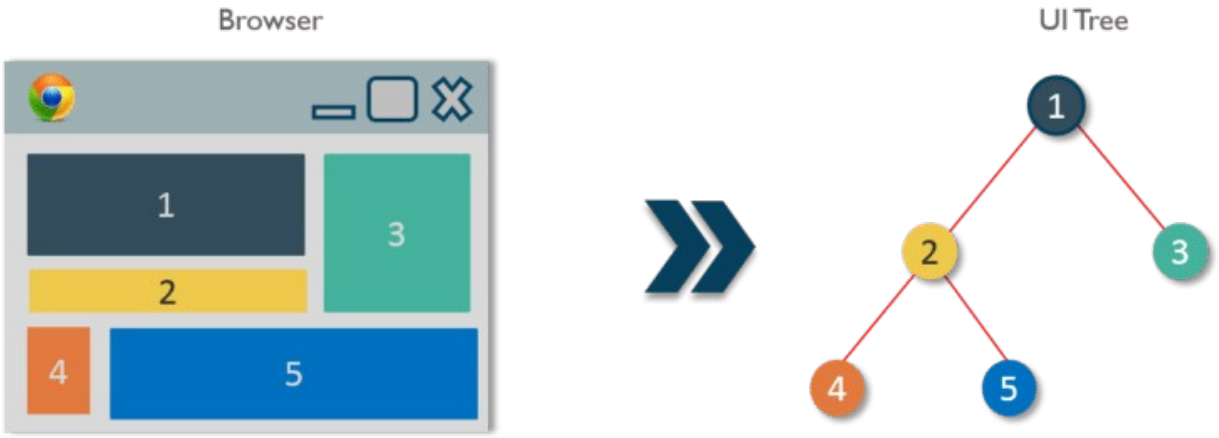
## Introduction to Components, props and states 38th Day

Earlier the developers had to write 1000 lines of code for developing a simple single page application. Most of those applications followed the traditional DOM structure and making changes to them was very challenging and a tedious task for the developers. They manually had to search for the element which needed the change and update it accordingly. Even a small mistake would lead to application failure. Moreover, updating DOM was very expensive. Thus, the component–based approach was introduced. In this approach, the entire application is divided into logical chunks which are called the **Components**. React was one of the frameworks who opted for component based approach.

React components are considered as the building blocks of the User Interface. Each of these components exists within the same space but execute independently from one another. React components have their own structure, methods as well as APIs. They are reusable and can be injected into interfaces as per need. To have a better understanding, consider the entire UI as a tree. Here the starting component becomes the root and each of the independent pieces becomes branches, which are further divided into sub–branches.




According to Facebook react official docs, **“components are like JavaScript functions**. **They accept arbitrary inputs** (**called “props”**) **and return React elements describing what should appear on the screen**.**”**

**Advantages of React Components**

1). **Code Re-usability** − A component-based approach makes your application development easier and faster. If you want to use a pre-existing functionality in your code, you can just put that code in yours instead of building it from scratch. It also allows your application architecture to stay up to date over time as you can update the specific areas which need up-gradations.

2). **Fast Development** − A component-based UI approach leads to an iterative and agile application development. These components are hosted in a library from which different software development teams can access, integrate and modify them throughout the development process.

3). **Consistency** − Implementing these reusable components helps to keep the design consistent and can provide clarity in organizing code throughout the application.

4). **Maintainability** − Applications with a set of well–organized components can be quickly updated and you can be confident about the areas which will be affected and which won't.

5). **Scalability** − The development becomes easier with a properly organized library of ready to implement components. Ensuring the components are properly namespaced helps to avoid style and functionality leaking or overlapping into the wrong place as the project scales up.

6). **Easy Integration** − The component codes are stored in repositories like GitHub, which is open for public use. Application development teams are well–versed in using source code repositories, and so they are able to extract the code as needed and inject it into the application.

The main two ways that components receives data is by using Props and States.

**Props**

Props stand for Properties. They are the read–only components which work similar to the HTML attributes. Prop is a way of passing data from parent to child component.

**Example**:

```
import React from 'react';

import ReactDOM from 'react-dom';

class MyComponent extends React.Component{
        render(){
        return(
                <div>
                  <h1>Hello</h1>
                  <Header name="maxx" id="101"/>
            </div>
        );
    }
}

function Header(props) {
    return (
            <div>
              <Footer name = {props.name} id = {props.id}/>
            </div>
    );
}
function Footer(props) {
    return (
            <div>
              <h1> Welcome : {props.name}</h1>
              <h1> Id is : {props.id}</h1>
            </div>
   );
}
ReactDOM.render(
    <MyComponent/>, document.getElementById('content')
);
```

Since the props can only be passed from parent components, they cannot be changed. This makes them immutable and dumb. This poses a great challenge as the modern apps do not have all of its states ready on page load. Ajax or Events can happen when data returns, so someone needs to take responsibility for handling the updates. This is where React states come into the picture.

