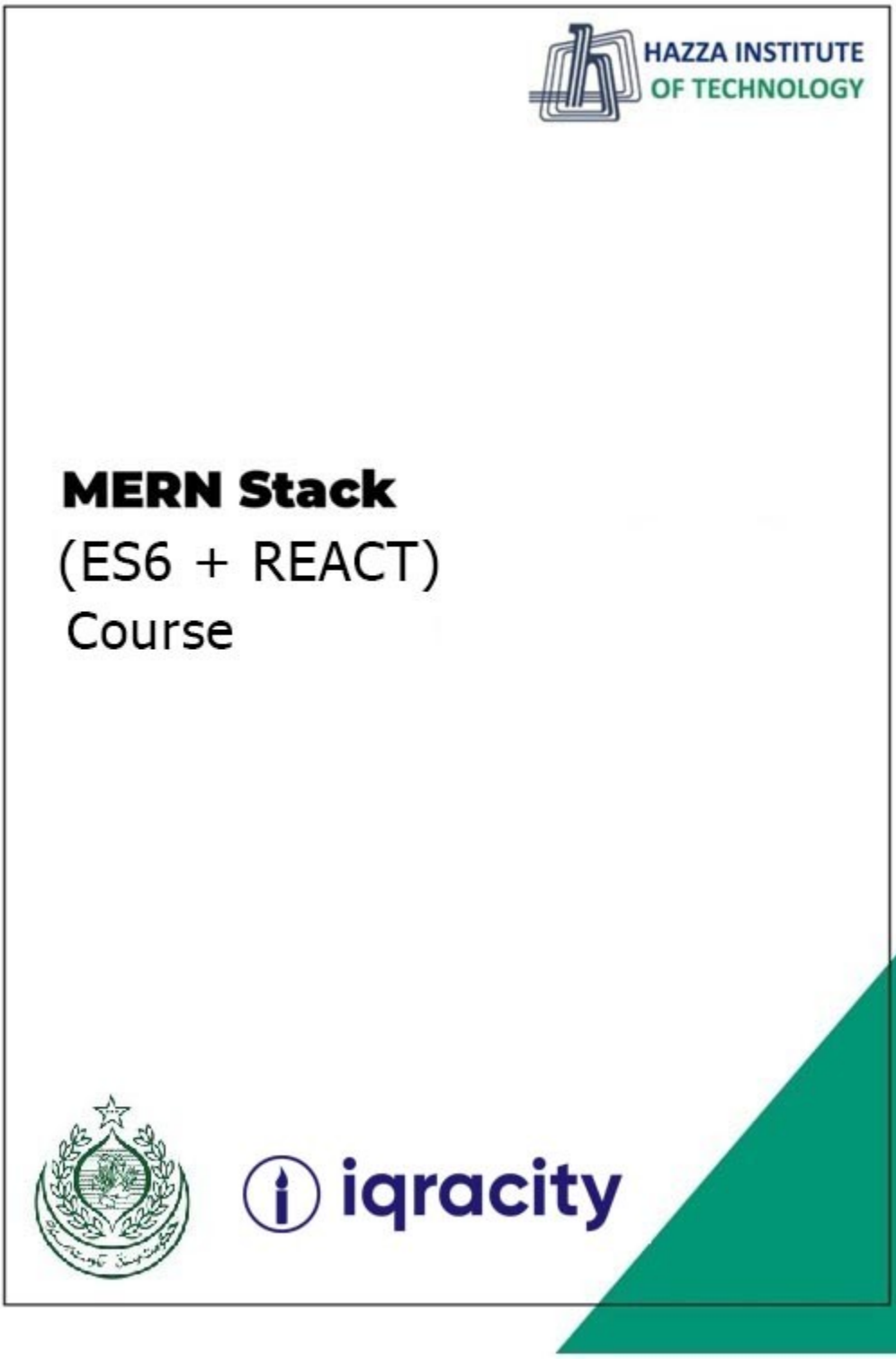

Lab 14

Total Time:
3 hours

Pre-Lab Activities:

- No Pre-Lab Activity

Learning Outcomes:

- Perform the concepts of what React is and how it works.

