Learning JSX

What is JSX?

JSX stands for JavaScript XML. It is syntactic sugar for creating React Elements. It is a syntax extension to JavaScript. It is used with React to describe what the UI should look like. JSX may remind you of a template language, but it comes with the full power of JavaScript. It produces React "elements." Example:

```
const element = <h1>Hello world!</h1>;
```

Why JSX

JSX allows us to write HTML elements in JavaScript and place them in the DOM without any `createElement()` and `appendChild()` methods. JSX converts HTML tags into react elements. You are not required to use JSX, but JSX makes it easier to write React applications. Example:

Without JSX

```
const myElement = React.createElement('h1', {}, 'I do not use JSX!');
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

With JSX

```
const myElement = <h1>I Love JSX!</h1>;
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

React embraces the fact that rendering logic should be coupled with UI logic. It is helpful as a visual aid when working with UI inside the javascript. React separates concerns instead of separating technologies. It allows React to show more valuable errors and warning messages.

Babel is a JavaScript Compiler that allows us to use future JavaScript in today's browsers. Simply put, it can convert the latest version of JavaScript code into the one the browser understands.
Babel can convert JSX syntax. It is therefore used to convert JSX expressions into JavaScript code browsers can understand.
You can try it yourself. Here is the link.

React Fragments

It is a common pattern in React that a component returns multiple elements. Fragments let you group a list of children without adding extra nodes to the DOM. We know that we use the render method inside a component whenever we want to render something to the screen. We may generate single or multiple elements. However, rendering multiple elements will require a ‘div’ tag around the content as the render method will only render a single root node inside it at a time. Example:

```
function App() {
    return (
      <div>
        <h2>Hello</h2>
        <p>How are you doing?</p>
      </div>
    );
}
```

Reason to use Fragments: As we saw in the above code, when we are trying to render more than one root element, we have to put the entire content inside the ‘div’ tag, which is not a good approach because no one wants to include an extra div element if that is not required in the code. Hence

Fragments were introduced, and we use them instead of the extraneous ‘div’ tag. Example:

```
function App() {
    return (
      <React.Fragment>
        <h2>Hello</h2>
        <p>How are you doing?</p>
      </React.Fragment>
    );
}
```

Shorthand Fragment: The output of the first code and the code above is the same, but the main reason for using it is that it is a tiny bit faster when compared to the one with the ‘div’ tag inside it, as we didn’t create any DOM nodes. Also, it takes less memory. Another shorthand also exists for the above method in which we use ‘<>’ and ‘</>’ instead of the ‘React.Fragment’. Example:

```
function App() {
    return (
      <>
        <h2>Hello</h2>
        <p>How are you doing?</p>
      </>
    );
}
```

JSX Expression

With JSX, you can write expressions inside curly braces { }. JSX Expressions, written inside curly brackets, allow only things that evaluate some value like string, number, array map method, etc. The expression can be a React variable, property, or any other valid JavaScript expression. JSX will execute the expression and return the result.

Example:

```
function App() {
  var name = "John Doe";
  let age = 4;
  const header = <h2>This is Header</h2>;

  return (
    <div>
      {header}
      <p>My name is {name}.</p>
      <p>My age is {age}.</p>
    </div>
  );
}
```

Returning two items at a time is invalid in Javascript. Since JSX is still JavaScript, its return can only handle one expression therefore, to return more than one element, we need to wrap it in another (an outer element), which most commonly will be a .

For eg, The code below will throw an error because JSX expressions must have only one parent element.

```
  function App() {
    return (
      <div>Hello</div>
      <div>World</div>
    )
  }
```

Comments in JSX

To add JavaScript code inside JSX, we need to write it in curly brackets. To add a comment for that code, then you have to wrap that code in JSX expression syntax inside the /* and */ comment symbols like this:

```
{/* <p>This is some text</p> */}
```

Null/Undefines/Boolean in JSX

JSX ignores null, undefined, and Booleans (false, true). They don't render as JSX is syntactic sugar for React.createElement(component, props, ...children).

For Eg: These JSX expressions will all render the same thing:

```
<div />
<div></div>
<div>{false}</div>
<div>{null}</div>
<div>{undefined}</div>
<div>{true}</div>
```

If you want a value like false, true, null, or undefined to appear in the output, you have to convert it to a string first:

```
<div>
  My JavaScript variable is {String(myVariable)}.
</div>
```

Functions in JSX

A function can be defined and used by the expression. The component takes the function's output to produce content. We can simply define a function inside our component, and we can call it inside the return() method. We can invoke the function by adding parentheses () at the end.

