● What is the difference between Named Export, Default export and \* as export?

In JavaScript (and particularly in ES6 modules), there are three primary ways to export and import modules: **Named Export**, **Default Export**, and **\* as Export**. Each has a different use case and syntax. Here’s a breakdown of how they differ:

1. **Named Export**

A **Named Export** allows you to export multiple values (variables, functions, classes, etc.) from a module, each with its own name. When importing, you must use the exact name of the exported value.

1. **Default Export**

A **Default Export** is used when you want to export a single value from a module. It can be a function, class, object, etc. A module can have only one default export.

1. **\* as Export (Namespace Import)**

The **\* as Export** syntax allows you to import all the named exports from a module as a single object (namespace). This is useful when you want to group all exports together under a single name.

● What are React Hooks?

React Hooks are a set of utility functions that allow you to use state and other React features in functional components. Introduced in React 16.8, hooks provide a way to use stateful logic and lifecycle features without writing class components. They offer a simpler and more concise way to manage state, side effects, and other component behaviors in functional components.

● Why do we need a useState Hook?

The useState hook in React is essential for managing state within functional components. Prior to the introduction of hooks in React 16.8, state management was only possible in class components. The useState hook enables functional components to have their own local state, bringing several benefits:

**Reasons Why useState Hook is Needed**

1. **State Management in Functional Components**:
   * **Purpose**: The primary reason for using useState is to allow functional components to manage and track state internally.
   * **Benefit**: Without useState, functional components would be stateless and could only rely on props for data, making them less versatile compared to class components.
2. **Simplicity and Readability**:
   * **Purpose**: useState simplifies the process of adding state to functional components, making the code more readable and easier to understand.
   * **Benefit**: Functional components with useState are less verbose and more straightforward than class components with state management.
3. **Declarative State Updates**:
   * **Purpose**: useState provides a way to declare and update state in a functional component using a simple API.
   * **Benefit**: The setState function returned by useState is declarative, meaning you describe what the new state should be, and React takes care of updating the state and re-rendering the component.
4. **Avoiding Boilerplate**:
   * **Purpose**: useState eliminates the need for the boilerplate code required in class components for managing state.
   * **Benefit**: Functional components with hooks are more concise and eliminate the need for lifecycle methods to manage state transitions.
5. **Encapsulation of State Logic**:
   * **Purpose**: useState allows you to encapsulate state logic within functional components, making it easier to manage and reason about state.
   * **Benefit**: This encapsulation can lead to better-organized code and easier maintenance, especially when combined with other hooks.
6. **Improved Component Composition**:
   * **Purpose**: Hooks, including useState, support better composition of component logic.
   * **Benefit**: You can create custom hooks to encapsulate and reuse stateful logic across multiple components, promoting code reusability and separation of concerns.