**States**

Generally, components take in props and render them. These are called stateless components. But they can also provide state which are used to store data or information about the component which can change over time. Such components are called stateful components. The change in state can happen as a response to user event or system event. In other words, **state** is the heart of every react component which determines how the component will behave and render. They are also responsible for making a component dynamic and interactive. Thus they must be kept as simple as possible.

The state can be accessed with this reference, e.g., this.state. You can access and print variables in JSX using curly braces {}. Similarly, you can render this.state inside render(). You must set a default state for the component else it will set to null.

**Example**:

```
import React from 'react';

import ReactDOM from 'react-dom';

class MyComponent extends React.Component {
	constructor() {
		super();
		this.state = {
			name: 'Maxx',
			id: '101'
		}
	}
	render()
		{
			setTimeout(()=>;{this.setState({name:'Jaeha', id:'222'})},2000)
			return (
				<div>
				   <h1>Hello {this.state.name}</h1>
				   <h2>Your Id is {this.state.id}</h2>
				</div>
			  );
		}
	}
ReactDOM.render(
	<MyComponent/>, document.getElementById('content')
);
```

**React Component Lifecycle**

React provides various methods which notify when a certain stage in the lifecycle of a component occurs. These methods are called the lifecycle methods. These lifecycle methods are not very complicated. You can think of these methods as specialized event handlers that are called at various points during a components life. You can even add your own code to these methods to perform various tasks. Talking about the lifecycle of the component, the lifecycle is divided into 4 phases.

They are:

1. Initial Phase
2. Updating Phase
3. Props change Phase
4. Unmounting Phase

Each of these phases contains some lifecycle methods which are specific only to them. So let's now find out what happens during each of these phases.

**a. Initial Phase** – The first phase of the lifecycle of a React component is the initial phase or initial rendering phase. In this phase, the component is about to start its journey and make its way to the DOM. This phase consists of the following methods which are invoked in a predefined order.

1. **GetDefaultProps**(): This method is used to specify the default value of this.props. It gets called before your component is even created or any props from the parent are passed into it.
2. **GetInitialState**(): This method is used to specify the default value of this.state before your component is created.
3. **ComponentWillMount**(): This is the last method that you can call before your component gets rendered into the DOM. But if you call setState() inside this method your component will not re–render.
4. **Render**(): This method is responsible for returning a single root HTML node and must be defined in each and every component. You can return null or false in case you don't want to render anything.

5. **ComponentDidMount**(): Once the component is rendered and placed on the DOM, this method is called. Here you can perform any DOM querying operations.

**b**. **Updating Phase** – Once the component is added to the DOM, they can update and re–render only when a state change occurs. Each time the state changes, the component calls its render() again. Any component, that relies on the output of this component will also call its render() again. This is done, to ensure that our component is displaying the latest version of itself. Thus to successfully update the components state the following methods are invoked in the given order:

1. **shouldComponentUpdate**(): Using this method you can control your component's behavior of updating itself. If you return a true from this method, the component will update. Else if this method returns a false, the component will skip the updating.

2. **ComponentWillUpdate**(): This method is called just before your component is about to update. In this method, you can't change your component state by calling this.setState.

3. **Render**(): If you are returning false via should Component Update() (**should Component Update**() ), the code inside **render**() will be invoked again to ensure that your component displays itself properly.

4. **ComponentDidUpdate**(): Once the component is updated and rendered, then this method is invoked. You can put any code inside this method, which you want to execute once the component is updated.

**c**. **Props Change Phase** − After the component has been rendered into the DOM, the only other time the component will update, apart from the state change is when its prop value changes. Practically this phase works similar to the previous phase, but instead of the state, it deals with the props. Thus, this phase has only one additional method from the Updating Phase.

1. **ComponentWillReceiveProps**(): This method returns one argument which contains the new prop value that is about to be assigned to the component. Rest of the lifecycle methods behave identically to the methods which we saw in the previous phase.
2. **shouldComponentUpdate**()
3. **componentWillUpdate**()
4. **render**()
5. **componentDidUpdate**()

**d**). **The Unmounting Phase** − This is the last phase of components life cycle in which the component is destroyed and removed from the DOM completely.

The Unmounting Phase contains only one phase:

1. **ComponentWillUnmount**(): Once this method is invoked, your component is removed from the DOM permanently. In this method, you can perform any clean-up related tasks like removing event listeners, stopping timers, etc.

We decided to focus on the theoretical part of Component, states and props. We will later have a practical session.

Materials:

**https://reactjs.org/docs/react-component.html**

https://medium.com/javascript-in-plain-english/react-components-and-props-explained-for-non-devs-d801399ed429#:~:text=In%20React%2C%20props%20are%20properties,JS%20function%20serve%20as%20arguments.&text=Welcome(props)%20%7B-,return%20Hello%2C%20%7Bprops.,name%7D%3B&text=props%20is%20an%20object%2C%20so,name%3A%20%E2%80%9Cvalue%E2%80%9D%7D.

**Best Wishes, Lux Tech Academy**