COMPSCI 326
Web Programming

Interactivity and Component Life-Cycle

Objectives

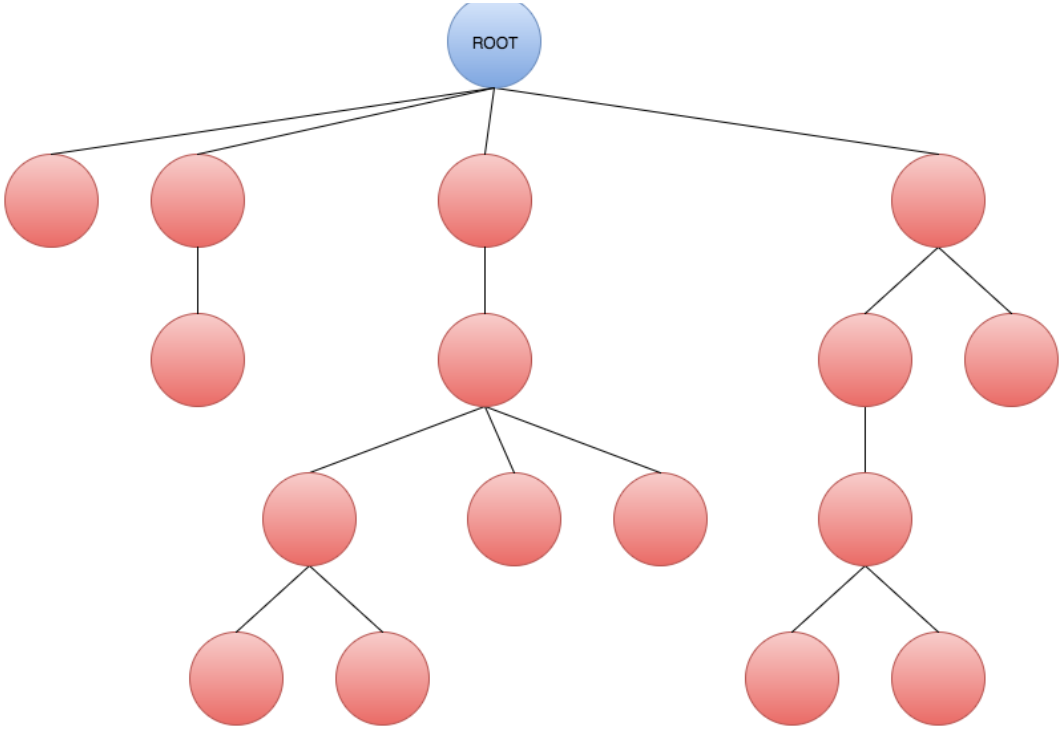
- Rendering and State
- State, Interactivity, and Design
  - Stateless and Stateful Components
- React Interactivity
- React Component Life-Cycle

Architecting React Applications

In order to build well architected React components we need to understand how data flows through our React applications.

- We need to understand how React views our components.
- We need to look at how data flows through a React application.
- **We need to understand the React component life-cycle**

React's View



React applications are React component compositions. Much like HTML, React is hierarchical and expressed easily as a tree.

A React Component would look like the following:

- Root – entry point to application
- Children (typically top level layout)
  - Menu
  - Footer
  - Side Bar
  - Main Section

Each node is a distinct component that produces a render fragment.

React Render

- The benefit of having a **render** method is that it can compute complicated logic and *choose* which value to return.

- We can use **props** and **state** to fill in parts that are generated dynamically from the model.

The ability to configure components using properties is a key factor in making React components **reusable** and **composable**.

The props are passed from parent to child components and they can't be changed from inside the child component.

Props are owned by the parent!

What about state?

Props are used for data that doesn't change inside a particular component – they are **presentational**.

State, however, is reserved for **interactivity** – when data changes over time.

How do changes occur over time?

React State-Related Methods

- **`this.getInitialState()`**
  Returns the initial state of the component. This is useful for returning a component to its original setup.

- **`this.setState(function | object)`**
  Modifies the state of the component. This kicks off the component life-cycle (more on this shortly).

