# 2
# Component Composition and Lifecycle

Now that we've covered the basics of creating components let's look at how we can compose our components to make more complex views. We will also look at how we can hook into a component's lifecycle events so that we can execute code before and after our component renders as well as prevent rendering based on incoming changes to state and props.

In this chapter we will be covering the following concepts:

- Component composition
    - How to compose simple components
    - Composing components with behavior
    - How to access child components
- Component lifecycle
    - Mounting and unmounting events
    - Updating events

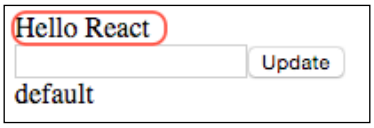
## How to compose simple components

One of the best things about React is that it is component based allowing us to easily compose our application from small autonomous components. Let's break up our Hello React application into smaller components to see how we can take advantage of React's component system.



www.it-ebooks.info

Let's start by separating our app into two components. We will make the Hello React title its own component called `HelloMessage` as shown below:




The code to create our HelloMessage component is shown in the following code:

```
var HelloMessage = React.createClass({
    render: function() {
        return <div>{this.props.message}</div>;
    }
});
```

This code defines a new component that simply writes out `this.props.message` in a `<div>` tag. Next let's update the rest of our code to use this new component to write out `Hello React` as shown in the following code:

```
var HelloReact = React.createClass({
  getInitialState: function() {
          return { message: 'default'}
      },
  updateMessage: function () {
      console.info('updateMessage');
      this.setState({
          message: this.refs.messageTextBox.value
});
  },
  render: function() {
    return (
      <div>
          <HelloMessage message='Hello React'></HelloMessage>
          <input type='text' ref='messageTextBox' />
          <button onClick={this.updateMessage}>Update</button>
          <div>{this.state.message}</div>
      </div>
      );
  }
});

ReactDOM.render(
  <HelloReact/>,
  document.getElementById('view'));
```



www.it-ebooks.info

In this code we are simply referencing our `HelloMessage` component and then setting message to `Hello React` as shown again in the following code:

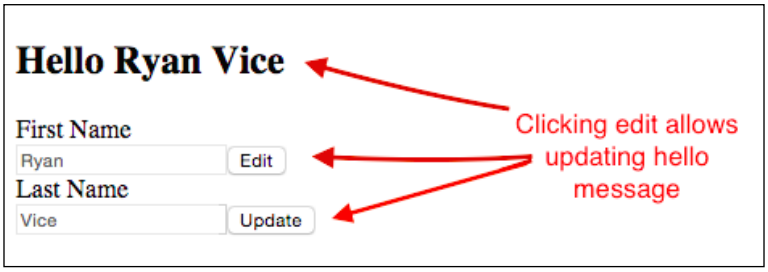
```
<HelloMessage message='Hello React'></HelloMessage>
```

This example is simple and contrived but it shows how easy it is to start breaking our application into smaller reusable chunks. This allows us to write our application by creating and using a **Domain Specific Language** (**DSL**). With our new DSL our JSX markup is made up of readable custom tags like `HelloMessage` instead of blocks of HTML markup. This will allow us to improve the readability and organization of our code dramatically.

Source code: `http://j.mp/Mastering-React-2-1-Gist`
Fiddle: `http://j.mp/Mastering-React-2-1-Fiddle`

# Composing components with behavior

Now let's make things more interesting by updating our app to be composed of components that have behavior. We are going to create a view with a form that allows for updating our `HelloMessage` as shown in the following screenshot:




If we click an **Edit** button then that button will change to an **Update** button and the associated text input box will be enabled. This will allow us to set the **First Name** or **Last Name** that is displayed in our `HelloMessage` component. After setting a **First Name** or **Last Name** we can then click the associated **Update** button and the `HelloMessage` will be updated to display the new first and last name. The preceding image shows what the component looks like after putting **Ryan** for **First Name** and **Vice** for **Last Name** and clicking an **Update** button.