Lab Tasks:

- React JSX
- Coding JSX
- Expressions in JSX
- Inserting large blocks of HTML
- Attribute Class
- React Components
- Rendering Components
- Using State Objects

Student Activities:

- Explore React JSX
- Explore Coding JSX
- Explore Expressions in JSX
- Explore Inserting large blocks of HTML
- Explore Attribute Class
- Explore React Components
- Explore Rendering Components
- Explore Using State Objects

Lab Solution

React JSX:

What is JSX?

JSX stands for JavaScript XML.

JSX allows us to write HTML in React.

JSX makes it easier to write and add HTML in React.

Coding JSX:

SX allows us to write HTML elements in JavaScript and place them in the DOM without any `createElement()` and/or `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.

Here are two examples. The first uses JSX and the second does not:

Example 1:

JSX:

```
const myElement =<h1>I Love JSX!</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);
```

Example 2:

JSX:

```
const myElement = React.createElement('h1',{},'I do not use
JSX!');

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);
```

As you can see in the first example, JSX allows us to write HTML directly within the JavaScript code.

JSX is an extension of the JavaScript language based on ES6, and is translated into regular JavaScript at runtime.

Expressions in JSX:

With JSX you can write expressions inside curly braces `{ }`.

The expression can be a React variable, or property, or any other valid JavaScript expression. JSX will execute the expression and return the result:

Example:

Execute the expression `5 + 5`:

```
const myElement =<h1>React is {5+5} times better with JSX</h1>;
```

Inserting a Large Block of HTML:

Create a list with three list items:

```
const myElement =(

<ul>

<li>Apples</li>

<li>Bananas</li>

<li>Cherries</li>

</ul>

);
```

Example:

Wrap two paragraphs inside one DIV element:

```
const myElement = (

<div>

<p>I am a paragraph.</p>

<p>I am a paragraph too.</p>

</div>

);
```

JSX will throw an error if the HTML is not correct, or if the HTML misses a parent element.

Alternatively, you can use a "fragment" to wrap multiple lines. This will prevent unnecessarily adding extra nodes to the DOM.

A fragment looks like an empty HTML tag: <><>.

Example:

Wrap two paragraphs inside a fragment:

```
const myElement =(

<>

<p>I am a paragraph.</p>

<p>I am a paragraph too.</p>

</>

);
```

Example:

Close empty elements with `/>`

```
const myElement =<input type="text"/>;
```

JSX will throw an error if the HTML is not properly closed.

Attribute class = className:

The `class`attribute is a much used attribute in HTML, but since JSX is rendered as JavaScript, and the `class`keyword is a reserved word in JavaScript, you are not allowed to use it in JSX.

Use attribute className instead.

JSX solved this by using `className`instead. When JSX is rendered, it translates `className`attributes into `class`attributes.

Example:

Use attribute`className`instead of`class`in JSX:

```
const myElement =<h1 className="myclass">Hello World</h1>;
```

Conditions - if statements:

React supports `if`statements, but not *inside* JSX.

To be able to use conditional statements in JSX, you should put the `if`statements outside of the JSX, or you could use a ternary expression instead:

Option 1:

Write `if`statements outside of the JSX code:

Write "Hello" if`x`is less than 10, otherwise "Goodbye":

```
const x =5;

let text ="Goodbye";

if (x <10){

  text ="Hello";

}

const myElement =<h1>{text}</h1>;
```

Option 2:

Use ternary expressions instead:

Example:

Write "Hello" if xis less than 10, otherwise "Goodbye":

```
const x =5;

const myElement =<h1>{(x)<10?"Hello":"Goodbye"}</h1>;
```

Note *that in order to embed a JavaScript expression inside JSX, the JavaScript must be wrapped with curly braces,* ***{}****.*

React Components:

Components are like functions that return HTML elements.

Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML.

Components come in two types, Class components and Function components, in this tutorial we will concentrate on Function components.

In older React code bases, you may find Class components primarily used. It is now suggested to use Function components along with Hooks, which were added in React 16.8. There is an optional section on Class components for your reference.

