Assignment 4 - Talk is cheap, show me the code

1. **Is JSX mandatory for React?**

   No, JSX is not mandatory for React. You can use React without JSX, but it is not recommended. JSX is a syntax extension for JavaScript that allows you to write HTML-like code within JavaScript. This makes it easier to write React components and to define their structure and behavior.
   JSX makes our code more readable. Also, the JSX code gets transplied to React. createElement() using transpilers like Babel during the build process.

   So, the below 2 lines do the same thing.

   const element = <h1>Hello World</h1>;

   const element = React.createElement('h1', null, 'Hello World');

2. **Is ES6 mandatory for React?**

   No, ES6 is not mandatory for React. You can use React without ES6, but it is not recommended. ES6 is a newer version of JavaScript that includes a number of features that make it easier to write React code. For example, ES6 includes classes, arrow functions, and modules, which are all used extensively in React.

   If you do not use ES6, you will need to use the older JavaScript syntax, which can be more verbose and less concise. Additionally, you will not be able to take advantage of some of the newer features that are available in ES6.

3. **{TitleComponent} vs {<TitleComponent />} vs {<TitleComponent> </TitleComponent>} in JSX.**

- **{TitleComponent}:** This value describes the TitleComponent as a javascript expression or a variable. The {} can embed a javascript expression or a variable inside it.
- **<TitleComponent/> :** This value represents a Component that is basically returning Some JSX value. In simple terms TitleComponent is a component that is returning some JSX value. A component is written inside the {< />} expression.
- **<TitleComponent></TitleComponent> :** <TitleComponent /> and <TitleComponent></TitleComponent> are equivalent only when < TitleComponent />

has no child components. The opening and closing tags are created to include the child components.

4. **How can I write comments in JSX ?**

   In JSX, you can write comments using the following syntax: {/* comment here */}. This is the regular way to write comments in JSX. The comment must be enclosed inside a pair of curly braces {} and use the forward-slash and asterisk syntax /* comment here */. Here's an example:

```
export default function App() {
  return (
    <div>
      <h1>Commenting in React and JSX~ </h1>
      {/* <p>My name is Bob</p> */}
      <p>Nice to meet you!</p>
    </div>
  );
}
```

5. **What is <React.Fragment></React.Fragment> and <> </> ?**

   In React, <React.Fragment> and <> (empty JSX fragment) are a special type of element that can be used to group multiple children without creating an extra DOM node.

   The React.Fragment syntax is the older syntax for creating a fragment. It is still supported, but it is not recommended to use it anymore. The <> </> syntax is the newer syntax for creating a fragment. It is more concise and easier to read.

   Example:

```
JavaScript
const App = () => {
  return (
    <>
      <h1>This is a fragment</h1>
      <p>This is another child of the fragment</p>
    </>
  );
};
```

6. **What is Virtual DOM ?**

   A virtual DOM (VDOM) is a lightweight JavaScript representation of the Document Object Model (DOM) used in declarative web frameworks such as React, Vue.js, and Elm. Updating the virtual DOM is comparatively faster than updating the actual DOM (via JavaScript). Thus, the framework is free to make necessary changes to the virtual DOM relatively cheaply. The framework then finds the differences between the previous virtual DOM and the current one, and only makes the necessary changes to the actual DOM.

7. **What is Reconciliation in React ?**

   Reconciliation in React refers to the process of updating the actual DOM to reflect the changes made to the React components and their underlying data. When the state or props of a React component change, React needs to determine what updates are required in the DOM to keep it in sync with the new state.

   The reconciliation algorithm is the set of rules that React uses to determine how to update the DOM in the most efficient way possible.

   The reconciliation algorithm works by comparing the current virtual DOM with the new one, calculating the differences (diff algorithm or diffing), and then making the necessary updates to the actual DOM. The diff algorithm is responsible for identifying the differences between the two DOM trees and determining which elements need to be updated.

   The reconciliation algorithm is a critical part of React's performance and helps make React one of the fastest and most efficient JavaScript libraries for building user interfaces.

8. **What is React Fiber ?**

   React Fiber is a complete rewrite of React's reconciliation algorithm. It was introduced in React 16.6 and is the default reconciliation algorithm for React 17 and beyond.

