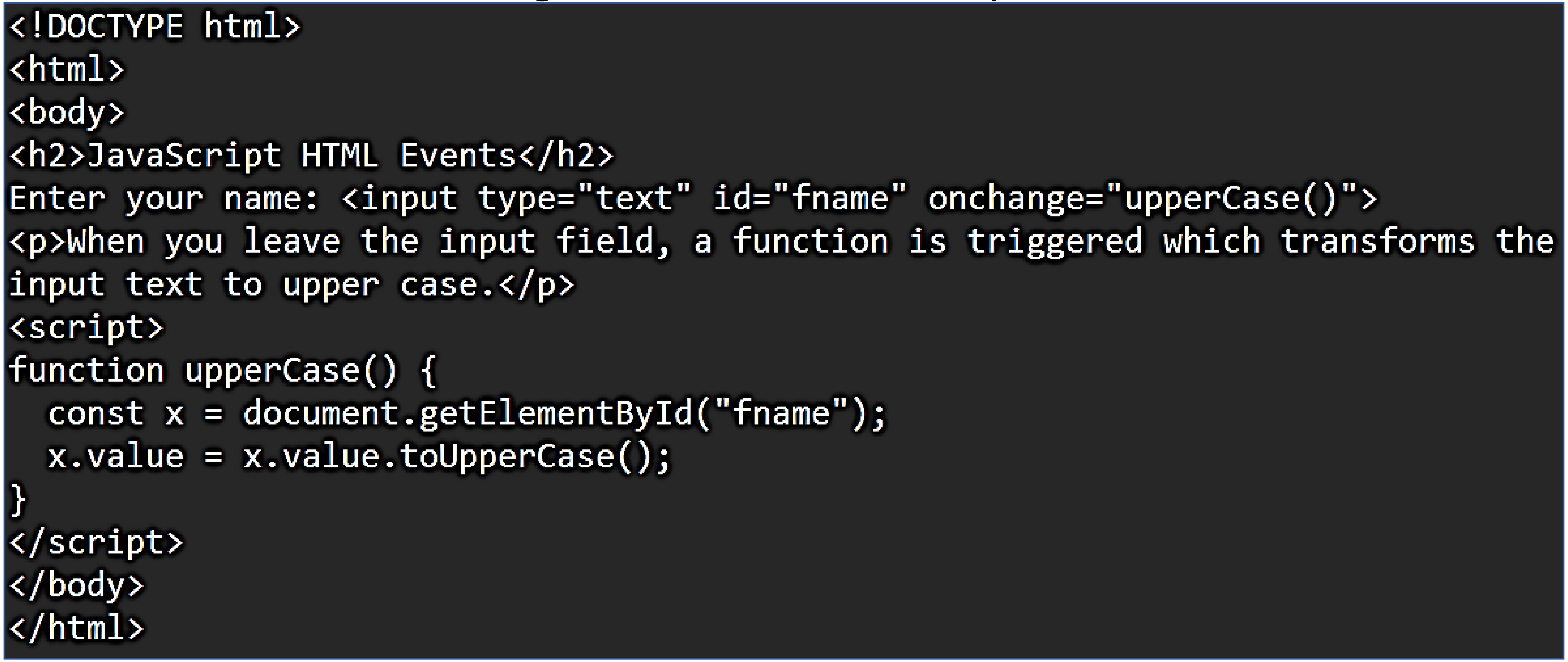
```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript HTML Events</h2>
<p>Click "Try it" to execute the displayDate() function.</p>
<button id="myBtn">Try it</button>
<p id="demo"></p>
<script>
document.getElementById("myBtn").onclick = displayDate;
function displayDate() {
  document.getElementById("demo").innerHTML = Date();
}
</script>
</body>
</html>
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript HTML Events</h2>
        <h2 onclick="changeText(this)">Click on this text!</h2>
<script>
        function changeText(id) {
                id.innerHTML = "Ooops!";
        }
</script>
</body>
</html>
```