Let's take a look at the following code:

```
var HelloMessage = React.createClass({
    render: function() {
        return <h2>{this.props.message}</h2>;
```



www.it-ebooks.info

```
    }
});

var TextBox = React.createClass({
    getInitialState: function() {
        return { isEditing: false }
    },
    update: function() {
        this.props.update(this.refs.messageTextBox.value);
        this.setState(
            {
                isEditing: false
            });
    },
    edit: function() {
        this.setState({ isEditing: true});
    },
    render: function() {
        return (
            <div>
              {this.props.label}<br/>
                <input type='text' ref='messageTextBox'
disabled={!this.state.isEditing}/>
                {
                    this.state.isEditing ?
                        <button onClick={this.update}>Update</button>
                        :
                        <button onClick={this.edit}>Edit</button>
                }
            </div>
        );
    }
});

var HelloReact = React.createClass({
    getInitialState: function () {
        return { firstName: '', lastName: ''}
    },
    update: function(key, value) {
        var newState = {};
        newState[key] = value;
        this.setState(newState);
    },
    render: function() {
```



www.it-ebooks.info

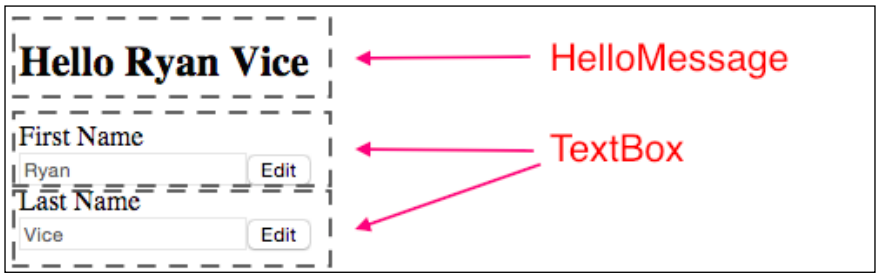
```
            return (
                <div>
                    <HelloMessage
                        message={'Hello ' + this.state.firstName + ' ' +
this.state.lastName}>
                    </HelloMessage>
                    <TextBox label='First Name' update={this.update.
bind(this, 'firstName')}>
                    </TextBox>
                    <TextBox label='Last Name'
                        update={this.update.bind(this, 'lastName')}>
                    </TextBox>
                </div>
            );
        }
    });

    ReactDOM.render(
        <HelloReact/>,
        document.getElementById('view'));
```

Run the code and get a feel for how it works.

Source code: `http://j.mp/Mastering-React-2-2-Gist`
Fiddle: `http://j.mp/Mastering-React-2-2-Fiddle`

# How it works

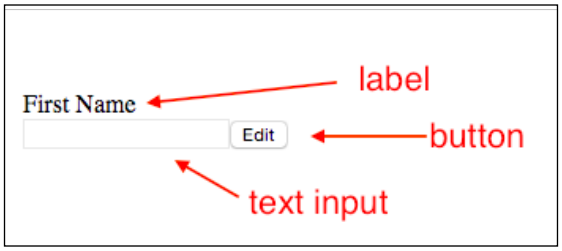
We have divided our view into the components shown in the following screenshot:






www.it-ebooks.info

The `HelloMessage` component is the same one that we created in the previous example but now we've added a new `TextBox` component to the mix. Each `TextBox` component has a label, text input and button as shown in the following screenshot:




We declare two instances of our `TextBox` component in our `HelloReact` component's render method shown in the following code:

```
var HelloReact = React.createClass({
    getInitialState: function () {
        return { firstName: '', lastName: ''}
    },
    update: function(key, value) {
        var newState = {};
        newState[key] = value;
        this.setState(newState);
    },
    render: function() {
        return (
            <div>
                <HelloMessage
                    message={'Hello '
+ this.state.firstName + ' '
+ this.state.lastName}>
                </HelloMessage>
                <TextBox label='First Name'
update={this.update.bind(null, 'firstName')}>
                </TextBox>
                <TextBox label='Last Name'
update={this.update.bind(null, 'lastName')}>
                </TextBox>
            </div>
        );
    }
});
```



www.it-ebooks.info

Here we are creating two `TextBox` component instances and setting their label and `update` properties. The `update` property needs to be set to the callback function that will be called when the input component's `onChange` event fires (we will look more closely at this in the following code). We are setting the `update` property to the new method created by calling Javascript's `bind` method on `this.update` shown in the following code:

```
update={this.update.bind(null, 'lastName')}
```

If you are not familiar with JavaScript's `bind` method it will return a new method that allows us to do two things. First, it allows us to set the function's context which is the value of the `this` variable in the function's scope. Second, it allows us to curry the method's arguments, which allows us to prepend arguments to the argument array that will be used to call the method when it's invoked. We are passing null for the first parameter as we are not interested in changing the functions context and this will result in `this.update` being called in the context of our `HelloReact` component instance meaning that `this.setState` will refer to `HelloReact.setState` which is what we want. More interesting to our goal, we are using JavaScript's bind method to curry the `this.update` function's arguments. Doing this allows us to provide the `this.update` method a key argument from `HelloReact` render's method. This technique allows us to configure how our callback method will be called. Here we are using JavaScript's bind method to let the consuming component pass the key argument when it invokes the update callback method in response to an `onChange` synthetic event.