For Eg:

```
const App = () => {
  const a = 4;
  const b = 6;
  const sum = (a, b) => a + b;
  return (
      <h2>Sum of {a} and {b} is: {sum(a, b)}</h2>
  );
};
```

Arrays in JSX

Arrays can be rendered inside the JSX Expressions easily using curly braces similar to any variable. For example:

```
const arr = [1, 2, 3, 4, 5];
const App = () => {
  return ( <h2>Array is: {arr}</h2> );
};
```

Output: Array is: 12345;

Iterating over the array

We usually use the map function to iterate through the array in React. The map is a JavaScript function that can be called on any array. With the map function, we map every element of the array to the custom components in a single line of code. That means there is no need to call components and their props array elements repeatedly.

The `.map()` method allows you to run a function on each item in the array, **returning a new array**. In React, map() can be used to generate lists and render a list of data to the DOM. To use the map() function, we attach it to an array we want to iterate over.

Example:

```
const myArray = ['apple', 'banana', 'orange'];
const myList =
myArray.map((item, index) => <p key={index}>{item}</p>)
```

Note: Keys allow React to keep track of elements. This way, if an item is updated or removed, only that item will be re-rendered instead of the entire list. Keys need to be unique to each sibling.

Generally, the key should be a unique ID assigned to each item. As a last resort, you can use the array index as a key.

Filter()

The filter() method is an iterative method. It calls a provided callbackFn function once for each element in an array and constructs a new array of all the values for which callbackFn returns a truthy value. Array elements that do not pass the callbackFn test are not included in the new array.
The JavaScript Array filter() Method is used to create a new array from a given array consisting of only those elements from the given array which satisfy a condition set by the argument method.

Example:

```
function isEven(value) {
    return value % 2 == 0;
  }
  var originalArr = [11, 98, 31, 23, 944];
  var newArr = originalArr.filter(isEven);
  console.log(newArr);
```

Output: [98,944]

The function passed in the filter function checks whether each element is even or odd. The filter function then returns only the truthy element; hence, the new array returned consists of only 2 elements, 98 and 944.

Objects in JSX

Objects can be defined as an unordered collection of data in the form of “key: value” pairs. JSX can’t render objects. React has no way to tell what to render when provided with an object, thus the Invariant Violation error pops up when attempting so. The error "Objects are not valid as a React child" is standard, and the solution is to manipulate the object so that it becomes a valid element.

And this is excellent because it now remains in the developer's hand to decide how to present the data in the object in its application.

Example:

```
const App = () =>
const myVariable = {
   productName: "Watermelon",
   price: 12
};
return (
    <div>
        {myVariable.productName} : $(myVariable.price}
    </div>
);
}
```

And the above code would render a div with the content: Watermelon: $12

Tables in JSX

The tables are created using the <table> tag in which the <tr> tag is used to create table rows, and the <td> tag is used to create data cells. The elements under <td> are regular and left aligned by default. Here, the border is an attribute of <table> tag, and it is used to put a border across all the cells. If you do not need a border, then you can use border = "0".

Table Heading

Table heading can be defined using <th> tag. This tag will be put to replace <td> tag, which is used to represent actual data cells. Normally you will put your top row as a table heading as shown below; otherwise, you can use <th> element in any row. Headings, which are defined in <th> tag are centred and bold by default.

Colspan and Rowspan Attributes

You can use the `colSpan` attribute if you want to merge two or more columns into a single column. Similarly, you can use `rowSpan` if you want to merge two or more rows. Example:

```
<table border = "1">
   <tr>
      <th>Column 1</th>
      <th>Column 2</th>
      <th>Column 3</th>
   </tr>
   <tr>
      <td rowSpan = "2">Row 1 Cell 1</td>
      <td>Row 1 Cell 2</td>
      <td>Row 1 Cell 3</td>
   </tr>
   <tr>
      <td>Row 2 Cell 2</td>
      <td>Row 2 Cell 3</td>
   </tr>
   <tr>
      <td colSpan = "3">Row 3 Cell 1</td>
   </tr>
</table>
```

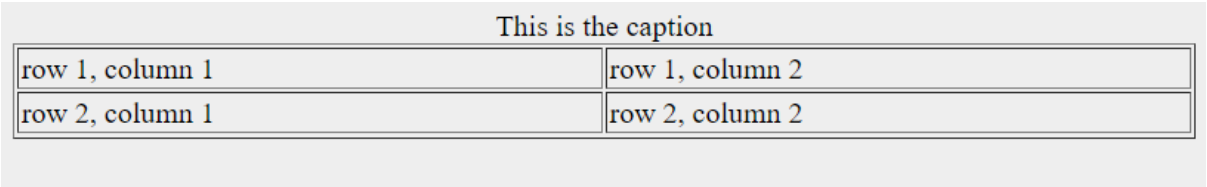
Column 1	Column 2	Column 3
Row 1 Cell 1	Row 1 Cell 2	Row 1 Cell 3
	Row 2 Cell 2	Row 2 Cell 3
Row 3 Cell 1		

Table Caption

The caption tag will serve as a title or explanation for the table and will show up at the top. It can be just before the table's 1st <tr> element.