- The **onload** and **onunload** Events
- The **onload** and **onunload** events are triggered when the user enters or leaves the page.
- The onload event can be used to check the visitor's browser type and browser version, and load the proper version of the web page based on the information.
- The **onload** and **onunload** events can be used to deal with cookies

```
<!DOCTYPE html>
<html>
<body onload="checkCookies()">
<h2>JavaScript HTML Events</h2>
<p id="demo"></p>
<script>
            function checkCookies() {
            var text = "";
                        if (navigator.cookieEnabled == true) {
                        text = "Cookies are enabled.";
  } else {
                        text = "Cookies are not enabled.";
  }
            document.getElementById("demo").innerHTML = text;
}
</script>
</body>
</html>
```

- The **onchange** Event
- The onchange event is often used in combination with validation of input fields.
- Below is an example of how to use the **onchange**. The **upperCase()** function will be called when a user changes the content of an input field.



```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript HTML Events</h2>
Enter your name: <input type="text" id="fname" onchange="upperCase()">
<p>When you leave the input field, a function is triggered which transforms the
input text to upper case.</p>
<script>
function upperCase() {
  const x = document.getElementById("fname");
  x.value = x.value.toUpperCase();
}
</script>
</body>
</html>
```

The addEventListener() method

a) The addEventListener() method attaches an event handler to the specified element.
b) The addEventListener() method attaches an event handler to an element without overwriting existing event handlers.
c) You can add many event handlers to one element.
d) You can add many event handlers of the same type to one element, i.e two "click" events.
e) You can add event listeners to any DOM object not only HTML elements. i.e the window object.
f) The addEventListener() method makes it easier to control how the event reacts to bubbling.
g) When using the addEventListener() method, the JavaScript is separated from the HTML markup, for better readability and allows you to add event listeners even when you do not control the HTML markup.
h) You can easily remove an event listener by using the removeEventListener() method.

Syntax

- ***element*.addEventListener(*event*, *function*, *useCapture*);**
- The first parameter is the type of the event (like "click" or "mousedown" or any other HTML DOM Event.)
- The second parameter is the function we want to call when the event occurs.
- The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.

The Render Function

- React's goal is in many ways to render HTML in a web page.
- React renders HTML to the web page by using a function called **ReactDOM.render()**
- The **ReactDOM.render()** function takes two arguments, HTML code and an HTML element.
- The purpose of the function is to display the specified HTML code inside the specified HTML element.
- But render where?
- There is another folder in the root directory of your React project, named "**public**". In this folder, there is an **index.html** file.
- You'll notice a single <div> in the body of this file.
- This is where our React application will be rendered.

Example

- import React from 'react';
- import ReactDOM from 'react-dom';
- ReactDOM.render(<p>Hello</p>, document.getElementById('root’));

- The result is displayed in the <div id="root"> element:
- <body>
-   <div id="root"></div>
- </body>

JSX

- **JSX stands for JavaScript XML.**
- **JSX allows us to write HTML in React.**
- **JSX makes it easier to write and add HTML in React.**
- **JSX** is basically a syntax extension of regular **JavaScript** and is used to create **React** elements. These elements are then rendered to the React DOM.
- JSX provides you to write HTML-like structures (e.g., DOM-like tree structures) in the same file where you write JavaScript code, then preprocessor will transform these expressions into actual JavaScript code. Just like HTML, JSX tags have a tag name, attributes, and children.