State, Interactivity, and Design

- **Stateless Components**
  - A component that only receives **props** are *stateless*.
  - Their primary goal is to just render data.
  - A good design consists mostly of stateless components.

State, Interactivity, and Design

- **Stateless Components**
  - A component that only receives **props** are *stateless*.
  - Their primary goal is to just render data.
  - A good design consists mostly of stateless components.

- **Stateful Components**
  - A component that changes through interactivity using **state** is *stateful*.

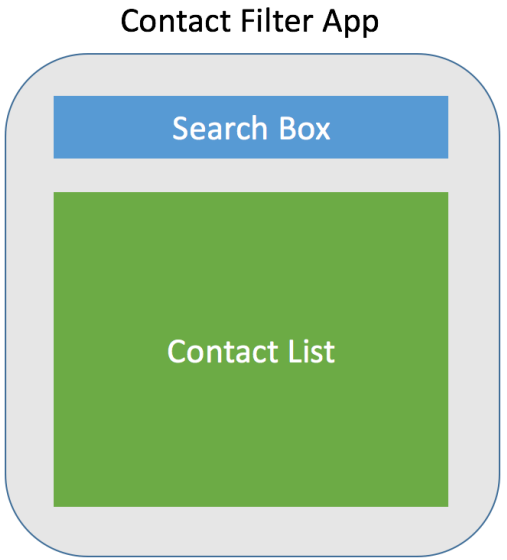
State, Interactivity, and Design

- **Stateless Components**
  - A component that only receives **props** are *stateless*.
  - Their primary goal is to just render data.
  - A good design consists mostly of stateless components.

- **Stateful Components**
  - A component that changes through interactivity using **state** is *stateful*.
  - **Best Practice:**
    - have a stateful component that wraps stateless components passing in state to its children using props.
    - This encapsulates all the **interaction** logic in one place, the stateful component, while the stateless components take care of rendering data declaratively.

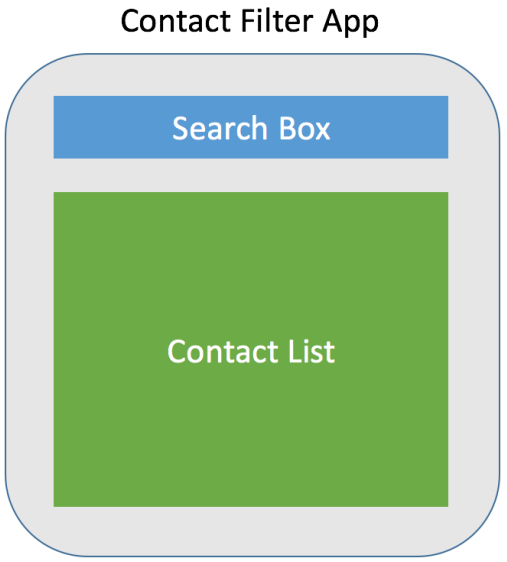
Best Practice Example

```
let list = [
  {id: 1463777842462,
   name: 'Jordan Walke',
   email: 'jordan@somemail.com'},
  {id: 1463777853704,
   name: 'Dan Abramov',
   email: 'dan@somemail.com'},
  {id: 1463777863341,
   name: 'Sebastian Markbage',
   email: 'vjeux@somemail.com'},
  {id: 1463777872559,
   name: 'Pete Hunt',
   email: 'pete@somemail.com'}
 ];
```



Best Practice Example

```
ReactDOM.render(
  <App initialItems={list}/>,
  document.getElementById('app-container'));
```




```
let list = [
  {id: 1463777342462,
   name: 'Jordan Walke',
   email: 'jordan@sammasi.com'},
  {id: 1463777853704,
   name: 'Dan Abramov',
   email: 'dan@sammasi.com'},
  {id: 1463777863341,
   name: 'Sebastian Markbage',
   email: 'sjava@sammasi.com'},
  {id: 1463777872539,
   name: 'Pete Hunt',
   email: 'pete@sammasi.com'}
];
```

Best Practice Example

```
class App extends React.Component {
  getInitialState() {
    return {text: '', items: this.props.initialItems,};
  }

  render() {
    var text = this.state.text;
    var filterdContacts = this
      .state
      .items
      .filter(function(contact) {
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.changeText}/>
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```

We define the **getInitialState** method to return the initial state of this component.

Here is an example of an outer component that will handle the state of the application and its interactivity.

Best Practice Example

```
class App extends React.Component {
  getInitialState() {
    return {text: '', items: this.props.initialItems,};
  }

  render() {
    var text = this.state.text;
    var filterdContacts = this
      .state
      .items
      .filter(function(contact) {
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.changeText}/>
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```

The **changeText** method defines the interactivity aspect of this component. It sets the “text” of the App.

