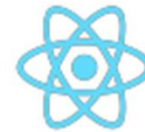


# 4

## Virtual DOM Architecture

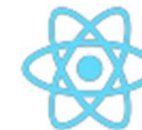


# Virtual DOM



## React - Characteristics

- React apps are built with the latest version of ES.
- React makes use of reusable components.
  - A component is a function/class that returns a section of the interface.
- React is declarative
  - React allows to describe what the application interface should look like
- Unidirectional data flow
  - React applications are built as a combination of parent and child components.
    - Data always flows from parent to child and never in the other direction.
- Powerful type-checking using PropTypes



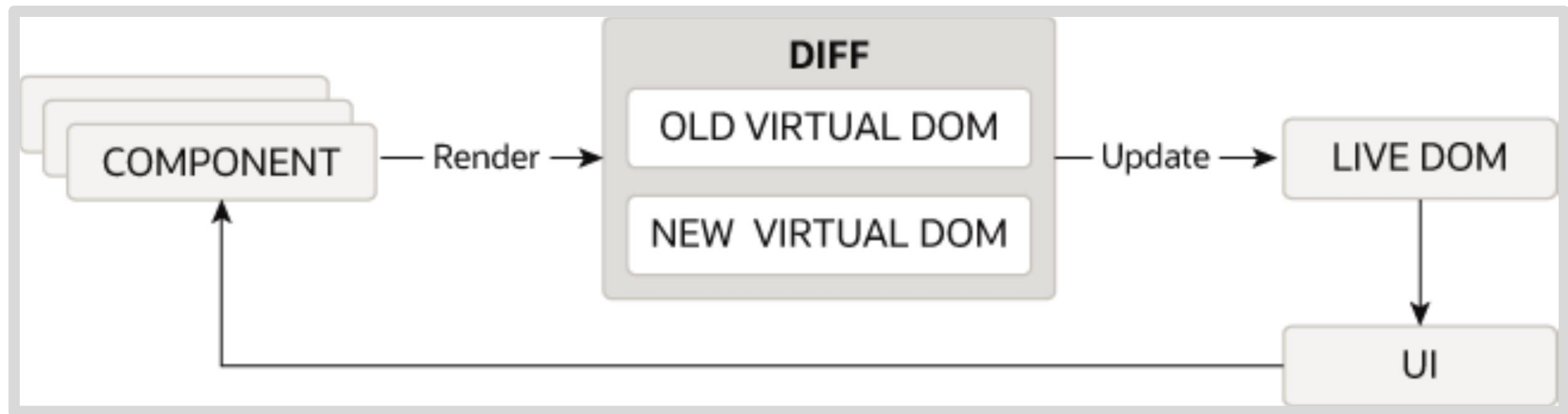
## React – The Virtual DOM

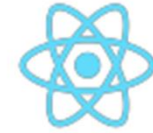
- DOM manipulation is the heart of most modern and interactive web applications.
- DOM manipulation is slow, moreover many JavaScript libraries and frameworks update the DOM more than needed.
- In React for every DOM object there is a corresponding Virtual DOM object.
- A Virtual DOM object is an in-memory representation of the real DOM object.
  - It has the same properties as the real DOM object.
- It lacks the power to directly change what is on the screen.

## About Virtual DOM Architecture

- Virtual DOM architecture is a different way of building apps and web components from the Model-View-ViewModel (MVVM) architecture
- Virtual DOM architecture is a programming pattern where a virtual representation of the DOM is kept in memory and synchronized with the live DOM by a JavaScript library.
- As a pattern, it has gained popularity for its ability to efficiently update the browser's DOM (the live DOM).
- React is the JavaScript library uses to synchronize changes in the virtual DOM to the live DOM.