Coding JSX

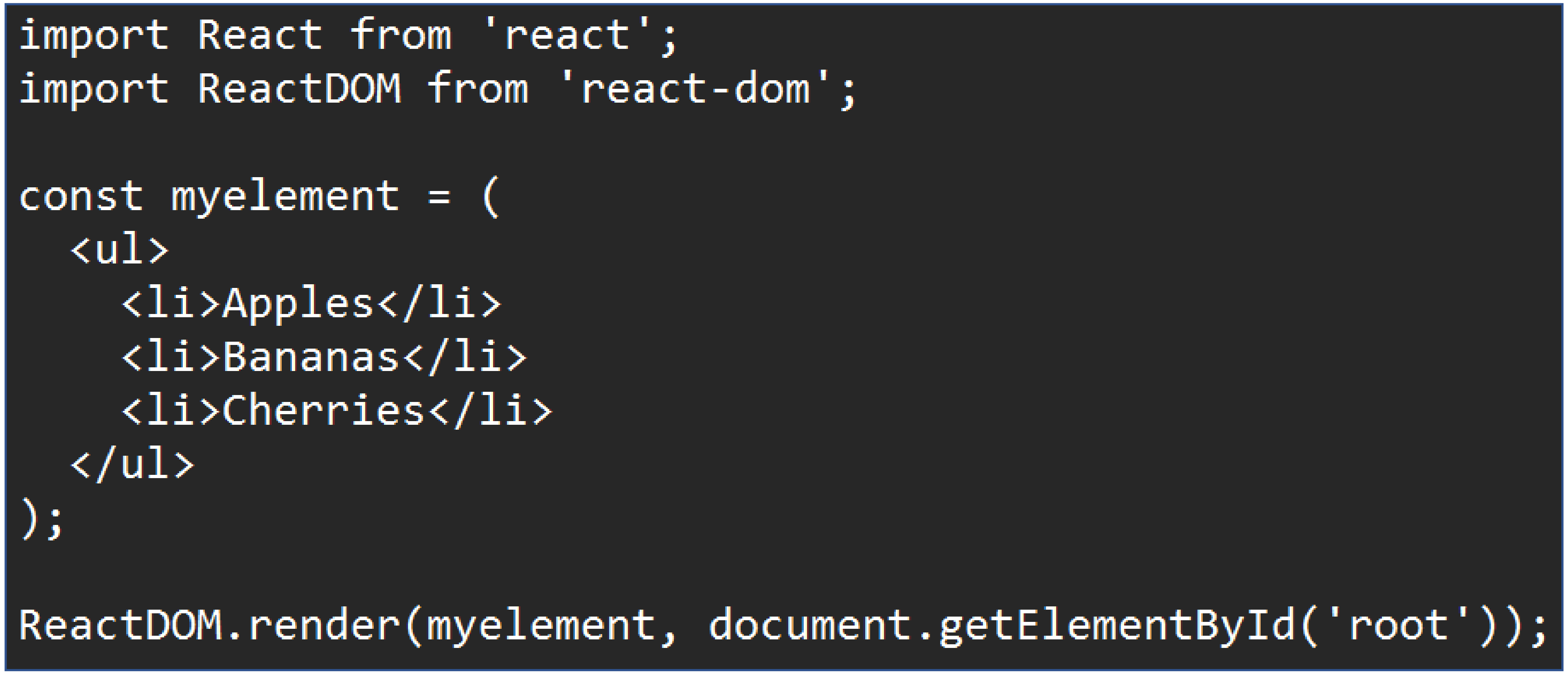
- JSX allows us to write HTML elements in JavaScript and place them in the DOM without any **createElement()** and/or **appendChild()** methods.
- JSX converts HTML tags into react elements.
- Here are two examples. The first uses JSX and the second does not:
- Example 1 with JSX:
- const myelement = <h1>I Love JSX!</h1>;
- ReactDOM.render (myelement, document.getElementById('root’));
- Example 2 Without JSX:
- const myelement = React.createElement('h1', {}, 'I do not use JSX!');
- ReactDOM.render(myelement, document.getElementById('root'));

Expressions in JSX

- With JSX you can write expressions inside curly braces { }.
- The expression can be a React variable, or property, or any other valid JavaScript expression. JSX will execute the expression and return the result:
- Example
- import React from 'react';
- import ReactDOM from 'react-dom';

- const myelement = <h1>React is {5 + 5} times better with JSX</h1>;
- ReactDOM.render(myelement, document.getElementById('root'));