In the `HelloReact` component's `update` method, as shown in the following code, we are expecting to be passed a key and value. As we just discussed, the key was sent via the `bind` method and we will use the key along with the value that was passed from the react synthetic event to update the `HelloReact` component's state. We update the state by first creating a new object called `newState`. Then we use JavaScript's index operator on the `newState` object with our key to create a new property on the `newState` object using JavaScript's index operator. We then assign value to the new property that was created on the `newState` object. Finally we call `this.setState` and pass in `newState` which will merge `newState` with `this.state` causing our component to rerender with the updated value.

```
update: function(key, value) {
    var newState = {};
    newState[key] = value;
    this.setState(newState);
},
```

After updating our state our `HelloMessage.message` property will be updated shown in the following code:

```
{'Hello ' + this.state.firstName + ' ' + this.state.lastName}
```



www.it-ebooks.info

This will make it so that our `HelloMessage` will get rerendered and updated every time the `HelloReact` components state is updated from the call to `this.setState` method from the `this.update` method.

Next let's look at the `TextBox` component that will call the `HelloReact` component's `update` method. The `TextBox` component's code is shown below:

```
var TextBox = React.createClass({
    getInitialState: function() {
        return { isEditing: false }
    },
    update: function() {
        this.props.update(this.refs.messageTextBox.value);
        this.setState(
            {
                isEditing: false
            });
    }
```

Here we first pass the value in our text input via `this.refs.messageTextBox.value` into the `this.props.update` method. We then update our state so that `isEditing` is false. As we saw in the preceding code, we used JavaScript's `bind` method to wire up the `TextBox.update` property. Now when we call `this.props.update(value)` this will result in the call being `this.props.update(key, value)` where key was assigned in the bind call in the `HelloReact` component's render method. The remaining code in `TextBox` deals with controlling the components enabled and disabled state and the text displayed in button shown in the following code:

```
    edit: function() {
        this.setState({ isEditing: true});
    },
    render: function() {
        return (
            <div>
                {this.props.label}<br/>
                <input type='text'
ref='messageTextBox'
disabled={!this.state.isEditing}/>
                {
                    this.state.isEditing ?
                        <button onClick={this.update}>
Update
  </button>
                        :
                        <button onClick={this.edit}>
```



www.it-ebooks.info

```
Edit
  </button>
                    }
            </div>
        );
    }
});
```

We are defining an edit method that simply sets `this.state.isEditing` to `true` by calling the `this.setState` method. Then we are defining a render method that creates a label, an input text box and a button. We are using JavaScript's ternary operator to conditionally create a different button depending on the value of `this.state.isEditing`. If `this.state.isEditing` is `true` then we create an `Update` button while if `this.state.isEditing` is `false` we will create an `Edit` button. We also set our input component's `disabled` property to `!this.state.Editing` so that our input will be disabled when we are not editing.

# Accessing a component's children

In React when we want to access the inner HTML of a component or a component that's been embedded inside of a component we can use `this.props.children`. This feature is very similar to Angular's Transclusion, WebComponent's Contents or Ember's Yield. To demonstrate this we are going to update the button from our previous example to be the Button component shown in the following code:

```
var Button = React.createClass({
   render: function() {
       return (
<button onClick={this.props.onClick}>
{this.props.children}
</button>
);
   }
});
```

What we have created here is `Button` component that can have its opening and closing tags wrapped around the HTML elements and React components that it wants to display within the button that it will render. To demonstrate this we will create a component for displaying glyph icons using bootstrap shown in the following code:

```
var GlyphIcon = React.createClass({
   render: function() {
       return (
<span className={'glyphicon glyphicon-'
```

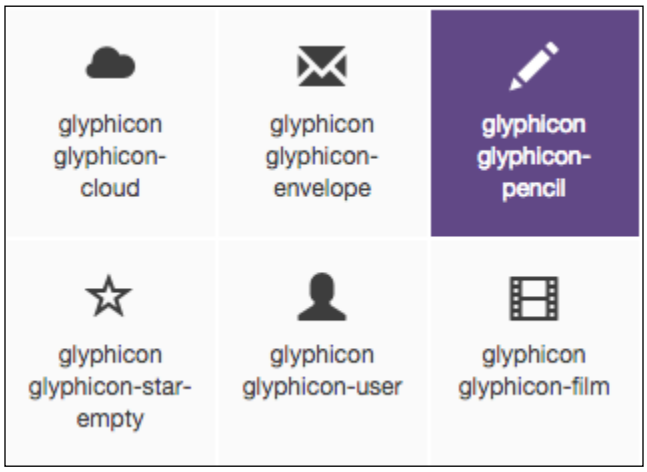


www.it-ebooks.info

```
+ this.props.icon}>
</span>
);
   }
});
```

Note that to make this work we updated the JsFiddle references to include a reference to Twitter's Bootstrap framework. For more information about Bootstrap see the documentation here: `http://getbootstrap.com/`

