Class 2 ReactJS

Components

ReactJS applications are built around a component-based architecture. A component in React "represents a part of the user interface." This modular approach allows for the breakdown of complex UIs into smaller, manageable, and self-contained pieces.

Key properties of React Components:

1. **Represent a Part of the UI:** Each component describes a specific section or element of the user interface. For example, an application might have components for a "header," "side nav," "main content," and "footer."
2. **Hierarchical Structure (Nesting):** Components can contain other components. A "root component" (often named App component) typically contains all other components, forming a tree-like structure. The source notes, "app component contains the other components."
3. **Reusability:** Components are designed to be reusable. The same component can be employed multiple times, potentially with "different properties to display different information." An example given is a "side nav component" being used for both a left and right side navigation.
4. **Building Blocks:** Components are fundamental to any React application.

Component Code Structure and File Naming

1. File Extension: .js and .jsx
2. Code Location: The "component code is usually placed in a JavaScript file." For instance, the App component is found in App.js.

Types of React Components

1. **Stateless Functional Components:**
   Description: These are "literally JavaScript functions."

Functionality: Their main purpose is to "return HTML which describes the UI."
Example:

```
// HelloFunctional.js
import React from 'react';

function HelloFunctional() {
  return <h1>Hello, World!</h1>;
}

export default HelloFunctional;
```

Key Feature: They do not manage their own internal state (hence "stateless").

2. **Stateful Class Components:**
   Description: These are "regular es6 classes that extend the Component class from the react library."
   Requirement: They "must contain a render method."
   Functionality: The render method "in turn returns HTML."
   Example:

```
// HelloClass.js
import React, { Component } from 'react';

class HelloClass extends Component {
  render() {
    return <h1>Hello, World!</h1>;
  }
}

export default HelloClass;
```

Key Feature: They can manage their own internal state (hence "stateful"). The source indicates that the App component in the simple "hello world" example is a class component.

Functional Components

Functional components are a fundamental building block in ReactJS. They are defined as simple JavaScript functions that are responsible for describing a part of the user interface (UI).
Definition: "Functional components are just JavaScript functions."
Inputs: They "can optionally receive an object of properties which is referred to as props."
Outputs: They "return HTML which describes the UI." The tutorial clarifies that this "HTML is actually something known as JSX, but for the sake of understanding from a beginner's point of view let's just call it HTML."

Creating a Functional Component: Step-by-Step

1. *File Structure:* Create a new folder (e.g., components) and a new JavaScript file for the component (e.g., Greet.js).
2. *Naming Convention:* For component files, use "Pascal case" (e.g., Greet.js).
3. *Import React:* Every component file must import React.
   import React from 'react';
4. *Define the Function:* Create a JavaScript function that returns the UI.
5. *Initial Approach:*
   function Greet() {
   return <h1>Hello LPU</h1>;
   }
6. *Export the Component:* The component must be exported so it can be used in other parts of the application.

Exporting and Importing Components

There are two primary ways to export and import components: Default Exports and Named Exports.

1. Default Exports (Most Common)
   Definition: "This is what allows us to import the component with any name."
   Export Syntax: Prepend export default to the function or variable declaration.

   ```
   // In Greet.js
   const Greet = () => {
      return <h1>Hello LPU</h1>;
   };
   export default Greet;
   ```

   Import Syntax: Import the component without curly braces. The imported name can be different from the exported name.

   ```
   // In App.js
   import MyComponent from './components/Greet'; // 'MyComponent' can be any name
   // Usage: <MyComponent />
   ```

2. Named Exports
   Definition: "In this situation you have to import the component with the exact same name."
   Export Syntax: Prepend export to the function or variable declaration.

   ```
   // In Greet.js
   export const Greet = () => { // Note 'export' keyword
      return <h1>Hello LPU</h1>;
   };
   ```

   Import Syntax: Import the component using curly braces and the exact same name as it was exported.

   ```
   // In App.js
   import { Greet } from './components/Greet'; // Must be 'Greet'
   ```

// Usage: <Greet />

Attempting to import a named export as a default export will result in an "Attempted import error."

Including Components in the Application

Once a component is created and exported, it needs to be imported into another component (e.g., App.js) to be rendered.

1. Import:// In App.js
   import Greet from './components/Greet'; // Assuming default export
   // The '.js' extension can be omitted: import Greet from './components/Greet';
2. Usage (as a custom HTML tag):// In App.js's return statement

   <Greet></Greet>
   {/* Or, if no content between tags, use a self-closing tag: */}
   <Greet />

This process "allows us to include it in the app component." After saving and viewing in the browser, "your first functional component is up and running."