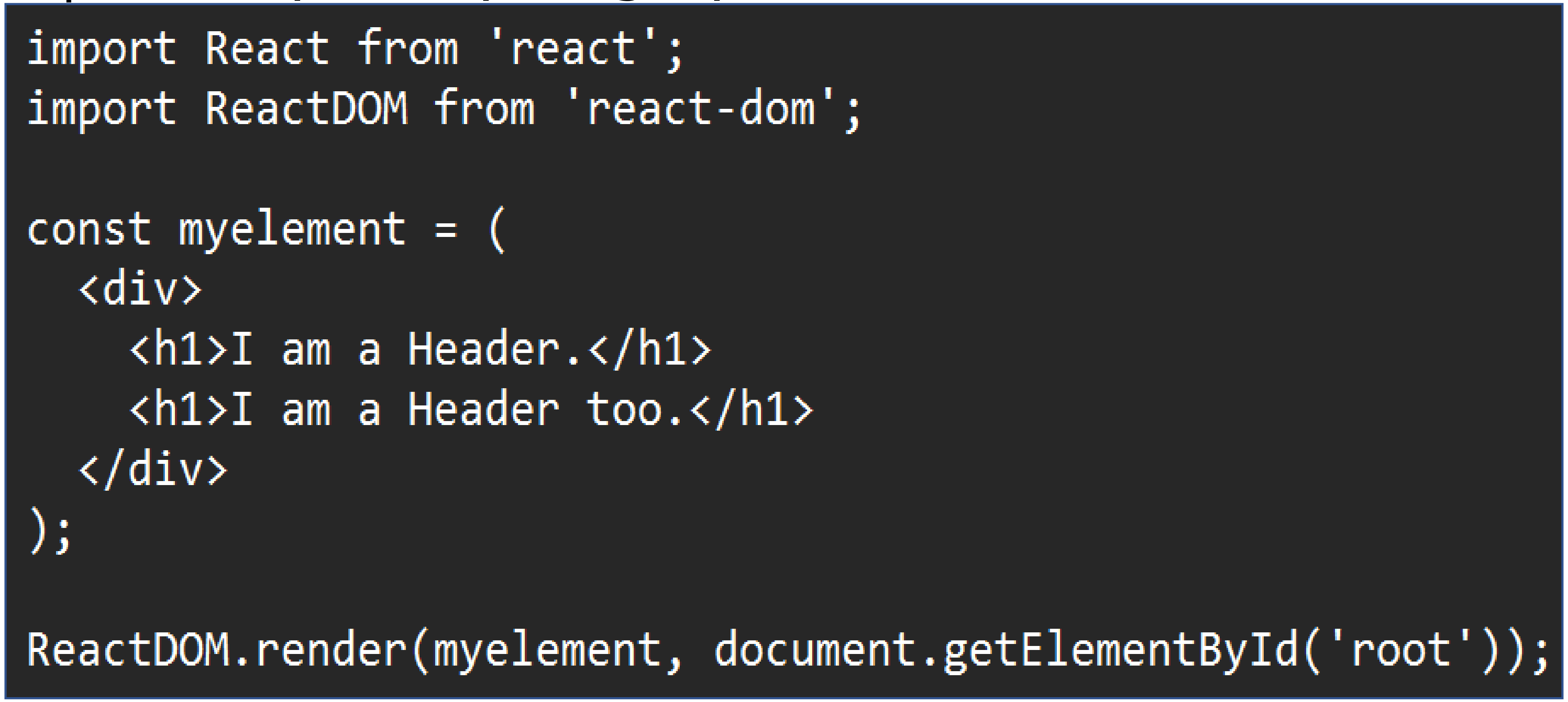
- Inserting a Large Block of HTML
- To write HTML on multiple lines, put the HTML inside parentheses:



```
import React from 'react';
import ReactDOM from 'react-dom';

const myelement = (
  <ul>
    <li>Apples</li>
    <li>Bananas</li>
    <li>Cherries</li>
  </ul>
);

ReactDOM.render(myelement, document.getElementById('root'));
```