# VDOM

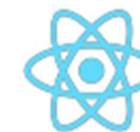




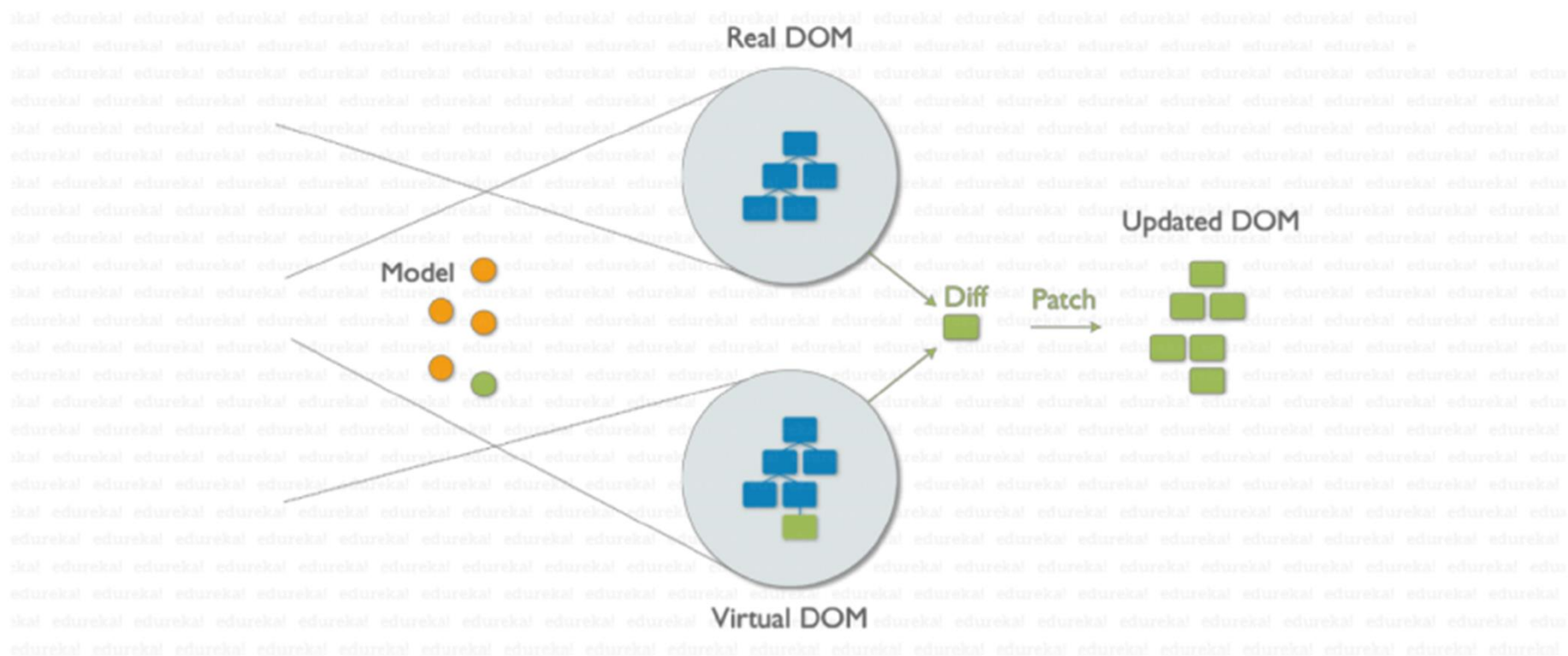
## Virtual DOM...

- Manipulating the DOM is slow whereas manipulating the virtual DOM is fast as nothing gets drawn on the screen.
- Once the virtual DOM is updated React compares it with its previous state.
- “**Diffing**” is the process by which React figures the virtual DOM objects that have changed.
- React updates only the changed objects in the real DOM.
- Changes on the real DOM cause the screen to change.