React Fiber is designed to be more efficient and flexible than the previous reconciliation algorithm. It also introduces a number of new features, such as the ability to pause and resume rendering, and the ability to prioritize work.

React Fiber is asynchronous whereas the old reconciliation algorithm used a stack and was synchronous.
https://www.geeksforgeeks.org/what-is-react-fiber/

9. **Why do we need keys in React? When do we need keys in React?**

In React, keys are used to provide a unique identifier for elements in an array of components or elements rendered within a loop. Keys help React efficiently update the Virtual DOM and the actual DOM during the reconciliation process.

When we render a list of components or elements using a loop, React needs a way to uniquely identify each item in the list. By assigning a unique "key" prop to each item, React can keep track of the items and efficiently update them when the list changes.

The main purpose of keys is to help React differentiate and distinguish elements from each other, increasing its performance when diffing between the virtual and real DOM.

We should use unique and descriptive keys and avoid using index or numbers as keys. A good rule of thumb is that elements inside the map() call need keys. Keys used within arrays should be unique among their siblings. However, they don’t need to be globally unique. We can use the same keys when we produce two different arrays.

Some examples of when we need keys in React:

- When rendering a list or array
- When using the map() function
- When using the useState() hook

10. **Can we use index as keys in React?**

While using the index as keys is technically possible, it's not recommended for dynamic lists that can change or be reordered. It's crucial to choose keys that are unique, stable, and associated with the data itself to ensure proper and efficient updating of the React component tree.

Using the index as keys can lead to issues in certain situations like:

- **Performance and Reordering**: If the list can be reordered or if items are frequently added or removed, using the index as keys can cause performance problems. When the order changes, React might unnecessarily re-render items or get confused about the actual changes, leading to unexpected behavior.

- **Stability of Keys**: Using the index as keys can lead to unstable keys if the order of items changes. For example, if an item is removed from the middle of the list, the indexes of the remaining items will shift, and their keys will change as well. This can cause React to re-render more elements than necessary.

- **Key Collisions**: If the data contains duplicate values, using the index as keys will not differentiate between the items, leading to key collisions and rendering issues.

11. What is props in React? Ways to use it?

Props in React are short for properties. They are data that is passed from a parent component to a child component. Props are used to make a component more dynamic and customizable, allowing data to flow down the component tree. Props are immutable, which means that they cannot be changed once they are passed to a child component.

There are several ways to use props in React. One way is to pass props to a component via HTML attributes. For example, we can add a “brand” attribute to a Car element like this:

```
const myElement = <Car brand="Ford" />;
```

The component receives the argument as a props object and we can use the brand attribute in the component like this:

```
function Car(props) {
        return <h2>I am a { props.brand }!</h2>;
};
```

Another way to use props is by destructuring them in the function signature of a functional component. For example, instead of writing

function Course (props) { return <div> {props.courseName}</div> };

we can write

function Course ({ courseName }) { return <div> {courseName}</div> };

12. What is a Config Driven UI ?

A Config Driven UI (User Interface) is an approach in software development where the user interface elements and behavior are determined and controlled by configuration files or data instead of being hard-coded in the application's source code. The goal of a Config Driven UI is to make the UI more flexible, configurable, and easy to modify without requiring code changes.

In a Config Driven UI, various aspects of the user interface, such as the layout, styling, content, components, and even the business logic, can be specified using configuration files or data files. These configuration files are typically in a structured format like JSON, YAML, XML, or any other format that can be easily parsed and interpreted by the application.

Config-driven UIs are becoming increasingly popular because they offer a number of advantages over traditional UIs.

- Flexibility: Config-driven UIs are very flexible because they can be easily customized by changing the configuration file. This makes them ideal for applications that need to be customized for different users or devices.
- Scalability: Config-driven UIs are also very scalable because they can be easily reused in different applications. This can save a lot of time and effort, especially for large applications.
- Maintainability: Config-driven UIs are also very maintainable because the code is separated from the appearance of the UI(separation of concerns). This makes it easier to change the code without affecting the appearance of the UI.

Here are some of the challenges of using config-driven UIs:

- Complexity: Config-driven UIs can be complex to implement, especially if the UI is complex.
- Performance: Config-driven UIs can be slower than traditional UIs, especially if the configuration file is large.
- Security: Config-driven UIs can be less secure than traditional UIs, if the configuration file is not properly secured.