One Top Level Element

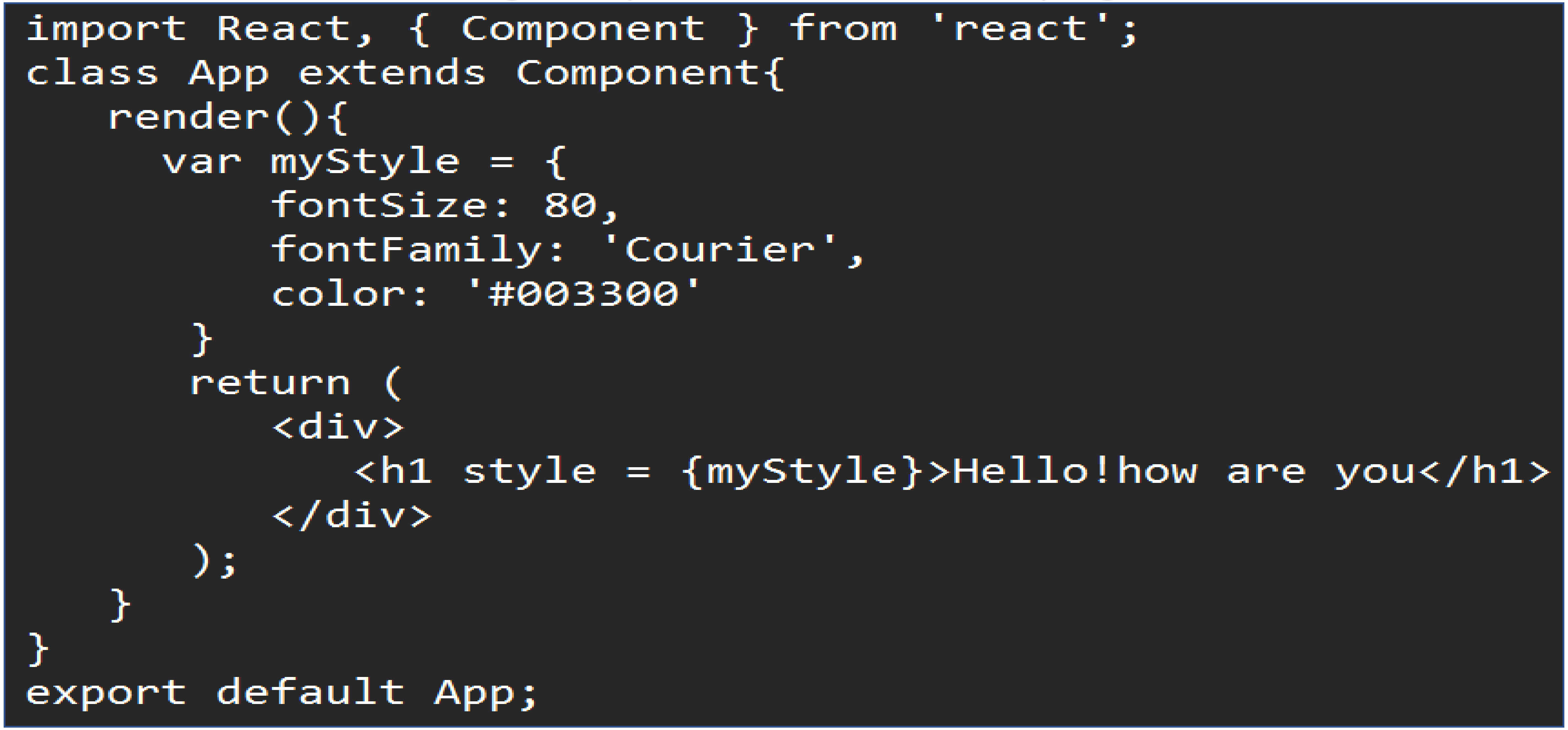
- The HTML code must be wrapped in ONE top level element.
- So if you like to write two paragraphs, you must put them inside a parent element, like a div element.
- Example: Wrap two paragraphs inside one DIV element:



```
import React from 'react';
import ReactDOM from 'react-dom';

const myelement = (
  <div>
    <h1>I am a Header.</h1>
    <h1>I am a Header too.</h1>
  </div>
);

ReactDOM.render(myelement, document.getElementById('root'));
```

JSX Styling:

- React always recommends to use **inline** styles. To set inline styles, you need to use **camelCase** syntax. React automatically allows appending **px** after the number value on specific elements. The following example shows how to use styling in the element.



```
import React, { Component } from 'react';
class App extends Component{
   render(){
     var myStyle = {
         fontSize: 80,
         fontFamily: 'Courier',
         color: '#003300'
      }
      return (
         <div>
            <h1 style = {myStyle}>Hello!how are you</h1>
         </div>
      );
   }
}
export default App;
```

ReactDOM

- ReactJS is a library to build active User Interfaces thus rendering is one of the integral parts of ReactJS.
- React provides the developers with a package **react-dom** alais **ReactDOM** to access and modify the DOM
- **What is DOM?**
- DOM, abbreviated as Document Object Model, is a World Wide Web Consortium standard logical representation of any webpage.
- Before React, Developers directly manipulated the DOM elements which resulted in frequent DOM manipulation, and each time an update was made the browser had to recalculate and repaint the whole view according to the particular CSS of the page, which made the total process to consume a lot of time.