Create Your First Component:

When creating a React component, the component's name *MUST* start with an uppercase letter.

Class Component:

A class component must include the `extends React.Component` statement. This statement creates an inheritance to React.Component, and gives your component access to React.Component's functions.

The component also requires a `render()` method, this method returns HTML.

Example:

Create a Class component called `Car`

```
class Car extends React.Component{

render(){

    return <h2>Hi, I am a Car!</h2>;

}

}
```

Example:

Create a Function component called `Car`

```
function Car(){

  return <h2>Hi, I am a Car!</h2>;

}
```

Rendering a Component:

Now your React application has a component called Car, which returns an `<h2>` element.

To use this component in your application, use similar syntax as normal HTML: `<Car />`

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car/>);
```

Props:

Components can be passed as `props`, which stands for properties.

Props are like function arguments, and you send them into the component as attributes.

You will learn more about `props`in the next chapter.

Example:

Use an attribute to pass a color to the Car component, and use it in the render() function:

```
function Car(props){

  return <h2>I am a {props.color} Car!</h2>;

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Carcolor="red"/>);
```

Components in Components:

We can refer to components inside other components:

Example:

Use the Car component inside the Garage component:

```
function Car(){
  return <h2>I am a Car!</h2>;
}
function Garage(){
  return (
<>
<h1>Who lives in my Garage?</h1>
<Car/>
</>
);
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage/>);
```

Components in Files:

React is all about re-using code, and it is recommended to split your components into separate files.

To do that, create a new file with a `.js` file extension and put the code inside it:

Note that the filename must start with an uppercase character.

Example:

This is the new file, we named it "Car.js":

```
function Car(){

  return <h2>Hi, I am a Car!</h2>;

}

export default Car;
```

To be able to use the Car component, you have to import the file in your application.

Example:

Now we import the "Car.js" file in the application, and we can use the `Car`component as if it was created here.

```
import React from 'react';

import ReactDOM from 'react-dom/client';

import Car from './Car.js';

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car/>);
```

React Class Components:

Before React 16.8, Class components were the only way to track state and lifecycle on a React component. Function components were considered "state-less".

With the addition of Hooks, Function components are now almost equivalent to Class components. The differences are so minor that you will probably never need to use a Class component in React.

Even though Function components are preferred, there are no current plans on removing Class components from React.

This section will give you an overview of how to use Class components in React.

React Components:

Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML via a render() function.

Components come in two types, Class components and Function components, in this chapter you will learn about Class components.

Create a Class Component:

When creating a React component, the component's name must start with an upper case letter.

The component has to include the `extends React.Component` statement, this statement creates an inheritance to React.Component, and gives your component access to React.Component's functions.

The component also requires a `render()` method, this method returns HTML.

Example

Create a Class component called `Car`

```
class Car extends React.Component{

render(){

    return <h2>Hi, I am a Car!</h2>;

}

}
```

Now your React application has a component called Car, which returns a `<h2>` element.

To use this component in your application, use similar syntax as normal HTML: `<Car />`

Example:

Display the `Car`component in the "root" element:

```
const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car/>);
```

Component Constructor:

If there is a `constructor()` function in your component, this function will be called when the component gets initiated.

The constructor function is where you initiate the component's properties.

In React, component properties should be kept in an object called `state`.

You will learn more about `state`later in this tutorial.

The constructor function is also where you honor the inheritance of the parent component by including the `super()` statement, which executes the parent component's constructor function, and your component has access to all the functions of the parent component (`React.Component`).

Example:

Create a constructor function in the Car component, and add a color property:

```
class Car extends React.Component{

constructor(){

    super();

    this.state ={color:"red"};

}

render(){

    return <h2>I am a Car!</h2>;

}

}
```

Use the color property in the render() function:

Example:

```
class Car extends React.Component{

constructor(){

    super();

    this.state ={color:"red"};

}

render(){

    return <h2>I am a {this.state.color} Car!</h2>;