Our `GlyphIcon` component will simplify displaying a bootstrap Glyphicon if we simply configure it by specifying the last part of the Glyphicon's style name. In the following screenshot, we have shown a few of the Glyphicon styles:




So for example we can display a pencil by specifying pencil for our `GlyhIcon` component's `icon` property. Next we will update our `TextBox` component to use our `GlyhIcon` class shown in the following code:

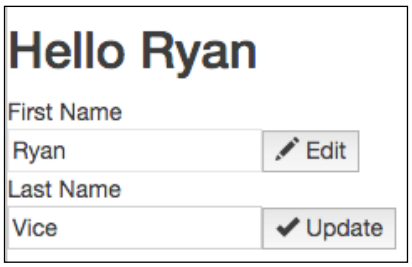
```
        render: function() {
            return (
                <div>
                  {this.props.label}<br/>
                    <input
type='text'
ref='messageTextBox'
disabled={!this.state.isEditing}/>
                    {
```



www.it-ebooks.info

```
                    this.state.isEditing ?
                        <Button onClick={this.update}>
<GlyphIcon icon='ok'/> Update
  </Button>
                        :
                        <Button onClick={this.edit}>
<GlyphIcon icon='pencil'/> Edit
  </Button>
                    }
            </div>
        );
    }
```

In this code we are using our button component to wrap both a `GlyphIcon` component and also some text which will allow us to display buttons with both text and icon's as shown in the following screenshot:




The full code is shown below:

```
var HelloMessage = React.createClass({
    render: function() {
        return <h2>{this.props.message}</h2>;
    }
});

var Button = React.createClass({
   render: function() {
       return (
<button onClick={this.props.onClick}>
{this.props.children}
</button>
);
   }
});

var GlyphIcon = React.createClass({
```



www.it-ebooks.info

```
    render: function() {
        return (
<span className={'glyphicon glyphicon-'
+ this.props.icon}>
</span>
);
    }
});

var TextBox = React.createClass({
    getInitialState: function() {
        return { isEditing: false, text: this.props.label }
    },
    update: function() {
        this.setState(
            {
                text: this.refs.messageTextBox.getDOMNode().value,
                isEditing: false
            });
        this.props.update();
    },
    edit: function() {
        this.setState({ isEditing: true});
    },
    render: function() {
        return (
            <div>
              {this.props.label}<br/>
                <input
type='text'
ref='messageTextBox'
disabled={!this.state.isEditing}/>
                {
                    this.state.isEditing ?
                        <Button onClick={this.update}>
<GlyphIcon icon='ok'/> Update
  </Button>
                        :
                        <Button onClick={this.edit}>
<GlyphIcon icon='pencil'/> Edit
  </Button>
                }
            </div>
        );
```



www.it-ebooks.info

```
    }
});

var HelloReact = React.createClass({
    getInitialState: function () {
        return { firstName: '', lastName: ''}
    },
    update: function () {
        this.setState({
            firstName:
                this.refs.firstName.refs.messageTextBox.getDOMNode().
value,
            lastName:
                this.refs.lastName.refs.messageTextBox.getDOMNode().
value});
    },
    render: function() {
        return (
            <div>
                <HelloMessage
                    message={'Hello ' + this.state.firstName + ' ' +
this.state.lastName}>
                </HelloMessage>
                <TextBox label='First Name' ref='firstName'
                    update={this.update}>
                </TextBox>
                <TextBox label='Last Name'  ref='lastName'
                    update={this.update}>
                </TextBox>
            </div>
        );
    }
});

ReactDOM.render(
    <HelloReact/>,
    document.getElementById('view'));
```

Source code: `http://j.mp/Mastering-React-2-3-Gist`
Fiddle: `http://j.mp/Mastering-React-2-3a-Fiddle`



www.it-ebooks.info

# Component lifecycle - mounting and unmounting

Components in React have a lifecycle of events that we can easily subscribe to by defining the associated methods on our component definition object. Let's go ahead and update our previous example to see this feature in action.

```
var HelloMessage = React.createClass({
    componentWillMount: function() {
        console.log('componentWillMount');
    },
    componentDidMount: function() {
        console.log('componentDidMount');
    },
    componentWillUnmount: function() {
        console.log('componentWillUnmount');
    },
    render: function() {
        console.log('render');
        return <h2>{this.props.message}</h2>;
    }
});
```

Here we have updated our `HelloMessage` component to log to the console the following three React component lifecycle events:

- `componentWillMount`: This event will be called right before a component mounts
- `componentDidMount`: This event will be called right after a component mounts
- `componentWillUnmount`: This event will be called right before a component unmounts

We are also logging our `render` method to the console so that we can see when the various lifecycle events occur relative to render.

