React Components

- In React, components are the building blocks of the user interface. They are JavaScript functions or classes that optionally accept inputs (known as "props") and return React elements that describe what should appear on the screen. There are two main types of components in React: functional components and class components.

- **Functional Components :** Functional components are simple JavaScript functions that accept props as an argument and return React elements. They are often preferred for their simplicity and ease of use, especially with the introduction of hooks like useState and useEffect.

```jsx
import React from 'react';

function Greeting(props) {
  return <h1>Hello, {props.name}</h1>;
}

export default Greeting;
```

With hooks, functional components can manage state and side effects:

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

function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  }, [count]);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}

export default Counter;
```

- **Class Components :** Class components are ES6 classes that extend from React.Component and must define a render method that returns a React element. They can have state and lifecycle methods, but with the advent of hooks, they are used less frequently in new React applications.

```jsx
import React, { Component } from 'react';

class Greeting extends Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}

export default Greeting;
```

This is final code which using React Component Card and returning it using App().

```jsx
function Card(props) {
  const { id, title, thumbnail, brand, price } = props;
  return (
    <div className="card" key={id}>
      <img src={thumbnail} alt={title} />
      <div className="card-content">
        <h3>{title}</h3>
        <p>{brand}</p>
        <p>
          <b>{price}</b>
        </p>
      </div>
    </div>
  );
}

function App() {
  const [products, setProducts] = useState([]); // State to hold the list of products

  // useEffect hook to fetch product data from the API when the component mounts
  useEffect(() => {
    fetch("https://dummyjson.com/products")
      .then((res) => res.json())
      .then((data) => setProducts(data.products)); // Update the state with the fetched products
  }, []); // Empty dependency array ensures this effect runs only once when the component mounts

  return (
    <>
      <div className="container">
        {/* Map through the products array and render a Card component for each product */}
        {products.map((product) => (
          <Card
            key={product.id}
            id={product.id}
            title={product.title}
            thumbnail={product.thumbnail}
            brand={product.brand}
            price={product.price}
          />
        ))}
      </div>
    </>
  );
}
```

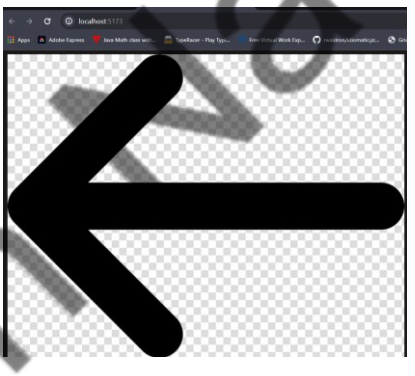
Using Images in React

- We can’t directly use images using it’s path in react like this:

```jsx
function App() {
  return (
    <>
      <img src="assets/images/arrow.png" alt="arrow" />
    </>
  );
}

export default App;
```

- Instead of this we import image using its path and then pass it as variable:

```jsx
import Arrow from "./assets/images/arrow.png";
function App() {
  return (
    <>
      <img src={Arrow} alt="arrow" />
    </>
  );
}
```



- **Naming and Placing of files should be done in this manner:**

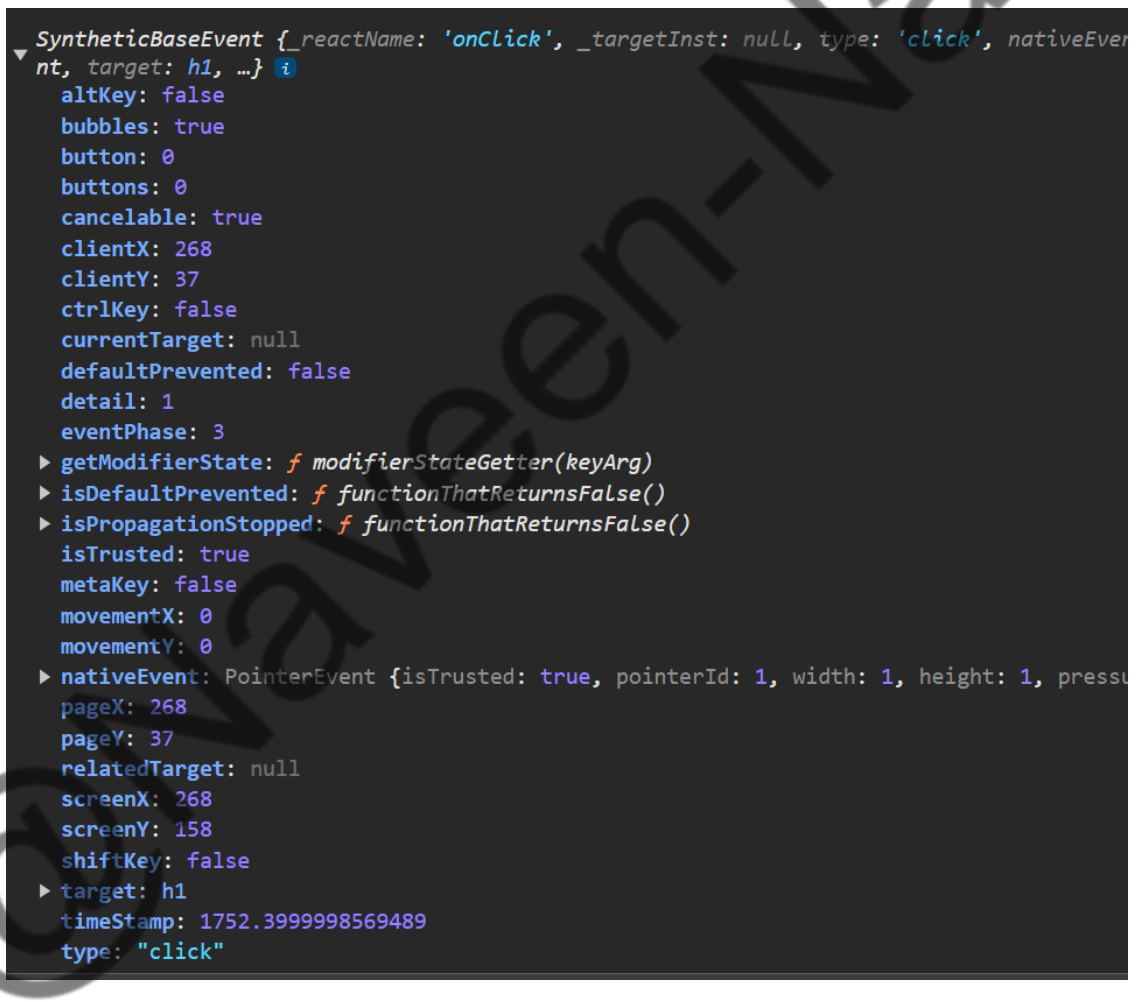
```
react-components/
|
├── node_modules/
|   └── ...
|
├── public/
|   └── ...
|
├── src/
|   ├── assets/
|   |   └── ... (images, fonts, etc.)
|   ├── components/
|   |   ├── Button/
|   |   |   ├── Button.jsx
|   |   |   └── Button.module.css (or Button.css)
|   |   └── ... (other components)
|   ├── styles/
|   |   ├── app.css
|   |   ├── main.css
|   |   └── ... (other global styles)
|   ├── App.jsx
|   └── main.jsx
|
├── .gitignore
├── index.html
├── package.json
├── vite.config.js
└── ...
```

Event Handling in React

- In react unlike normal java script we can use event handlers as function as shown:

```
<h1
  onClick={() => {
    console.log("Clicked");
  }}
>
  Click me
</h1>
```

- And the passing function accepts an event '***e'***, let's console log this event and we can see here that it's a synthetic event generated by React:



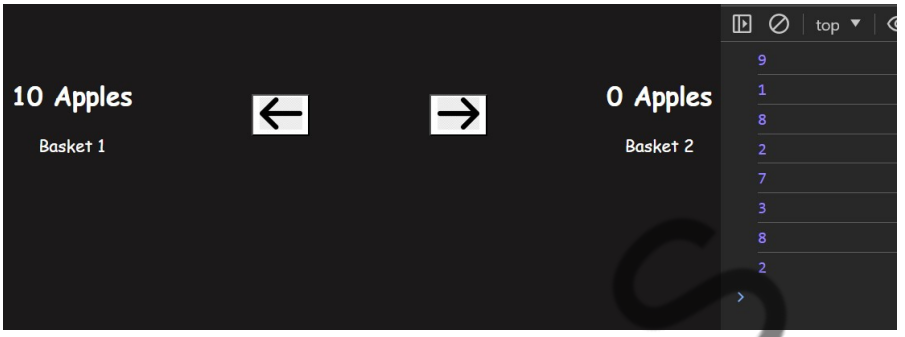
- ***Event can be only applied on html element NOT on component,*** *because React component is a function and if we pass event handler to it it'll take it as prop, yes we can pass event handler as prop in component and then use it in html body in the react component:*