Example:

```
<table border="1">
    <caption>This is the caption</caption>
    <tr>
        <td>row 1, column 1</td>
        <td>row 1, column 2</td>
    </tr>
    <tr>
        <td>row 2, column 1</td>
        <td>row 2, column 2</td>
    </tr>
</table>
```



<thead>, <tbody>, <tfoot>

The `<thead>` tag is used to group header content in a table. The `<tbody>` tag is used to group the body content in a table. The `<tfoot>` tag is used to group footer content in a table. These are the semantic tags that not only provide meaning to the elements but also have some other useful functions as well. Browsers can use these elements to enable scrolling of the table body independently of the header and footer. Also, when printing a large table that spans multiple pages, these elements can enable the table header and footer to be printed at the top and bottom of each page.

Note: You can use one table inside another table. Not only tables, but you can also use almost all the tags inside table data tag <td>. This is called a nested table.

Conditional Rendering

Conditional rendering in React works the same way conditions work in JavaScript. We can use JavaScript operators like if - else or the conditional operator (ternary operator) or AND operator or OR operator in JSX.

Conditional rendering with if else statement

We can use the if-else statements to render a JSX expression on the basis of some conditions. Note that if-else statements can't return additional JSX elements apart from the elements which are inside the if-else statements. This is because the return keyword inside the App component will return the elements which are after the return statement. So, if any condition inside the if-else statement is true, then just the elements inside that condition gets rendered and it doesn't even check the rest of the conditions or elements which are put outside that truthy condition. Example:

```
const App = () => {
    const email = "demo@codingninjas.com";
    const password = "demo";
    if (email == "demo@codingninjas.com") {
        if (password == "demo") {
            return <h1>User is an employee.</h1>;
        } else {
            return <h1>Incorrect password</h1>;
        }
    } else {
        return <h1>User is a student.</h1>;
    }
};
```

Conditional rendering with the ternary operator

The conditional (ternary) operator is the only JavaScript operator that takes three operands: a condition followed by a question mark (?), then an expression to execute if the condition is truthy followed by a colon (:), and

finally, the expression to execute if the condition is falsy. This operator is frequently used as an alternative to an if...else statement.

Example:

```
const App = () => {
    const email = "demo@codingninjas.com";
    const password = "demo";
    return (
        <>
            {
                (email == "demo@codingninjas.com")
                    ? (password == "demo")
                        ? <h1>User is an employee.</h1>
                        : <h1>Incorrect password.</h1>
                    : <h1>User is a student.</h1>
            }
        </>
    )
};
```

Conditional rendering with AND operator

Another way to conditionally render a React component is by using the && operator. It returns the first falsy and last truthy value.

Example:

```
const App = () => {
  const email = "demo@codingninjas.com";
  const password = "demo";
  return (
    <>
        {
          (email == "demo@codingninjas.com" && password == "demo")
          && (<h1>User is an employee.</h1>)
        }
        {
          (email == "demo@codingninjas.com" && password != "demo")
          && (<h1>Incorrect password.</h1>)
```

```
        }
        {email != "demo@codingninjas.com"
        && <h1>User is a student.</h1>}
    </>
  );
};
```

Conditional rendering with OR operator

We can also render a React component by using the || operator. It returns last falsy and first truthy value.

Example:

```
const App = () => {
    const email = "demo@codingninjas.com";
    return (
        <>
            {( email == "demo@codingninjas.com" ||
            email == "demo2@codingninjas.com") && (
            <h1>User is an employee.</h1>
            )}
        </>
    );
};
```

Summarising it

Let's summarise what we have learned in this module:

- Learned about the JSX.
- Learned about React Fragments and its shortcut.
- Learned about JSX Expressions.
- Learned about how arrays and lists are rendered in React.
- Learned about how to use objects in JSX.
- Learned about how to use tables in JSX.
- Learned about different types of conditional rendering in JSX.

Additional References (if you want to explore more):

- More information JSX Expressions: Link
- To apply React Developer tools: Link
- To read more about the array iteration methods: Link
- More information on Tables: Link
- Advanced information on Tables: Link
- You can read about nullish coalescing operator: Link
- Logical operators: Link