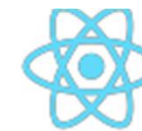




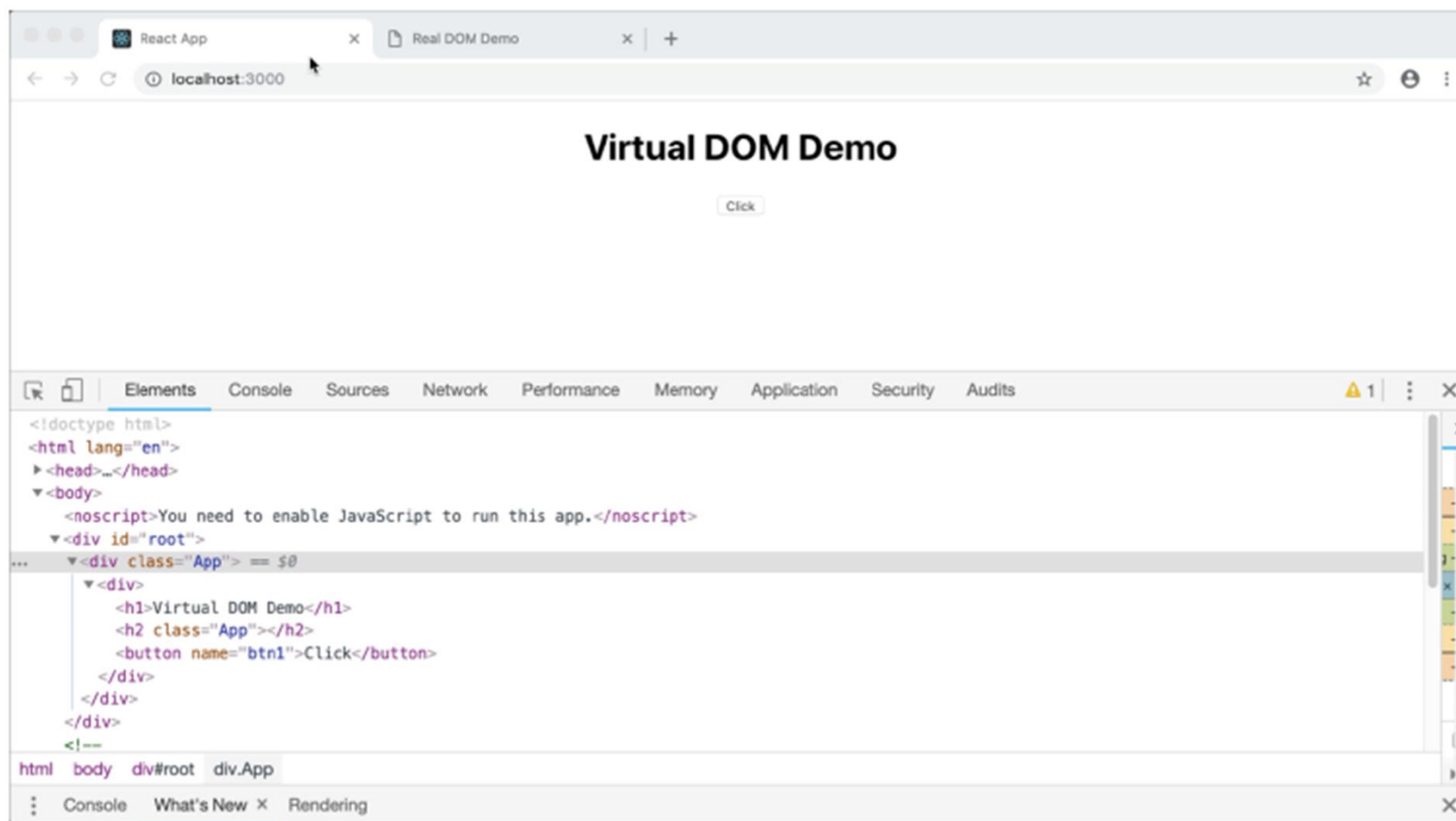
# The Virtual DOM







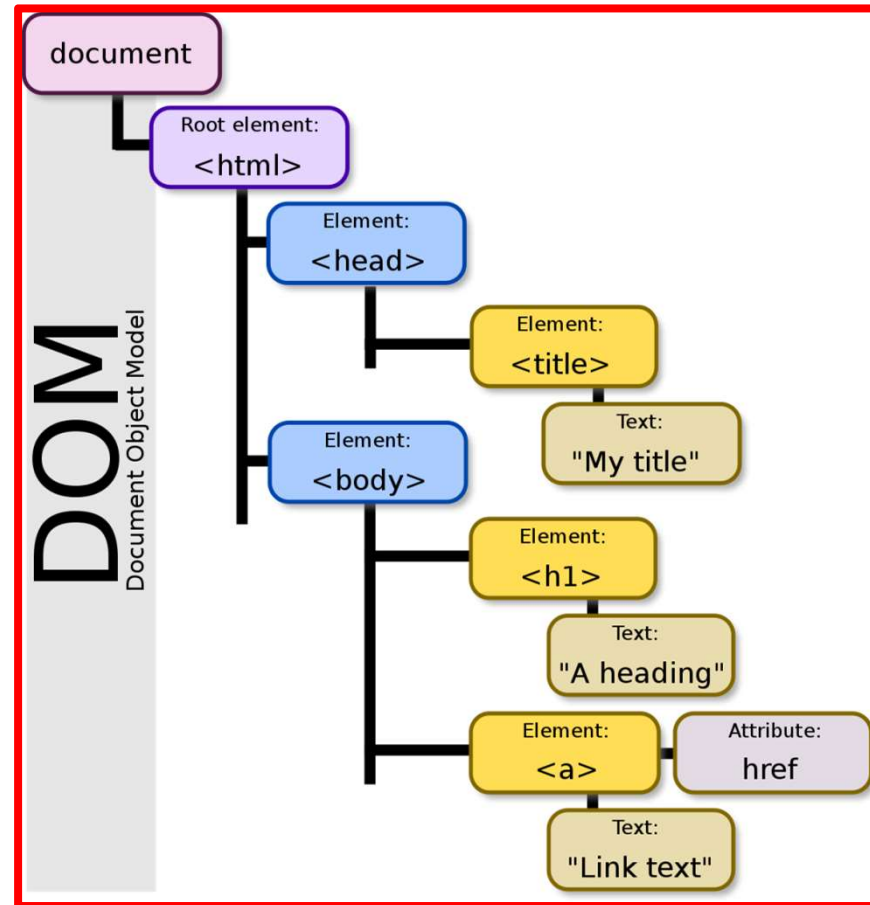
# Virtual DOM / Real DOM



- Preact JS Virtual DOM is an in-memory representation of the DOM. DOM refers to the **Document Object Model** that represents the content of XML or HTML documents as a tree structure so that the programs can be read, accessed and changed in the document structure, style, and content.

# What is DOM ?

- DOM stands for '**Document Object Model**'.
- In simple terms, it is a structured representation of the HTML elements that are present in a webpage or web app.
- DOM represents the entire UI of your application.
- The DOM is represented as a tree data structure. It contains a node for each UI element present in the web document.
- It is very useful as it allows web developers to modify content through JavaScript, also it being in structured format helps a lot as we can choose specific targets and all the code becomes much easier to work with.