```
return (
  <>
    <AppleCounter
      onClick={(e) => {
        console.log(e);
      }}
    />
  </>
```

```
function AppleCounter(props) {
  const { onClick } = props;
  return (
    <section onClick={onClick}>
      <AppleBasket count={10} basket={1} />
      <Button imageURL={LeftArrow} title={"Left Arrow"} />
      <Button imageURL={RightArrow} title={"Right Arrow"} />
      <AppleBasket count={0} basket={2} />
    </section>
  );
}
```

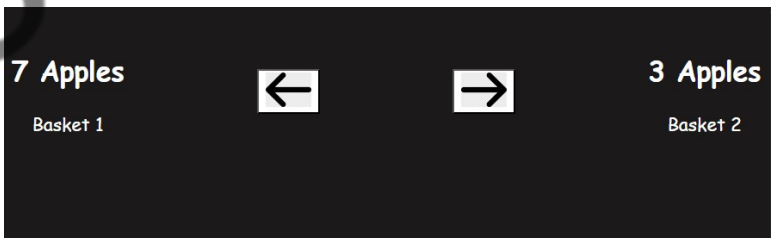
- Now in our Apple counter project if we are using **onClick** event handler to change the apple count then changes will not show on web browser, but we can see them in console:

```
function AppleCounter() {
  const totalAppleCount = 10;

  let rightAppleCount = 0;
  let leftAppleCount = totalAppleCount - rightAppleCount;

  const leftClickHandler = () => {
    leftAppleCount++;
    rightAppleCount--;
  };

  const rightClickHandler = () => {
    rightAppleCount++;
    leftAppleCount--;
  };

  return (
    <section>
      <AppleBasket count={leftAppleCount} basket={1} />
      <Button
        event={leftClickHandler}
```

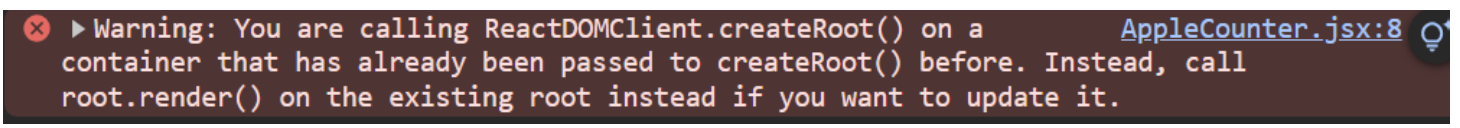


- This is because react it not rendering it by itself (there is another way in React to render things which called as **"states"**) but before using states let's do it using custom rendering to understand the functionality. In this program we are re-rendering AppleCounter by using **root.render(<Applecounter />)** in the event handler function (Note : don't forget to define ReactDOM in root variable first before using it):

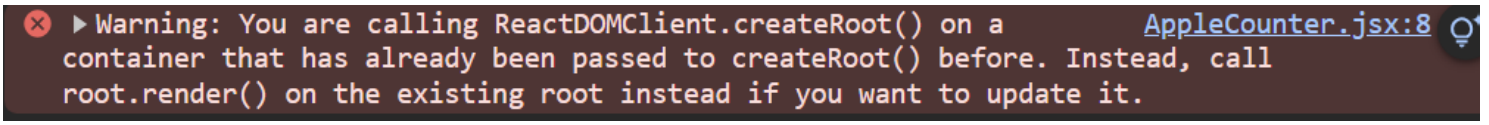
```
const root = ReactDOM.createRoot(document.getElementById("root"));
const totalAppleCount = 10;

let rightAppleCount = 0;
let leftAppleCount = totalAppleCount - rightAppleCount;

function AppleCounter() {
  const leftClickHandler = () => {
    leftAppleCount++;
    rightAppleCount--;
    console.log(leftAppleCount);
    console.log(rightAppleCount);
    root.render(<AppleCounter />);
  };