Let's also update our `HelloReact` component to add a button that will reload our `HelloMessage` component allowing us to see what happens when it unmounts. We've added this button to the `render` method shown in the following code:

```
    render: function() {
        return (
            <div>
                <HelloMessage
                    message={'Hello '
```



www.it-ebooks.info

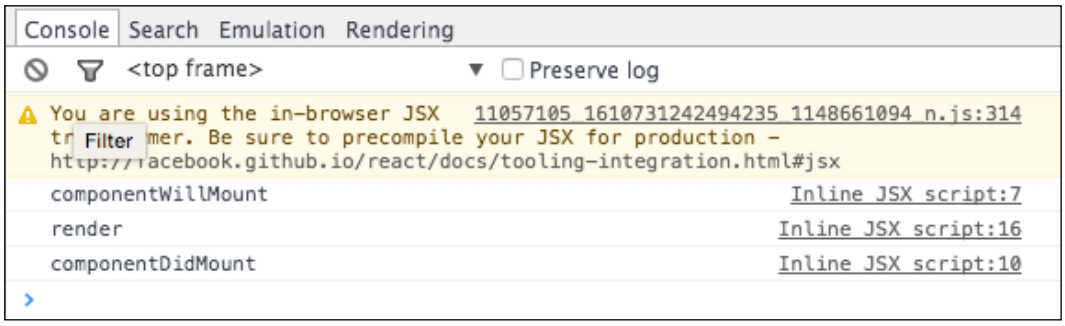
```
    + this.state.firstName + ' '
    + this.state.lastName}>
                    </HelloMessage>
                    <TextBox label='First Name' ref='firstName'
                        update={this.update}>
                    </TextBox>
                    <TextBox label='Last Name'  ref='lastName'
                        update={this.update}>
                    </TextBox>
                    <button onClick={this.reload}>Reload</button>
                </div>
            );
        }
```

And then let us add a `reload` method to our `HelloReact` component that will call `React.unmountComponentAtNode` which will unmount our component. We then call `ReactDOM.render` to mount our component.

```
        reload: function() {
            ReactDOM.unmountComponentAtNode(
    document.getElementById('view'));
            ReactDOM.render(
                <HelloReact/>,
                document.getElementById('view'));
        },
```

Source code: `http://j.mp/Mastering-React-2-4a-Gist`
Fiddle: `http://j.mp/Mastering-React-2-4a-Fiddle`

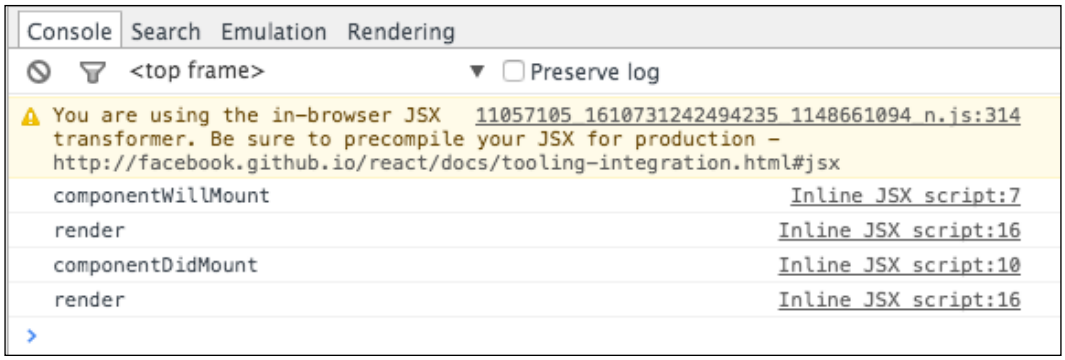
Now let's go ahead and run the code in JsFiddle and open our browsers debugging tools (*F12* in Chrome) so that we can see the console output. After running the code we see the following output:



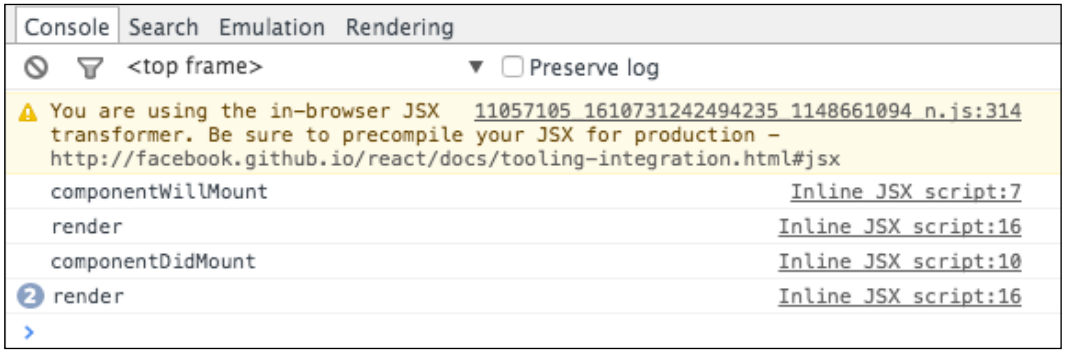



www.it-ebooks.info

And as we can see we get a call to componentWillMount right before Render is called and then we get a call to `componentDidMount` right after render is called. This gives us an opportunity to run code both before and after our render method. Next let's add a **First Name** and we can see what happens in the console as shown below:



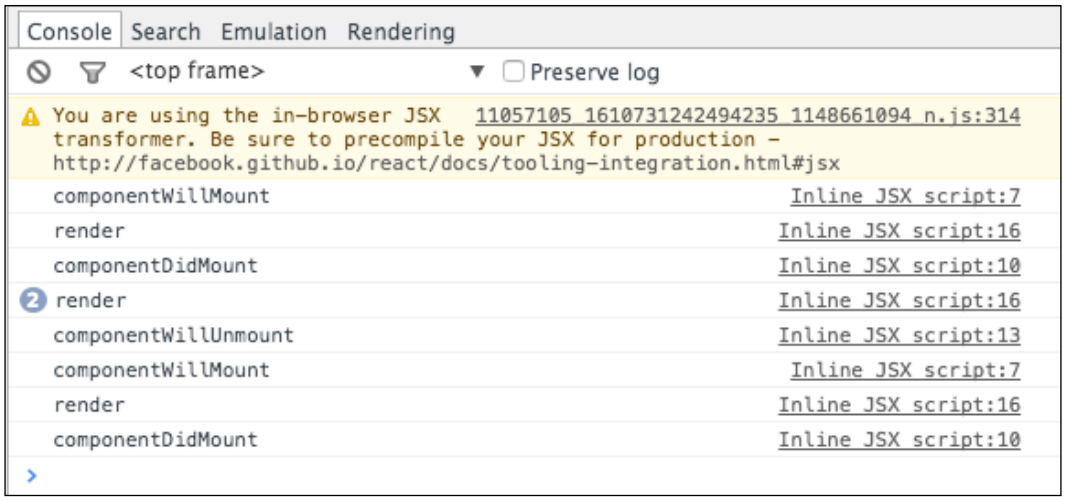

We get one more call to render but `componentWillMount` and `componentDidMount` are not called because our `HelloMessage` component is already mounted and we are simply causing React to call the `HelloMessage.render` method. Let's set a last name and look at the console output:




Probably no surprise here but we find that render is called again but that none of the lifecycle events are called. Next let's click the **Reload** button and look at the following output:



www.it-ebooks.info




Now we see that `componentWillUnmount` is called as our component is unmounted and then we repeat the same sequence we saw earlier as the component is mounted again.

# Component lifecycle – updating events

There are also events that will allow us to execute code relative to when our component's state and properties get updated. To demonstrate this we will look at the sample application shown below:


This is an extremely contrived example that is intended to help us see updating events in action. This application has two buttons:

- **Like** button: This button will increase the like count
- **Unlike** button: This button will decrease the like count



www.it-ebooks.info

The application also has the following features:

- It displays a total count of likes
- It has a GlyphIcon component that will show an up arrow if the like count is increasing or a down arrow if the like count is decreasing
- It will not update the view until after we have two or more likes