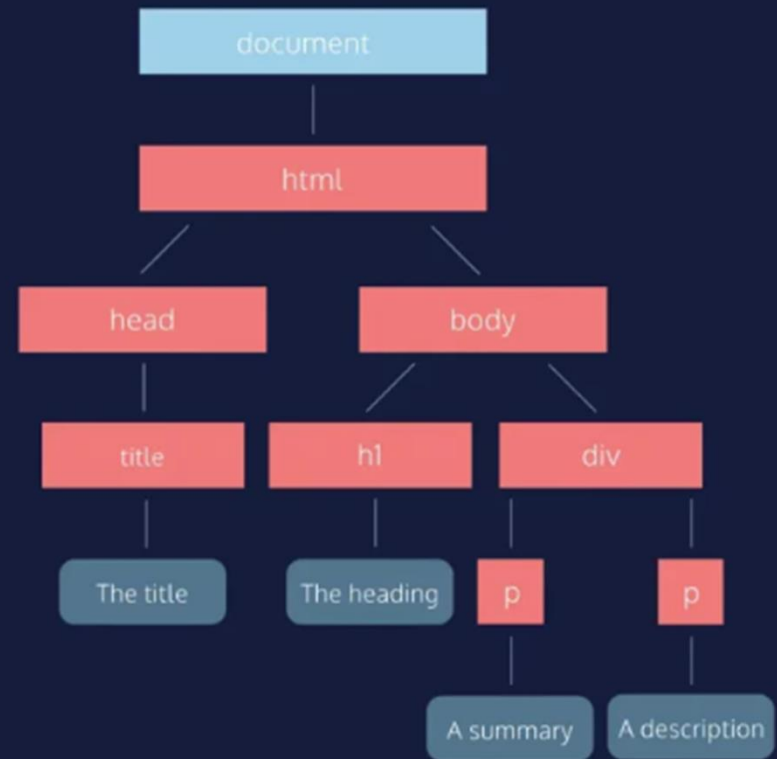
## HTML File

```
<html>
  <head>
    <title>The title</title>
  </head>

  <body>
    <h1>The heading</h1>
    <div>
      <p>A description</p>
      <p>A summary</p>
    </div>
  </body>
</html>
```

## DOM

Document Object Model



## Disadvantages of real DOM :

- Every time the DOM gets updated, the updated element and its children have to be rendered again to update the UI of our page. For this, each time there is a component update, the DOM needs to be updated and the UI components have to be re-rendered.

```
// Simple getElementById() method  
document.getElementById('some-id').innerHTML = 'updated value';
```

## Workflow

When writing the above code in the console or in the JavaScript file, these things happen:

- The browser parses the HTML to find the node with this id.
- It removes the child element of this specific element.
- Updates the element(DOM) with the 'updated value'.
- Recalculates the CSS for the parent and child nodes.
- Update the layout.
- Finally, traverse the tree and paint it on the screen(browser) display.

Recalculating the CSS and changing the layouts involves complex algorithms, and they do affect the performance.



# Virtual DOM

- Preact 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 Preact 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, the virtual DOM gets updated first instead of the real DOM.

## How does virtual DOM actually make things faster?

- When anything new is added to the application, a virtual DOM is created and it is represented as a tree.
- Each element in the application is a node in this tree. So, whenever there is a change in the state of any element, a new Virtual DOM tree is created.
- This new Virtual DOM tree is then compared with the previous Virtual DOM tree and make a note of the changes.
- After this, it finds the best possible ways to make these changes to the real DOM. Now only the updated elements will get rendered on the page again.

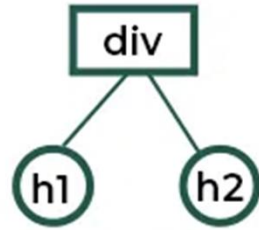
## How virtual DOM Helps Preact?

- In Preact, everything is treated as a component be it a functional component or class component. A component can contain a state. Whenever the state of any component is changed react updates its Virtual DOM tree. Though it may sound like it is ineffective the cost is not much significant as updating the virtual DOM doesn't take much time.
- 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.

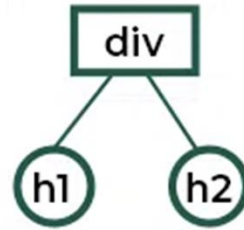
- Now it compares the pre-update version with the updated Virtual DOM and figures out what exactly has changed in the DOM like which components have been changed. This process of comparing the current Virtual DOM tree with the previous one is known as 'diffing'. Once React finds out what exactly has changed then it updates those objects only, on real DOM.
- React uses something called batch updates to update the real DOM. It just means that the changes to the real DOM are sent in batches instead of sending any update for a single change in the state of a component.