  const rightClickHandler = () => {
    rightAppleCount++;
    leftAppleCount--;
    console.log(leftAppleCount);
    console.log(rightAppleCount);
    root.render(<AppleCounter />);
  };
```



- And we can also see the changes in our website too, but in console there you see a warning that you can't do it in this way, so the official method is using states in React but this method is not wrong ultimately this is how React states work.



- Also React don't rendering our whole page it'll just rendering the relevant part, we can see it in inspect elements in the variables we can see the direct +- changes.

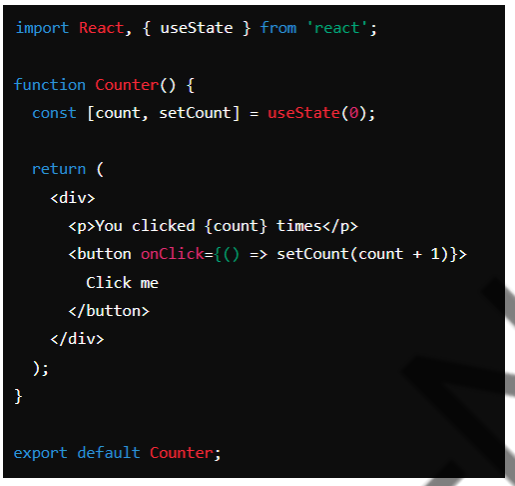


This warning message can also be removed if we just pass the root method created in **main.jsx** as props to components, then warning message also gets removed and our AppleCounter program will work fine without any error message and problem.

useState in React

`useState` is a fundamental hook in React that allows functional components to manage state. Before the introduction of hooks, state management was primarily the domain of class components. With useState, functional components can have their own state, making them more powerful and versatile.

Working of `useState` :



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

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}

export default Counter;
```

- **`useState(0)`** initializes a state variable count with the value 0.
- It returns an array with two elements: the current state (count) and a function to update that state (setCount).
- Calling setCount with a new value will update count and trigger a re-render of the component.

Internal Working of `useState` :

Internally, useState works through a combination of React's hooks mechanism and its component lifecycle management. Here's a high-level overview of what happens:

❖ Initialization:

- When a component with useState is first rendered, React creates a list of state values and updater functions for that component.
- useState initializes the state variable to the provided initial value (in the above example, 0).

❖ State Storage:

- React maintains an internal state array for each component instance. When useState is called, it looks at the current position in this state array to get the state value and updater function.
- This position is managed by a pointer known as a "hook index," which ensures that the correct state value is used for each hook call within a component.

❖ **State Update:**

- When the state updater function (setCount in the example) is called, it schedules a re-render of the component.
- During the re-render, React will again call useState, but this time it will use the current state value from the internal state array rather than the initial value.
- The internal state array and the hook index ensure that state values are consistent across renders.

➢ Here's a highly simplified pseudo-code to illustrate the concept:

```
// Internal state storage for hooks
let stateArray = [];
let currentIndex = 0;

function useState(initialValue) {
  const index = currentIndex; // Capture current index
  if (stateArray[index] === undefined) {
    // Initialize state value
    stateArray[index] = initialValue;
  }

  const setState = (newValue) => {
    stateArray[index] = newValue;
    // Trigger re-render (simplified)
    renderComponent();
  };

  currentIndex++; // Move to the next state
  return [stateArray[index], setState];
}

function renderComponent() {
  currentIndex = 0; // Reset hook index before each render
  // ...component rendering logic
}
```

- **Key Points:**
    - **Isolation:** Each component instance maintains its own state array, ensuring isolation of state.
    - **Order of Hooks:** The order in which hooks are called matters. Hooks must be called unconditionally in the same order on every render to maintain consistent state management.
    - **State Persistence:** Between renders, the state array keeps the current state values, allowing the component to access and update state across re-renders.

- In conclusion, useState brings stateful logic into functional components, enhancing their capability and simplifying the code structure by eliminating the need for class components for state management. The internal mechanism relies on maintaining an ordered list of state values and ensuring that hooks are called in the same order on every render.

- **Note :** If we pass same previous value to setCount() without updating it then useState don't re-render it, as with the help of memorization react knows that it the same value.

- **`useState` in React leverages closures to manage state.** Closures are an essential concept in JavaScript that allows functions to "remember" the environment in which they were created. In the context of useState, closures enable state management by capturing the current state and providing access to the state updater function.

- **How useState Uses Closures** :

  Here's a detailed breakdown of how useState uses closures:

    - **State Initialization and Closure Creation:** When useState is called, it initializes state and returns an array containing the current state value and a state updater function. This updater function forms a closure that retains access to the state variable and can update it when called.

