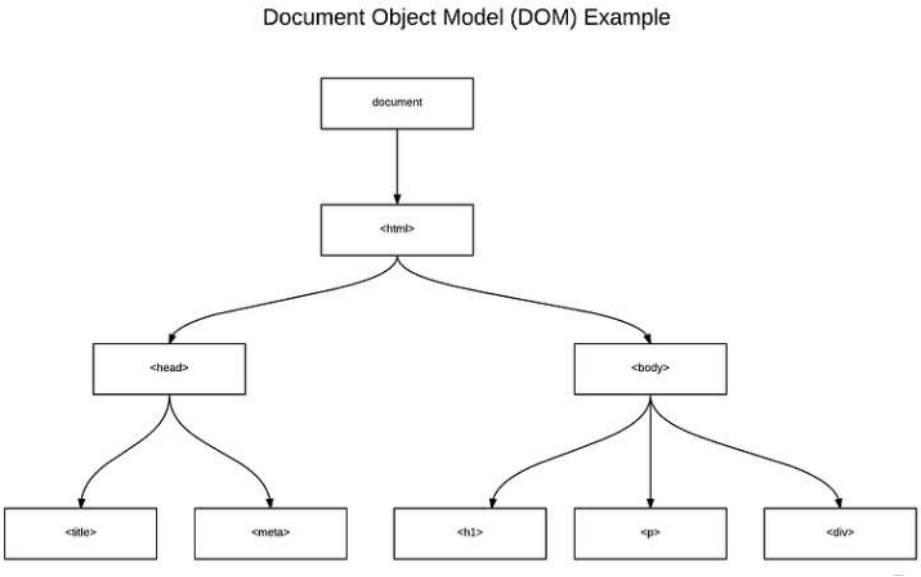
Virtual DOM

In React, for every DOM object, there is a corresponding “virtual DOM object.” A virtual DOM object is a representation of a DOM object, like a lightweight copy.

A virtual DOM object has the same properties as a real DOM object, but it lacks the real thing’s power to directly change what’s on the screen.

Manipulating the DOM is slow. Manipulating the virtual DOM is much faster because nothing gets drawn onscreen. Think of manipulating the virtual DOM as editing a blueprint, as opposed to moving rooms in an actual house.



let fruits = [‘Apple’, ‘Orange’, ‘Banana’]

We want to update here from Orange to lemon. Then we need to create a new array.

let fruits = [‘Apple’, ‘Lemon’, ‘Banana’]

In an efficient way we can just traverse to the *fruits[2]* and update only this element.

The Virtual DOM is a light-weight abstraction of the DOM. You can think of it as a copy of the DOM, that can be updated without affecting the actual DOM. It has all the same properties as the real DOM object, but doesn’t have the ability to write to the screen like the real DOM. The virtual DOM gains it’s speed and efficiency from the fact that it’s lightweight. In fact, a new virtual DOM is created after every re-render.

Reconciliation is a process to compare and keep in sync the two files (Real and Virtual DOM). Diffing algorithm is a technique of reconciliation which is used by React.

How does updates work in React?

- On the first load, *ReactDOM.render()* will create the Virtual DOM tree and real DOM tree.
- As React works on Observable patterns, when any event(like key press, left click, api response, etc.) occurred, Virtual DOM tree nodes are notified for props change, If the properties used in that node are updated, the node is updated else left as it is.

- React compares Virtual DOM with real DOM and updates real DOM. This process is called Reconciliation. React uses Diffing algorithm techniques of Reconciliation.
- Updated real DOM is repainted on browser.

Reconciliation

React compares the Virtual DOM with Real DOM. It finds out the changed nodes and updates only the changed nodes in Real DOM leaving the rest nodes as it is. This process is called Reconciliation.

Diffing Algorithm

React first compares the two root elements. The behavior is different depending on the types of the root elements.

React compared the root DOM Elements Types.

- **Elements of different types**: Whenever the root elements have different types, React will tear down the old tree and build the new tree from scratch. Going from <a> to <img>, or from <Article> to <Comment>, or from <Button> to — any of those will lead to a full rebuild. This will lead to component unmount and mount lifecycle calls too.
- **DOM Elements Of The Same Type**: When comparing two React DOM elements of the same type, React looks at the attributes of both, keeps the same underlying DOM node, and only updates the changed attributes.After

handling the DOM node, React then recurses on the children. This will lead to component update lifecycle calls.

key should be used in lists?

By default, when recursing on the children of a DOM node, React just iterates over both lists of children at the same time and generates a mutation whenever there's a difference.

Old

<ul>

*
first
*

*
second
*

*
*

New

<ul>

*
first
*

*
second
*

*
third
*

*
*

Here React will add a third element directly. But if the order changes then react gets confused in comparing. To overcome this **key** is introduced.

<ul>

*<li key=”2015">Duke
*

*<li key=”2016">Villanova
*

*
*

Updated list.

<ul>

*<li key=”2014">Connecticut
*

*<li key=”2015">Duke
*

*<li key=”2016">Villanova
*

*
*

Here react will identify the component with key and update only the changed items.

Keys should be stable, predictable, and unique. Unstable keys (like those produced by *Math.random()*) will cause many component instances and DOM nodes to be unnecessarily recreated, which can cause performance degradation and lost state in child components.

We don’t recommend using indexes for keys if the order of items may change. This can negatively impact performance and may cause issues with component state.