- We have seen that the re-rendering of the UI is the most expensive part and React manages to do this most efficiently by ensuring that the Real DOM receives batch updates to re-render the UI. This entire process of transforming changes to the real DOM is called **Reconciliation**.
- This significantly improves the performance and is the main reason why React and its Virtual DOM are much loved by developers all around.

Actual DOM

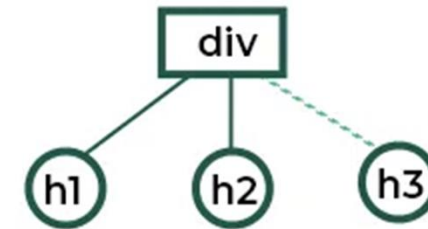


Old Virtual DOM

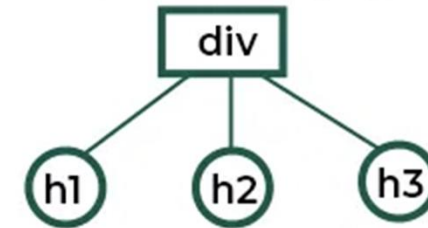


Compare

New Virtual DOM



New Actual DOM



**Browser DOM <=> Virtual DOM**

## Virtual DOM Key Concepts :

- Virtual DOM is the virtual representation of Real DOM
- React update the state changes in Virtual DOM first and then it syncs with Real DOM
- Virtual DOM is just like a blueprint of a machine, can do changes in the blueprint but those changes will not directly apply to the machine.
- Virtual DOM is a programming concept where a virtual representation of a UI is kept in memory synced with “Real DOM ” by a library such as ReactDOM and this process is called reconciliation
- Virtual DOM makes the performance faster, not because the processing itself is done in less time. The reason is the amount of changed information – rather than wasting time on updating the entire page, you can dissect it into small elements and interactions



## Differences between Virtual DOM and Real DOM

Virtual DOM	Real DOM
It is a lightweight copy of the original DOM	It is a tree representation of HTML elements
It is maintained by JavaScript libraries	It is maintained by the browser after parsing HTML elements
After manipulation it only re-renders changed elements	After manipulation, it re-render the entire DOM
Updates are lightweight	Updates are heavyweight
Performance is fast and UX is optimised	Performance is slow and the UX quality is low
Highly efficient as it performs batch updates	Less efficient due to re-rendering of DOM after each update

A Virtual DOM is a simple description of a tree structure using objects:

```
let vdom = {  
  type: 'p',          // a <p> element  
  props: {  
    class: 'big',     // with class="big"  
    children: [  
      'Hello World!' // and the text "Hello World!"  
    ]  
  }  
}
```

- Libraries like Preact provide a way to construct these descriptions, which can then be compared against the browser's DOM tree. As each part of the tree is compared, and the browser's DOM tree is updated to match the structure described by the Virtual DOM tree.
- This is a useful tool, because it lets us compose user interfaces *declaratively* rather than *imperatively*. Instead of describing *how* to update the DOM in response to things like keyboard or mouse input, we only need to describe *what* the DOM should look like after that input is received. It means we can repeatedly give Preact descriptions of tree structures, and it will update the browser's DOM tree to match each new description – regardless of its current structure.

## Creating Virtual DOM trees

There are a few ways to create Virtual DOM trees:

1. `createElement()`: a function provided by Preact
2. JSX: HTML-like syntax that can be compiled to JavaScript
3. HTM: HTML-like syntax you can write directly in JavaScript

- Simplest approach, which would be to call Preact's **createElement()** function directly:

```
import { createElement, render } from 'preact';

let vdom = createElement(
  'p',           // a <p> element
  { class: 'big' }, // with class="big"
  'Hello World!'  // and the text "Hello World!"
);

render(vdom, document.body);
```

- The code above creates a Virtual DOM "description" of a paragraph element. The first argument to createElement is the HTML element name. The second argument is the element's "props" - an object containing attributes (or properties) to set on the element. Any additional arguments are children for the element, which can be strings (like 'Hello World!') or Virtual DOM elements from additional createElement() calls.
- The last line tells Preact to build a real DOM tree that matches our Virtual DOM "description", and to insert that DOM tree into the <body> of a web page.