What is ReactDOM?

- **ReactDOM** is a package that provides DOM specific methods that can be used at the top level of a web app to enable an efficient way of managing DOM elements of the web page. **ReactDOM** provides the developers with an API containing the following methods and a few more.

✓ render()

✓ findDOMNode()

✓ unmountComponentAtNode()

✓ hydrate()

✓ createPortal()

- **Pre-requisite:** To use the **ReactDOM** in any React web app we must first import **ReactDOM** from the **react-dom** package by using the following code snippet:
- *import ReactDOM from 'react-dom'*

render() Function

- This is one of the most important methods of **ReactDOM**. This function is used to render a single React Component or several Components wrapped together in a Component or a div element.
- **Syntax**: ReactDOM.render(element, container, callback)
- **Parameters**: This method can take a maximum of three parameters as described below.
- **element:** This parameter expects a JSX expression or a React Element to be rendered.
- **container:** This parameter expects the container in which the element has to be rendered.
- **callback:** This is an optional parameter that expects a function that is to be executed once the render is complete.
- **Return Type:** This function returns a reference to the component or null if a stateless component was rendered.

findDOMNode() Function

- This function is generally used to get the DOM node where a particular React component was rendered. This method is very less used like the following can be done by adding a ref attribute to each component itself.
- **Syntax**: ReactDOM.findDOMNode(component)
- **Parameters**: This method takes a single parameter component that expects a React Component to be searched in the Browser DOM.
- **Return Type:** This function returns the DOM node where the component was rendered on success otherwise null.

unmountComponentAtNode() Function

- This function is used to unmount or remove the React Component that was rendered to a particular container.
- As an example, you may think of a notification component, after a brief amount of time it is better to remove the component making the web page more efficient.
- **Syntax**: ReactDOM.unmountComponentAtNode(container)
- **Parameters**: This method takes a single parameter container which expects the DOM container from which the React component has to be removed.
- **Return Type:** This function returns true on success otherwise false.

hydrate() Function

- This method is equivalent to the render() method but is implemented while using server-side rendering.
- **Syntax**: ReactDOM.hydrate(element, container, callback)
- **Parameters**: This method can take a maximum of three parameters as described below.
- **element:** This parameter expects a JSX expression or a React Component to be rendered.
- **container:** This parameter expects the container in which the element has to be rendered.
- **callback:** This is an optional parameter that expects a function that is to be executed once the render is complete.
- **Return Type:** This function attempts to attach event listeners to the existing markup and returns a reference to the component or null if a stateless component was rendered.

createPortal() Function

- Usually, when an element is returned from a component’s render method, it’s mounted on the DOM as a child of the nearest parent node which in some cases may not be desired. Portals allow us to render a component into a DOM node that resides outside the current DOM hierarchy of the parent component.
- **Syntax**: ReactDOM.createPortal(child, container)
- **Parameters**: This method takes two parameters as described below.
- **child:** This parameter expects a JSX expression or a React Component to be rendered.
- **container:** This parameter expects the container in which the element has to be rendered.
- **Return Type:** This function returns nothing.

Virtual DOM

- React uses Virtual DOM exists which is like a lightweight copy of the actual DOM(a virtual representation of the DOM).
- So for every object that exists in the original DOM, there is an object for that in React Virtual DOM. It is exactly the same, but it does not have the power to directly change the layout of the document.
- Manipulating DOM is slow, but manipulating Virtual DOM is fast as nothing gets drawn on the screen.
- So each time there is a change in the state of our application, virtual DOM gets updated first instead of the real DOM

How Virtual DOM helps React

- In react, everything is treated as a component be it a functional component or class component. A component can contain a state.
- Each time we change something in our JSX file or let's put it in simple terms, whenever the state of any component is changed react updates it's Virtual DOM tree.
- React maintains two Virtual DOM at each time, one contains the updated Virtual DOM and one which is just the pre-update version of this updated Virtual DOM.
- Now it compares the pre-update version with the updated Virtual DOM and figures out what exactly has changed in the DOM like which components have been changed.
- This process of comparing the current Virtual DOM tree with the previous one is known as **'diffing'**.
- This entire process of transforming changes to the real DOM is called **Reconciliation**