Here is an example of an outer component that will handle the state of the application and its interactivity.

Best Practice Example

```
class App extends React.Component {
  getInitialState() {
    return {text: '', items: this.props.initialItems,};
  }

  render() {
    var text = this.state.text;
    var filterdContacts = this
      .state
      .items
      .filter(function(contact) {
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.changeText}/>
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```

The **render** method is kicked off when the state changes.

Here is an example of an outer component that will handle the state of the application and its interactivity.

Best Practice Example

```
class App extends React.Component {
  getInitialState() {
    return {text: '', items: this.props.initialItems,};
  }

  render() {
    var text = this.state.text;
    var filterdContacts = this
      .state
      .items
      .filter(function(contact) {
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.changeText}/>
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```

We grab the text from the state and filter the contacts by name or by email address.

Here is an example of an outer component that will handle the state of the application and its interactivity.

Best Practice Example

```
class App extends React.Component {
  getInitialState() {
    return {text: '', items: this.props.initialItems,};
  }

  render() {
    var text = this.state.text;
    var filterdContacts = this
      .state
      .items
      .filter(function(contact) {
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.changeText}/>
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```

We render a **Search** component and a **ContactList**.

Here is an example of an outer component that will handle the state of the application and its interactivity.

Best Practice Example

```
class App extends React.Component {
  getInitialState() {
    return {text: '', items: this.props.initialItems,};
  }

  render() {
    var text = this.state.text;
    var filterdContacts = this
      .state
      .items
      .filter(function(contact) {
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.changeText}/>
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```

We render a **Search** component and a **ContactList**.

We pass the state (text) of the App to the Search component.

Here is an example of an outer component that will handle the state of the application and its interactivity.

Best Practice Example

```
class App extends React.Component {
  getInitialState() {
    return {text: '', items: this.props.initialItems,};
  }

  render() {
    var text = this.state.text;
    var filterdContacts = this
      .state
      .items
      .filter(function(contact) {
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.changeText}/>
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```

We render a **Search** component and a **ContactList**.

We pass the state (text) of the App to the Search component. **And** the this.changeText method to the *stateless* Search component.

Here is an example of an outer component that will handle the state of the application and its interactivity.

Best Practice Example

```
class App extends React.Component {
  getInitialState() {
    return {text: '', items: this.props.initialItems,};
  }

  render() {
    var text = this.state.text;
    var filterdContacts = this
      .state
      .items
      .filter(function(contact) {
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```

We render a **Search** component and a **ContactList**.

We pass the state (text) of the App to the Search component. **And** the this.changeText method to the *stateless* Search component.

We pass the filtered contacts to the *stateless* **ContactList** component.

Here is an example of an outer component that will handle the state of the application and its interactivity.

Best Practice Example

```
class Search extends React.Component {
  render() {
    return (<input type='text'
                   placeholder='search'
                   value={this.props.text}
                   onChange={this.handleTextChange}/>);
  }

  handleTextChange(event) {
    this
      .props
      .onTextChange(event.target.value);
  }
}
```

Remember, the search component is a *stateless* component.

The **render** method returns an **input** element to render the search input text box.

Here is an example of an outer component that will handle the state of the application and its interactivity.

Best Practice Example

```
class Search extends React.Component {
  render() {
    return (<input type='text'
                   placeholder='search'
                   value={this.props.text}
                   onChange={this.handleTextChange}/>);
  }

  handleTextChange(event) {
    this
      .props
      .onTextChange(event.target.value);
  }
}
```

Remember, the search component is a *stateless* component.

The **render** method returns an **input** element to render the search input text box.