Let's take a look at how we can implement these features by taking advantage of the updating lifecycle events as shown in the following code:

```
var Button = React.createClass({
    render() {
        return (
<button onClick={this.props.onClick}>
{this.props.children}
</button>
);
    }
});

var GlyphIcon = React.createClass({
    render() {
        return (
<span className={'glyphicon glyphicon-'
+ this.props.icon}>
</span>
);
    }
});

var HelloReact = React.createClass({
    getDefaultProps() {
        return {likes: 0};
    },
    getInitialState() {
        return {isIncreasing: false};
    },
    componentWillReceiveProps(nextProps) {
        this._logPropsAndState('componentWillReceiveProps()');
        console.log('nextProps.likes: ' + nextProps.likes);

        this.setState({
isIncreasing: nextProps.likes > this.props.likes
  });
```



www.it-ebooks.info

```
    },
    shouldComponentUpdate(nextProps, nextState) {
        this._logPropsAndState('shouldComponentUpdate()');
        console.log(
'nextProps.likes: ',
nextProps.likes,
' nextState.isIncreasing: ',
nextState.isIncreasing);
        return nextProps.likes > 1;
    },
    componentDidUpdate(prevProps, prevState) {
        this._logPropsAndState('componentDidUpdate');
        console.log(
'prevProps.likes: ',
prevProps.likes,
' prevState.isIncreasing:',
prevState.isIncreasing);
        console.log('componentDidUpdate() gives an opportunity to
execute code after react is finished updating the DOM.');
    },
    _logPropsAndState(callingFunction) {
        console.log('=> ' + callingFunction);
        console.log('this.props.likes: ' + this.props.likes);
        console.log('this.state.isIncreasing: '
+ this.state.isIncreasing);
    },
    like() {
        this.setProps({likes: this.props.likes+1});
    },
    unlike() {
        this.setProps({likes: this.props.likes-1});
    },
    render() {
this._logPropsAndState("render()");
        return (
            <div>
                <Button onClick={this.like}>
<GlyphIcon icon='thumbs-up'/> Like
</Button>
                <Button onClick={this.unlike}>
<GlyphIcon icon='thumbs-down'/> Unlike
</Button>
                <br/>
                Likes {this.props.likes}
```



www.it-ebooks.info

```
                <GlyphIcon icon={
(this.state.isIncreasing)
? 'circle-arrow-up' : 'circle-arrow-down'}/>
            </div>
        );
    }
});

ReactDOM.render(
    <HelloReact/>,
    document.getElementById('view'));
```

Source code: `http://j.mp/Mastering-React-2-5a-Gist`
Fiddle: `http://j.mp/Mastering-React-2-5-Fiddle`

# How it works

We've implemented the following component lifecycle events in our `HelloReact` component in the preceding code:

- `componentWillReceiveProps`
- `shouldComponentUpdate`
- `componentDidUpdate`

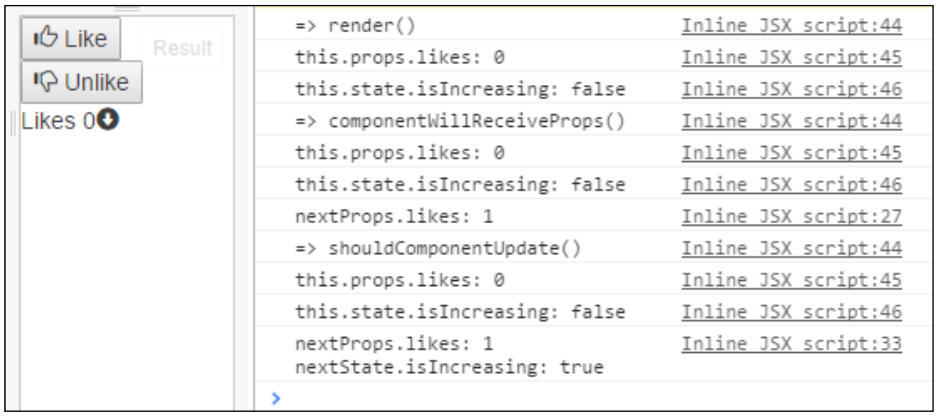
We've also added a good bit of logging code that will allow us to see the state and properties of our component in these lifecycle event methods. We've added the method shown in the following code that we can call and write out the calling method name along with the current value of `this.props.likes` and `this.state.isIncreasing`.

```
    _logPropsAndState(callingFunction) {
        console.log('=> ' + callingFunction);
        console.log('this.props.likes: ' + this.props.likes);
        console.log('this.state.isIncreasing: ' + this.state.
isIncreasing);
    },
```



www.it-ebooks.info

Let's run the code and confirm that it works as described in the preceding code. First let's click the **Like** button. We will see that clicking the **Like** button does not have any effect on the UI because of the rule we added to the `shouldComponentUpdate` method as shown below:




Here we are looking at the console log and can see that the `componentWillReceiveProps` method is called after the render method but before our components state is updated. When the `componentWillReceiveProps` method is called the props haven't changed from what we saw in the render method and `this.props.likes` is `0` and `this.state.isIncreasing` is `false`.