}

}
```

Props:

Another way of handling component properties is by using **props**.

Props are like function arguments, and you send them into the component as attributes.

You will learn more about **props**in the next chapter.

Example:

Use an attribute to pass a color to the Car component, and use it in the render() function:

```
class Car extends React.Component{

render(){

    return <h2>I am a {this.props.color} Car!</h2>;

}

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Carcolor="red"/>);
```

Props in the Constructor:

If your component has a constructor function, the props should always be passed to the constructor and also to the React.Component via the `super()` method.

Example:

Use an attribute to pass a color to the Car component, and use it in the render() function:

```
class Car extends React.Component{

constructor(props){

    super(props);

}

render(){

    return <h2>I am a {this.props.model}!</h2>;

}

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Carmodel="Mustang"/>);
```

Components in Components:

We can refer to components inside other components:

Example:

Use the Car component inside the Garage component:

```
class Car extends React.Component{

render(){

    return <h2>I am a Car!</h2>;

}

}

class Garage extends React.Component{

render(){

    return (

<div>

<h1>Who lives in my Garage?</h1>

<Car/>

</div>

);

}

}
```

```
const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Garage/>);
```

Components in Files:

React is all about re-using code, and it can be smart to insert some of your components in separate files.

To do that, create a new file with a `.js` file extension and put the code inside it:

Note that the file must start by importing React (as before), and it has to end with the statement `export default Car;`.

Example:

Use the Car component inside the Garage component:

```
import React from 'react';

classCar extends React.Component{

render(){

    return <h2>Hi, I am a Car!</h2>;

}

}

export default Car;
```

To be able to use the `Car`component, you have to import the file in your application.

Example:

Now we import the `Car.js` file in the application, and we can use the `Car`component as if it was created here.

```
import React from 'react';

import ReactDOM from 'react-dom/client';

import Car from './Car.js';

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car/>);
```

React Class Component State:

React Class components have a built-in `state`object.

You might have noticed that we used `state`earlier in the component constructor section.

The `state`object is where you store property values that belongs to the component.

When the `state`object changes, the component re-renders.

Creating the state Object:

The state object is initialized in the constructor:

Example:

Specify the `state`object in the constructor method:

```
class Car extends React.Component{

constructor(props){

    super(props);

  this.state ={brand:"Ford"};

}

render(){

    return (

<div>

<h1>My Car</h1>

</div>

);

}

}
```

The state object can contain as many properties as you like:

Example:

Specify all the properties your component need:

```
class Car extends React.Component{

constructor(props){

    super(props);

    this.state ={

      brand:"Ford",

      model:"Mustang",

      color:"red",

      year:1964

};

}

render(){

    return (

<div>

<h1>My Car</h1>

</div>
```

```
);

}

}
```

Using the `state` Object

Refer to the `state` object anywhere in the component by using the `this.state.propertyname` syntax:

Example:

Refer to the `state` object in the `render()` method:

```
class Car extends React.Component{

constructor(props){

    super(props);

    this.state ={

      brand:"Ford",

      model:"Mustang",

      color:"red",

      year:1964

};
```

```
}
render(){
    return (
<div>
<h1>My {this.state.brand}</h1>
<p>
          It is a {this.state.color}
{this.state.model}
          from {this.state.year}.
</p>
</div>
);
}
}
```

Changing the state Object:

To change a value in the state object, use the `this.setState()` method.

When a value in the `state`object changes, the component will re-render, meaning that the output will change according to the new value(s).

Example:

Add a button with an `onClick`event that will change the color property:

```
class Car extends React.Component{

constructor(props){

    super(props);

    this.state ={

      brand:"Ford",

      model:"Mustang",

      color:"red",

      year:1964

};

}

changeColor=()=>{

    this.setState({color:"blue"});

}

render(){

    return (

<div>
```

```
<h1>My {this.state.brand}</h1>
<p>
          It is a {this.state.color}
{this.state.model}
          from {this.state.year}.
</p>
<button
type="button"
onClick={this.changeColor}
>Change color</button>
</div>
);
}
}
```

*Always use the **setState()** method to change the state object, it will ensure that the component knows its been updated and calls the render() method (and all the other lifecycle methods).*