    - **Maintaining State with Closures:** The state updater function retains a reference to the internal state array and the specific index of the state it manages. This reference is maintained through a closure, ensuring that each state updater function can update the correct piece of state.

❖ Internal Working with Closures:

Here's a simplified version to illustrate the closure mechanism:

```
// Simplified React-like implementation
let stateArray = [];
let currentIndex = 0;

function useState(initialValue) {
  const index = currentIndex; // Capture the current index
  if (stateArray[index] === undefined) {
    stateArray[index] = initialValue;
  }

  function setState(newValue) {
    stateArray[index] = newValue;
    renderComponent(); // Simplified re-rendering
  }

  currentIndex++;
  return [stateArray[index], setState];
}

function renderComponent() {
  currentIndex = 0; // Reset index before rendering
  // Render the component
}
// Example usage
function Counter() {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1); // `setCount` closure captures `count` and `setCount`
  };

  console.log(`Count: ${count}`); // Render output simulation
  return { increment };
}

// Simulating component usage
const counter = Counter();
counter.increment(); // Count: 1
counter.increment(); // Count: 2
```

❖ Key Points :

- **Closure Capture:** setState function captures the index and stateArray through a closure. When setState is called, it updates the correct state value based on the captured index.

- **Re-render and State Persistence:** During re-renders, useState maintains state through closures. The setState function created during the initial render retains its ability to update the correct state variable due to the closure.

- **Ensuring Consistent State Management:** Closures help useState ensure that state variables and their updater functions are correctly associated, even across multiple re-renders.

Using callback function in setState()

- In React, the setCount function returned by the useState hook can accept either a new state value or a callback function. This callback function is used to update the state based on the previous state. The concept is particularly useful when the new state depends on the previous state, ensuring that state updates are accurate even if there are multiple updates in quick succession.

- **Why Use a Callback Function in setState?** When updating state, using the current state directly can lead to problems in scenarios where updates are batched or when multiple updates are triggered at once. React's state updates are asynchronous, meaning that setState (or setCount in this context) doesn't immediately update the state. Instead, it schedules an update, and the actual update happens later. Using a callback function ensures that the state update always has the latest state.

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

function Counter() {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1);
    setCount(count + 1); // This won't work as expected
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}

export default Counter;
```

- In the above example, clicking the button will not increment the count by 2 as expected because both setCount calls use the same stale count value due to React's asynchronous state updates.

- **Why? :** Because when we call setCount in React, the state update is scheduled to happen, but it doesn't take effect immediately. React batches state updates for performance reasons, which means it processes them together during the next render cycle. This behavior is what can cause issues when multiple state updates rely on the current state value within the same event handler.
- That’s why we use callback functions, as it uses prev value instead of curr count.

❖ Why Does This Happen?

- **State Scheduling:** When we call setCount(count + 1), React schedules an update to increment count by 1. However, this update is not applied immediately. Instead, React marks the component to be re-rendered with the new state.

- **Asynchronous Behavior:** The second setCount(count + 1) call is still using the initial count value (before any update has been applied). Both calls essentially end up scheduling the same state update: incrementing the initial count by 1.

❖ Internal Process:

- First setCount(count + 1): Schedules count to be updated from 0 to 1.

- Second setCount(count + 1): Also schedules count to be updated from 0 to 1.

React then processes these state updates together, resulting in count being set to 1, not 2.

❖ **Solution with Callback:** Using a callback function ensures that each state update is based on the latest state value at the time of the update, not the value at the time the setCount call was made.

```
// setCount(count + 1);
// setCount(count + 1); // This is will not work correctly
setCount((prev) => prev + 1);
setCount((prev) => prev + 1);
```

❖ How It Works:

- First setCount: The callback receives the current count value (initially 0) and returns prevCount + 1, scheduling an update to count = 1.

- Second setCount: By the time the second callback is executed, React ensures it gets the updated count value (which is now 1) and increments it to 2.

❖ **Internal Process with Callback:** First setCount(prevCount => prevCount + 1): Schedules count to be updated from 0 to 1. Second setCount(prevCount => prevCount + 1): Schedules count to be updated from 1 to 2.

Fragment in React

- A React Fragment is a component provided by React that lets us group a list of children without adding extra nodes to the DOM. This is particularly useful when rendering multiple elements from a component without introducing unnecessary wrapper elements, which can interfere with styles and layout.