We also see that the `componentWillReceiveProps` is passed the future value of `this.props` in the `nextProps` argument and we can see that `nextProps.likes` is `1` as we would expect.

The `componentWillReceiveProps` method also gives us an opportunity to apply our business rule to determine if the component's like count is increasing or decreasing as shown in the following code:

```
    componentWillReceiveProps(nextProps) {
        this._logPropsAndState('componentWillReceiveProps()');
        console.log('nextProps.likes: ' + nextProps.likes);

        this.setState({
isIncreasing: nextProps.likes > this.props.likes});
    }
```



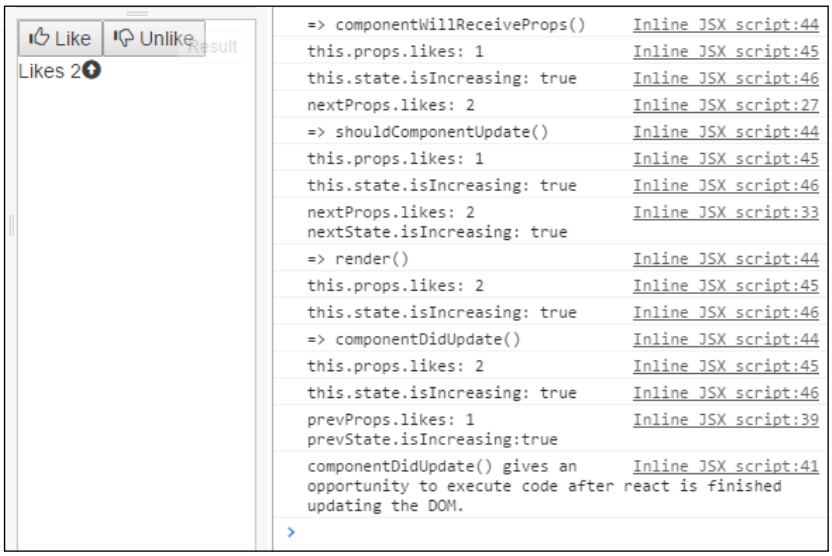
www.it-ebooks.info

We also see in the console that the `shouldComponentUpdate` method is called. The `shouldComponentUpdate` gets all the information available in the `componentWillReceiveProps` method but is also passed the future value of the `this.state` property via the `nextState` argument. Looking at the `nextState.isIncreasing` property we can see that it is true meaning that `this.state.isIncreasing` will be `true` when the component renders which is what we would expect. The updated value of `this.state.isIncreasing` reflects the call to `this.setState` from the `componentWillReceiveProps` method shown in the preceding code. The `shouldComponentUpdate` method also gives us an opportunity to apply our business rule that prevents the component from updating if the `this.props.likes` property is less than `2` as shown in the following code:

```
    shouldComponentUpdate(nextProps, nextState) {
        this._logPropsAndState('shouldComponentUpdate()');
        console.log('nextProps.likes: '
+ nextProps.likes
                + ' nextState.isIncreasing: '
+ nextState.isIncreasing);
        return nextProps.likes > 1;
    }
```

By returning `false` when evaluating `nextProps.likes > 1`, we are preventing our component from updating.

Next clear the console and click the **Like** button for a second time. We will see that the UI now updates to show two likes, an up arrow to indicate that likes are increasing as well as the log statements as shown below:






www.it-ebooks.info

Here we see in our log statements that we now get a call to the `componentDidUpdate` method which gets the previous properties and previous state passed to it allowing us to execute business rules and logic after our component updates. We are not implementing any rules at this time and are simply writing out some values to demonstrate this feature. The `componentDidUpdate` method is called now because we are returning true from `componentShouldUpdate` when evaluating `nextProps.likes > 1`.

This expression now evaluates to true because the `nextProps.likes` property is `2`.

Now let's click the **Like** button again followed by clicking the **Unlike** button. We will now see that our GlyphIcon arrow is pointing down as shown in the following screenshot:


This is because we've set `this.state.isIncreasing` to `false` in our `componentWillReceiveProps` method because this expression `nextProps.likes > this.props.likes` now evaluates to `false`.

# Summary

In this chapter we looked at how to compose components and how to access child components and\or the inner HTML of our components. We then looked at how to hook into the component lifecycle events to allow us to execute logic relative to mounting events and updating events.

In the next chapter will look at mixin's, dynamic components, property validation and forms.



www.it-ebooks.info