JSX

JSX, or JavaScript XML, is described as "*an extension to the JavaScript language syntax with the React library.*" It allows developers to "write XML-like code for elements and components" directly within their JavaScript files.

How Does JSX Work (Behind the Scenes)?

The JSX element is just syntactic sugar for calling React.createElement."

What is React.createElement?

The React.createElement method is the underlying JavaScript function that JSX sugarcoats. It accepts a minimum of three parameters:

1. *HTML Tag (as a string):* The type of HTML element to be rendered (e.g., 'div', 'h1').
2. *Optional Properties (as an object):* A JavaScript object containing key-value pairs representing attributes for the element (e.g., { id: 'hello', className: 'dummy-class' }). If no properties are needed, null is passed.
3. *Children (variable arguments):* Subsequent parameters represent the children of the element. These can be:
    a. *Plain text strings*.

*The React.createElement method can accept "*any number of elements as children.*"

Example Comparison (JSX vs. React.createElement):

This demonstrates the complexity of writing a simple "Hello LPU" component without JSX:

JSX Version:

```
<div>
  <h1>Hello LPU</h1>
</div>
```

React.createElement Version:

```
React.createElement(
  'div',
  null,
  React.createElement(
    'h1',
    null,
    'Hello LPU
  )
```

);
This comparison highlights how React.createElement can become "really clumsy" for components with many elements, underscoring the value of JSX.

Key Differences: JSX vs. Regular HTML Attributes

While JSX resembles HTML, there are crucial differences, primarily due to JavaScript reserved words and camelCasing conventions:

class vs. className:

1. In HTML, you use class for CSS classes.
2. In JSX, class is a JavaScript reserved word (used for defining classes/components). Therefore, you must use className for applying CSS classes.

for vs. htmlFor:

1. Similar to class, for is a JavaScript keyword.
2. In JSX, for form element labels (<label>), the for attribute is replaced by htmlFor.

CamelCasing for HTML Attributes:

1. HTML attributes are typically kebab-cased (e.g., onclick, tabindex).
2. In JSX, many attributes follow camelCasing convention (e.g., onClick, tabIndex). "We will see these differences as we progress through the series so don't worry about having to memorize them."

Introduction to Props

Props is a fundamental mechanism for passing data between components and making them dynamic and reusable.

Key Concepts:

1. *Optional Input:* Props are optional inputs that a React component can accept.

2. *Dynamic Components:* They allow components to be dynamic, meaning they can display different information based on the data passed to them.
3. *Reusability:* Props enhance component reusability.
4. *Immutable:* Props value cannot be changed within the component that receives them.
5. *Any DataTypes:* Props has the ability to pass different kinds of data, including primitive values (strings) and complex JSX/HTML structures

How Props Work

Props are specified as attributes when invoking a component, similar to HTML attributes.

Syntax: <ComponentName attributeName="value" />
Example: To pass a name property to a Greet component, you would write <Greet name="Munit" />.

Retrieving Props in Functional Components

1. *Add a Parameter*: The functional component's definition should include a parameter, conventionally named props. "You can actually name this anything you want to but the convention is to name it drops and I highly recommend you don't deviate from this."
2. *Use the Parameter:* Access individual prop values using dot notation (e.g., props.name, props.college, props.branch).

Example:

```
function Greet(props) {
   console.log(props); // Logs the props object to the console
   return <h1>Hello {props.name} from {props.college}</h1>;
}
```

JSX Evaluation: To display prop values within JSX, they must be wrapped in curly braces {}. This tells React to evaluate the expression rather than treating it as plain text.

Destructure Props:

Destructuring is a JavaScript ES6 feature that allows you to unpack properties from objects into distinct variables.

Instead of doing:

```
const studentName = props.studentName;
const college = props.college;
```

You can do:

```
const { studentName, college } = props;
```

Example:

```
function Greet(props) {
  const { studentName, college } = props;
  return (
    <h1>
      Hello {studentName}, from {college}
    </h1>
  );
}
```

Spread/Rest Operator (...):

The ... (spread/rest operator) is used to collect the remaining properties that were not destructured.

In our code:

```
const { studentName, college, ...prop } = props;

function Greet(props) {
  const { studentName, college, ...prop } = props; //destructure props
  return (
    <h1>
```

```
      Hello {studentName}, from {prop.branch} Branch, {college}
    </h1>
  );
}
```

Props Drilling:

Props drilling refers to the process of passing data from a top-level component to deeply nested child components via props - even if intermediate components don’t need that data.