```jsx
function MyComponent() {
  return (
    <React.Fragment>
      <h1>Hello, world!</h1>
      <p>This is a paragraph.</p>
    </React.Fragment>
  );
}
```

OR

```jsx
function MyComponent() {
  return (
    <>
      <h1>Hello, world!</h1>
      <p>This is a paragraph.</p>
    </>
  );
}
```

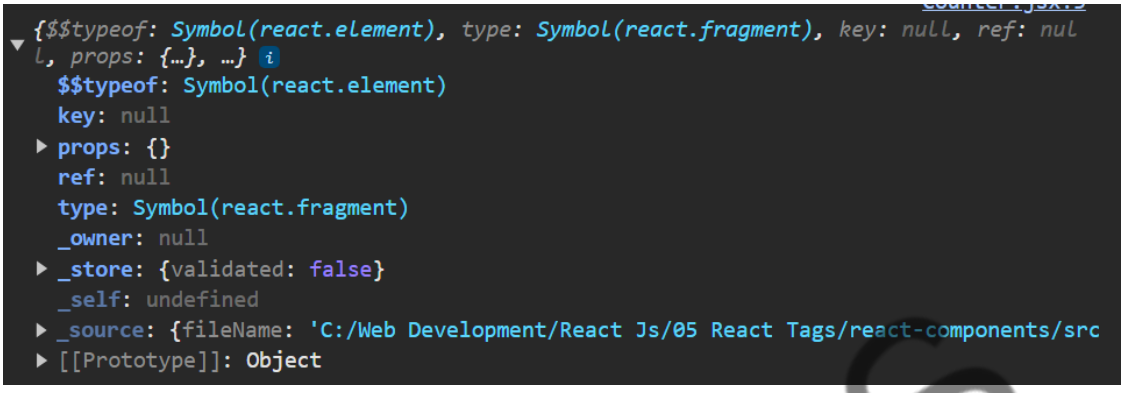
- **When to Use React Fragments:**
  - **Avoid Extra DOM Elements:** When you want to return multiple elements from a component without adding extra nodes to the DOM.
  - **Better Performance:** Reduces unnecessary complexity and improves performance by avoiding additional wrapping elements.
  - **Preserve HTML Structure:** When you need to keep your HTML structure clean and semantic without additional divs or spans.
- **Let's console log Fragment for better understanding:**
  - **console.log(Fragment):** When we console log it will show this, here Symbol is a primitive data type introduced in ECMAScript 2015 (ES6). Symbols are unique and immutable identifiers, often used to create unique keys for object properties that do not conflict with other property keys.
    In the case of React, the React team uses symbols to create unique identifiers for its internal use, ensuring that certain aspects of React (like fragments) are not confused with any other part of the code or libraries.

```
Symbol(react.fragment)
```

  - **console.log(typeof Fragment):**

```
Symbol(react.fragment)
Type is : symbol
```

- **Console.log(<> </>):** This will show a React element of type symbol, as ultimately fragments creating a React element.



❖ Can we use styling on Fragment?

➢ **No,** we cannot apply styles directly to React Fragments because they do not render any actual DOM elements. React Fragments are essentially invisible wrappers used to group multiple elements without adding extra nodes to the DOM. Since there's no corresponding DOM element, there is nothing to apply styles to.

Child prop in React

- In React, the children prop is a special prop that allows us to pass components or elements as children to other components. This is particularly useful for creating wrapper components or layout components that need to display or manipulate the nested child elements passed to them.

- Let's understand how to use it:
  - Take our Apple Counter project as example in AppleCounter() component we're using self closing button tag like this:

```
<Button
  event={rightClickHandler}
  imageURL={RightArrow}
  title={"Right Arrow"}
/>
```

  - But we can also use Button tag with separate closing tag to pass child arguments, like this:

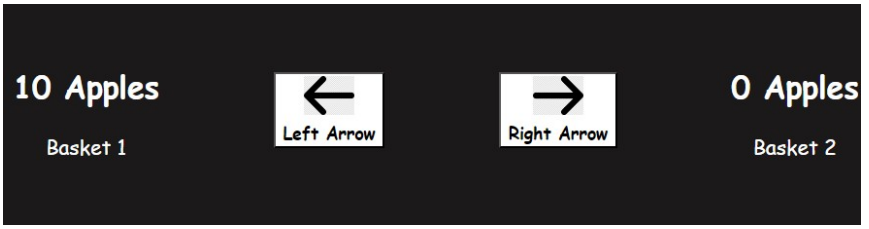
```
<Button
  event={leftClickHandler}
  imageURL={LeftArrow}
  title={"Left Arrow"}
>
  children
</Button>
```

  - Now we can pass anything as child, for eg: tags, array, function, objects, another component, React element, etc.

  - Also we can pass child prop to component and access it

