Conditionals & Lists

CS568 – Web Application Development I

Assistant Professor Umur Inan

Computer Science Department

Maharishi International University

Maharishi International University - Fairfield, Iowa

Content

- Class component constructor
- Event handler
- Conditionals
- Lists
- Immutable state and how React updates DOM

What is Constructor?

The constructor is a method for a React component that is called before the component is mounted to the DOM.

When implementing the constructor for a React.Component subclass, you should call **super(props)** before any other statement. Otherwise, this.props will be **undefined** in the constructor, which can lead to bugs.

Setting state in Constructor

You should not call setState() in the constructor(). Instead, if your component needs to use local state, assign the initial state to **this.state directly in the constructor**.

Constructor is the only place where you should assign this.state directly. In all other methods, you need to use this.setState() instead.

```
constructor(props) {
  super(props);
  // Don't call this.setState() here!
  this.state = { counter: 0 };
  this.handleClick = this.handleClick.bind(this);
}
```

When to use Constructor

Typically, in React constructors are only used for two purposes:

1. Initializing local state by assigning an object to this.state.
2. Binding event handler methods to an instance.

Handling Events

Handling events with React elements is very similar to handling events on DOM elements. There are some syntax differences:

- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

Rendering Content Conditionally

```
state = {
   students: [
     { name: 'Alice', age: 20 },
     { name: 'Bob', age: 19 }
   ],
   showStudents: true
 };

 showHideStudents = () => {
   const showStudents =
this.state.showStudents;
   this.setState({ showStudents:
!showStudents });
 }
```

```
render() {
   return (
     <div className="App">
       <button onClick={this.showHideStudents}>Show / Hide
Students</button>
       {this.state.showStudents ?
         <div>
           <Student name={this.state.students[0].name}
             age={this.state.students[0].age}>
           </Student>
           <Student name={this.state.students[1].name}
             age={this.state.students[1].age}>
           </Student>
         </div> : null
       }
     </div>
   );
 }
```

Better Way

```
render() {
   let students = null;

   if (this.state.showStudents) {
     students = (
       <div>
         <Student
name={this.state.students[0].name}

age={this.state.students[0].age}>
         </Student>
         <Student
name={this.state.students[1].name}

age={this.state.students[1].age}>
         </Student>
       </div>
     )
   }
```

```
    return (
      <div className="App">
        <button onClick={this.showHideStudents}>Show /
Hide Students</button>
        {students}
      </div>
    );

  }
```

Outputting Lists

```
render() {
    return (
      <div className="App">
        {
          this.state.students.map(student => {
            return (
              <Student name={student.name}
                age={student.age}>
              </Student>
…
```

Outputting List

Warning: Each child in a list should have a unique "key" prop. The key prop boosts the performance by not rendering all children over and over again. Instead, it only renders the new children.

```
render() {
   return (
     <div className="App">
       {
         this.state.students.map(item => {
           return (
             <Student
               key={item.id}
               name={item.name}
               age={item.age}>
             </Student>
…
```

Updating State

```
state = {
  students: [
    { name: 'Alice', age: 20 },
    { name: 'Bob', age: 19 },
    { name: 'Jimmy', age: 21 }
  ]
};

deleteStudent = (index) => {
  const students = this.state.students;
  students.splice(index, 1);
  this.setState({ students });
}
```

```
render() {
   return (
     <div className="App">
       {
         this.state.students.map((item, index) => {
           return (
             <Student
               key={index}
               name={item.name}
               age={item.age}
               myClickHandler={() =>
this.deleteStudent(index)}
             >
             </Student>
           )
         })
       }
     </div>
   );
 }
```

Quick Recap

- Objects and arrays are reference types.
- `const students = this.state.students;` Both are pointing the same reference now.
- When we change something on students, original state also gets changed.
- In other words, we mutate the original data.

What should we do?

- Copy the array.
- Then change it.
- Remember I: slice() without arguments copies the array
- Remember II : spread operator can be used.

How React Updates DOM

- Render method does not immediately render to real DOM.
- Render is a suggestion what the HTML should look like.

- It compares virtual DOM with re-rendered (future) virtual DOM
- Virtual DOM is faster than real DOM.
- Virtual DOM is DOM representation in JavaScript.
- React keeps **2 virtual DOM**. (old virtual DOM and re-rendered virtual DOM)

- It **compares** the differences between old virtual DOM and re-rendered virtual DOM.
- If there is a change between them, then it renders it to real DOM
- It does not render all the DOM. **It only renders the differences**.

Immutables

React compares the virtual DOM with re-rendered (future) before the update in order to know what changed. This is the reconciliation process.

If the state is immutable objects, you can check to see if they changed with a simple equality operator. From this perspective, immutability removes complexity.

Because sometimes, knowing what changes can be very hard. Think about deep fields: myPackage.sender.address.country.id = 1;

An immutable value or object cannot be changed, so every update creates new value, leaving the old one untouched. For example, if your application state is immutable, you can save all the states objects in a single store to easily implement undo/redo functionality e.g, GIT.

Add an element to Immutable Array

Use ‘concat()’ which returns new array

```
const array1= [1,2,3];

const array2 = array1.concat(4);
```

Remove from Immutable Array

Use filter()’ which returns new array

```
const array1= [1,2,3];
const array2 = array1.filter(item=> item !=2);
```

Manipulating State Immutably

```
 state = {
   students: [
     { name: 'Alice', age: 20 },
     { name: 'Bob', age: 19 },
     { name: 'Jimmy', age: 21 }
   ]
 };

 deleteStudent = (index) => {
   const students = this.state.students.slice();
//const students = [...this.state.students];
   students.splice(index, 1);
   this.setState({ students: students });
 }
```

Controlled Components

- Form elements such as <input>, <textarea>, and <select> typically maintain their own state and update it based on user input.
- An input form element whose value is controlled by React in this way is called a “controlled component