Example:

Component Structure:
App → Parent → Child → Grandchild

Code:

```
// App.js
function App() {
  const studentName = "Nitesh";
  return <Parent studentName={studentName} />;
}
// Parent.js
function Parent({ studentName }) {
  return <Child studentName={studentName} />;
}
// Child.js
function Child({ studentName }) {
  return <Grandchild studentName={studentName} />;
}
// Grandchild.js
function Grandchild({ studentName }) {
  return <h2>Hello, {studentName}</h2>;
}
```

Class 3 ReactJS

Introduction to Hooks:

What Are Hooks?

Hooks are special functions in React that let you “hook into” React features like state, lifecycle, and context - without using classes.

Why Were Hooks Introduced?

Before hooks, to manage things like state or lifecycle methods (e.g., componentDidMount), you had to use class components.

Hooks allow you to:

1. Use state and other features inside functional components
2. Write cleaner, reusable code
3. Avoid class-related confusion (like this binding)

Functional components were stateless before hooks - but now they can do everything class components can (and more).

Introduction to Component State

Definition: State is like a *memory* for a component.
It stores data that can change over time, and when it changes, the component re-renders (updates on the screen).

React components utilize two primary mechanisms to influence what is displayed on the screen: props and state. While props are passed to a component, state is managed within a component and holds information that "influences the UI in the browser."

Props vs. State: A Comparison

Feature	Props	State
Definition	Data passed from parent to child	Data managed within the component
Ownership	Controlled by parent component	Controlled by the same component
Editable?	Read-only	Mutable (using useState)
Trigger Re-render?	Yes, if parent changes the prop	Yes, when state is updated
Use Case	To configure or customize a component	To handle dynamic data / user interaction
Data Flow Direction	Top-down (from parent to child)	Internal to the component
Syntax Example (Usage)	<Profile name="Munit" />	const [count, setCount] = useState(0);

Syntax of useState:

const [stateVariable, setStateFunction] = useState(initialValue);

1. stateVariable⇒Current state value
2. setStateFunction ⇒ Function to update the value (triggers re-render)
3. initialValue ⇒Starting value of the state (string, number, array, etc.)

Implementing State in Components

Example:

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

function LikeButton({ name }) {
  const [likes, setLikes] = useState(0);

  return (
    <div>
      <h2>Hello, {name}!</h2>
      <p>Likes: {likes}</p>
      <button onClick={() ⇒ setLikes(likes + 1)}>Like</button>
    </div>
```

```
  );
}

export default LikeButton;
```

Timer Webapp

```
import React, { useState } from "react";

function Timer() {
  const [time, setTime] = useState(0);
  const [interval, setInter] = useState(null);

  function startTimer() {
    if (interval === null) {
      const i = setInterval(() => {
        setTime((prev) => prev + 1);
      }, 1000);
      setInter(i);
    } else console.log("Timer already started!");
  }

  function stopTimer() {
    clearInterval(interval);
    setInter(null);
  }

  function resetTimer() {
    setTime(0);
  }

  return (
    <main style={{ textAlign: "center" }}>
      <h1>Timer: {time}</h1>
      <div>
        <button onClick={startTimer}>Start</button>
```

```
           
        <button onClick={stopTimer}>Stop </button>
           
        <button onClick={resetTimer}>Reset </button>
      </div>
    </main>
  );
}

export default Timer;
```

Explanation

What startTimer does:

- It starts the timer only if it's not already running.
- setInterval(...) calls the callback function every 1000 milliseconds (1 second).
- Inside the interval callback, setTime increases the timer by 1 every second.

Why setTime((prev) ⇒ prev + 1) instead of setTime(time + 1)?

If you wrote: `setTime(time + 1);`

It may not work correctly because time might not reflect the latest value during asynchronous updates.

React state updates are batched and asynchronous, so using the previous value ensures accuracy.

Correct way: `setTime((prev) ⇒ prev + 1);`

This ensures you're always updating based on the latest internal state, not what was available when the function was created (due to closures).

What stopTimer does:

- Stop the interval using the ID stored in the interval.
- Clears the state to allow startTimer() to run again next time.

Why use useState for interval instead of a let or const variable?
You could use a normal let variable like this:
let intervalId = null;

However, this would not persist between re-renders of the component because function components re-run every time they render.

Each render would re-initialize intervalId to null and forget the previous value. So, the interval would be lost or not cleared correctly.

Why useState is used:

- React state (useState) persists across renders.
- So when the component re-renders, the interval ID is remembered.
- This allows us to:
    - Know if a timer is already running (interval !== null)
    - Stop it later using the correct ID