## Now with more JSX!

- We can rewrite the previous example using JSX without changing its functionality.

```
import { createElement, render } from 'preact';

let vdom = <p class="big">Hello World!</p>;

render(vdom, document.body);
```

```
let maybeBig = Math.random() > .5 ? 'big' : 'small';

let vdom = <p class={maybeBig}>Hello {40 + 2}!</p>;
           // ^---JS---^      ^---JS---^
```



- We can rewrite the previous example using JSX without changing its functionality. JSX lets us describe our paragraph element using HTML-like syntax, which can help keep things readable as we describe more complex trees. The drawback of JSX is that our code is no longer written in JavaScript, and must be compiled by a tool like Babel. Compilers do the work of converting the JSX example below into the exact `createElement()` code we saw in the previous example.
- It looks a lot more like HTML now!
- There's one final thing to keep in mind about JSX: code inside of a JSX element (within the angle brackets) is special syntax and not JavaScript. To use JavaScript syntax like numbers or variables, you first need to "jump" back out from JSX using an `{expression}` - similar to fields in a template. The example below shows two expressions: one to set class to a randomized string, and another to calculate a number.

## Once more with HTM

- HTM is an alternative to JSX that uses standard JavaScript tagged templates, removing the need for a compiler.
- If you haven't encountered tagged templates, they're a special type of String literal that can contain `${expression}` fields:

```
let str = `Quantity: ${40 + 2} units`; // "Quantity: 42 units"
```

- HTM uses `${expression}` instead of the `{expression}` syntax from JSX, which can make it clearer what parts of your code are HTM/JSX elements, and what parts are plain JavaScript:

```
import { html } from 'htm/preact';

let maybeBig = Math.random() > .5 ? 'big' : 'small';

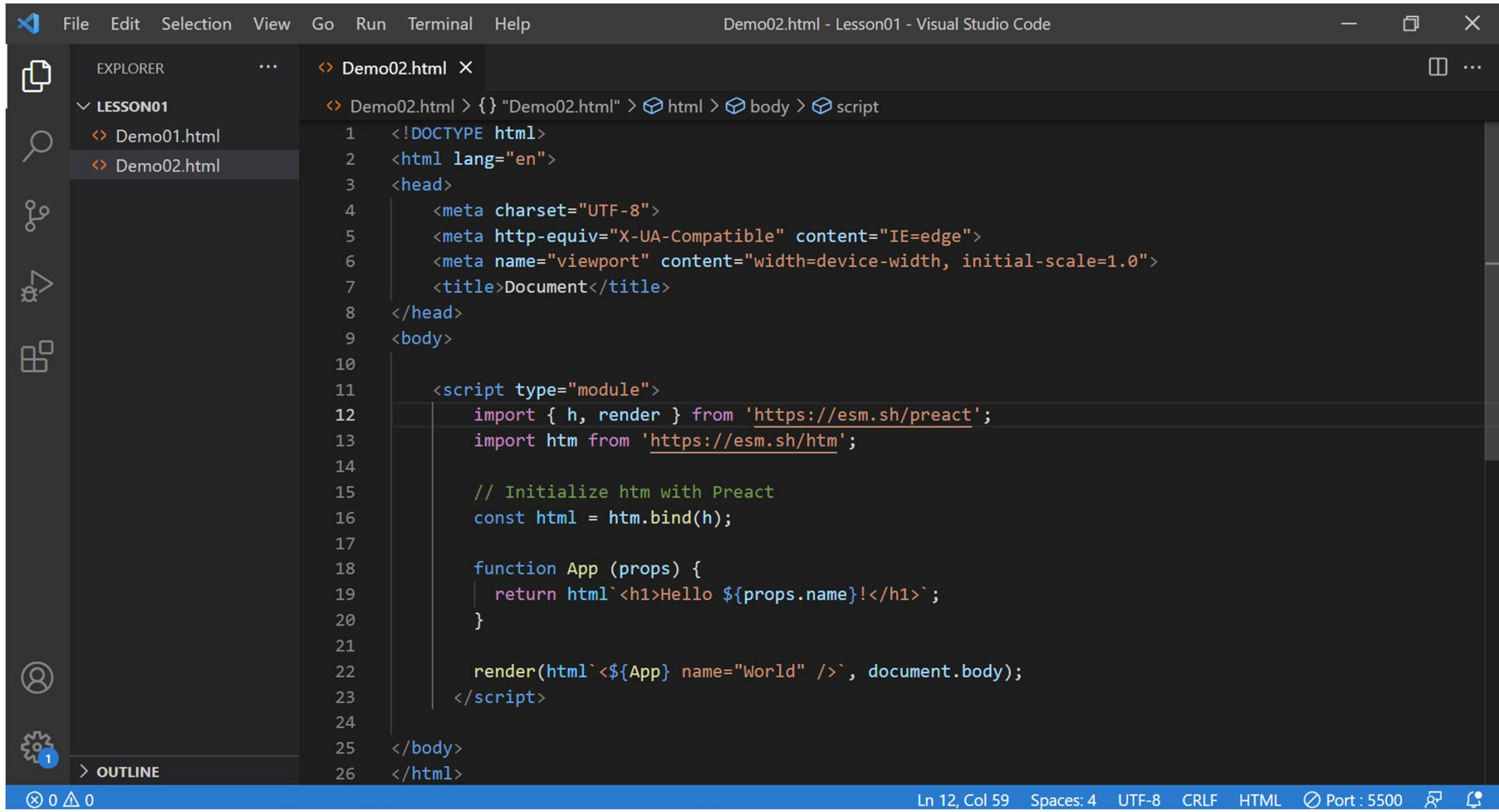
let vdom = html`<p class=${maybeBig}>Hello ${40 + 2}!</p>`;
                // ^--JS--^           ^-JS-^
```