```
function Button(props) {
  const { imageURL, title, event, children } = props;
  console.log(children);
```

  - For the above example we used it as to give name to button with image also.



  - So this is how we can use child prop in react, for more info read out this : [click here]

Conditional Rendering in React

- Conditional rendering in React refers to the process of dynamically displaying or hiding components or elements based on certain conditions. This is similar to conditionally rendering content in JavaScript, where different content is shown or hidden based on certain conditions.

- There are several common methods to implement conditional rendering in React:

  - **Using if Statements:** We can use traditional if statements to determine what component or element should be rendered.

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <h1>Welcome back!</h1>;
  }
  return <h1>Please sign up.</h1>;
}
```

  - **Using Element Variables:** Store elements in variables and conditionally assign them before rendering.

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  let greetingMessage;
  if (isLoggedIn) {
    greetingMessage = <h1>Welcome back!</h1>;
  } else {
    greetingMessage = <h1>Please sign up.</h1>;
  }
  return greetingMessage;
}
```

  - **Using Ternary Operators:** This is a concise way to handle simple conditional rendering.

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  return (
    <div>
      {isLoggedIn ? <h1>Welcome back!</h1> : <h1>Please sign up.</h1>}
    </div>
  );
}
```

- **Using Logical `&&` Operator:** Render a component or element only if the condition is true.

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
        </h2>
      }
    </div>
  );
}
```

- **Using switch Statements:** For more complex conditions, switch statements can be useful.

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}

function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  switch (isLoggedIn) {
    case true:
      return <UserGreeting />;
    case false:
      return <GuestGreeting />;
    default:
      return <GuestGreeting />;
  }
}
```

CSS as Module in React

- In React, the practice of naming CSS files with a **filename.module.css** extension and importing them as CSS modules is a way to ensure that the styles are scoped locally to the component, preventing CSS conflicts that are common in larger applications.

- **Why Use components.module.css in React:**
  - **Local Scope:** CSS modules automatically scope the CSS by generating unique class names. This prevents the styles from one component from accidentally affecting other components.
  - **Maintainability:** Scoped styles make it easier to manage and maintain the styles as your application grows.
  - **Encapsulation:** It aligns with the React philosophy of component-based development, where each component encapsulates its own styles.

- **How to Use CSS Modules in React:**
  - **Naming Convention:** Name your CSS file with the .module.css suffix (e.g., components.module.css).
  - **Importing the CSS Module:** Import the CSS module into your React component using a specific syntax that ensures the styles are scoped correctly.

```
import styles from './components.module.css';
```

  - **Using the Scoped Styles:** Apply the styles to your JSX elements using the imported styles object.

```
import React from 'react';
import styles from './components.module.css';

const MyComponent = () => {
  return (
    <div className={styles.myClass}>
      This div has locally scoped styles!
    </div>
  );
};

export default MyComponent;
```

❖ How React CSS Modules Prevent CSS Conflicts:

- **Unique Class Names:** When you import a CSS module, the class names defined in the CSS file are automatically transformed into unique identifiers. For example, a class named .myClass in `components.module.css` might be transformed into something like `components_module__myClass__3aGfK`.

- **No Global Scope:** Since the generated class names are unique, they do not interfere with other class names elsewhere in the application, effectively isolating the styles.

- Example: Given a CSS module file `components.module.css`:

```css
.myClass {
  color: blue;
}
```

When used in component:

```jsx
import styles from './components.module.css';

const MyComponent = () => {
  return <div className={styles.myClass}>Styled with CSS Module</div>;
};
```

The resulting HTML might look something like this:

```html
<div class="components_module__myClass__3aGfK">Styled with CSS Module</div>
```

This ensures that even if there is another .myClass defined elsewhere in the application, it will not conflict with the .myClass used in MyComponent because the class names are scoped locally by CSS modules.