Visual Studio Code interface showing the file explorer on the left with 'LESSON01' and 'Demo01.html'. The main editor displays the code for 'Demo01.html' with the following content:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatible" content="IE=edge">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>Document</title>
8 </head>
9 <body>
10
11   <script type="module">
12     import { h, render } from 'https://esm.sh/preact';
13
14     import { html } from 'https://esm.sh/htm/preact/standalone'
15
16     let maybeBig = Math.random() > .5 ? 'big' : 'small';
17
18     let vdom = html`<p class=${maybeBig}>Hello ${40 + 2}</p>`;
19
20     render(vdom, document.body);
21   </script>
22
23 </body>
24 </html>
```

On the right, a browser window is open at '127.0.0.1:5500/Demo01.html', displaying the output 'Hello 42!'.

# As a Component



The screenshot shows the Visual Studio Code editor interface. The Explorer sidebar on the left shows a project named 'LESSON01' with two files: 'Demo01.html' and 'Demo02.html'. The main editor area is open to 'Demo02.html'. The breadcrumb navigation at the top of the editor shows the file path: 'Demo02.html > {} "Demo02.html" > html > body > script'. The code in the editor is as follows:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatible" content="IE=edge">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>Document</title>
8 </head>
9 <body>
10
11   <script type="module">
12     import { h, render } from 'https://esm.sh/preact';
13     import htm from 'https://esm.sh/htm';
14
15     // Initialize htm with Preact
16     const htm1 = htm.bind(h);
17
18     function App (props) {
19       return htm1`<h1>Hello ${props.name}!</h1>`;
20     }
21
22     render(htm1`<${App} name="World" />`, document.body);
23   </script>
24
25 </body>
26 </html>
```

The status bar at the bottom indicates the current position is 'Ln 12, Col 59', with settings for 'Spaces: 4', 'UTF-8', 'CRLF', 'HTML', and 'Port : 5500'.