We set the value of the text box to be the value of the text given from the **App** component through the **props**.

Here is an example of an outer component that will handle the state of the application and its interactivity.

Best Practice Example

```
class Search extends React.Component {
  render() {
    return (<input type='text'
                   placeholder='search'
                   value={this.props.text}
                   onChange={this.handleTextChange}/>);
  }

  handleTextChange(event) {
    this
      .props
      .onTextChange(event.target.value);
  }
}
```

Remember, the search component is a *stateless* component.

The **render** method returns an **input** element to render the search input text box.

We set the value of the text box to be the value of the text given from the **App** component through the **props**.

We also indicate that when the text box changes we must call the **handleTextChange** method.

Here is an example of an outer component that will handle the state of the application and its interactivity.

Best Practice Example

```
class Search extends React.Component {
  render() {
    return (<input type='text'
                   placeholder='search'
                   value={this.props.text}
                   onChange={this.handleTextChange}/>);
  }

  handleTextChange(event) {
    this
      .props
      .onTextChange(event.target.value);
  }
}
```

The handleTextChange method *delegates* the responsibility of *interactivity* to the parent component (App) through the given callback function assigned to the **props** value **onTextChange**.

```
return (
  <div>
    <Search text={this.state.text} onTextChange={this.changeText}/>
    <ContactList items={filterdContacts}/>
  </div>
);
```

Here is an example of an outer component that will handle the state of the application and its interactivity.

Best Practice Example

```
class App extends React.Component {
  getInitialState() {
    return {text: '', items: this.props.initialItems,};
  }

  render() {
    var text = this.state.text;
    var filterdContacts = this
      .state
      .items
      .filter(function(contact) {
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.changeText}/>
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```

```
class Search extends React.Component {
  render() {
    return (<input type='text'
                   placeholder='search'
                   value={this.props.text}
                   onChange={this.handleTextChange}/>);
  }

  handleTextChange(event) {
    this
      .props
      .onTextChange(event.target.value);
  }
}
```

Here is an example of an outer component that will handle the state of the application and its interactivity.

Best Practice Example

```
class App extends React.Component {
  getInitialState() {
    return {text: '', items: this.props.initialItems,};
  }

  render() {
    var text = this.state.text;
    var filterdContacts = this
      .state
      .items
      .filter(function(contact) {
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.changeText}/>
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```


```
class Search extends React.Component {
  render() {
    return (<input type='text'
                   placeholder='search'
                   value={this.props.text}
                   onChange={this.handleTextChange}/>);
  }

  handleTextChange(event) {
    this
      .props
      .onTextChange(event.target.value);
  }
}
```

Here is an example of an outer component that will handle the state of the application and its interactivity.

Best Practice Example

```
class App extends React.Component {
  getInitialState() {
    return {text: '', items: this.props.initialItems,};
  }

  render() {
    var text = this.state.text;
    var filterdContacts = this
      .state
      .items
      .filter(function(contact) {
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.changeText}/>
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```

```
class Search extends React.Component {
  render() {
    return (<input type='text'
                   placeholder='search'
                   value={this.props.text}
                   onChange={this.handleTextChange}/>);
  }

  handleTextChange(event) {
    this
      .props
      .onTextChange(event.target.value);
  }
}
```

Here is an example of an outer component that will handle the state of the application and its interactivity.

Best Practice Example

```
class App extends React.Component {
  getInitialState() {
    return {text: '', items: this.props.initialItems,};
  }

  render() {
    var text = this.state.text;
    var filterdContacts = this
      .state
      .items
      .filter(function(contact) {
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.changeText}/>
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```

```
class Search extends React.Component {
  render() {
    return (<input type='text'
                   placeholder='search'
                   value={this.props.text}
                   onChange={this.handleTextChange}/>);
  }

  handleTextChange(event) {
    this
      .props
      .onTextChange(event.target.value);
  }
}
```

Here is an example of an outer component that will handle the state of the application and its interactivity.

Best Practice Example

```
class App extends React.Component {
  getInitialState() {
    return {text: '', items: this.props.initialItems,};
  }

  render() {
    var text = this.state.text;
    var filterdContacts = this
      .state
      .items
      .filter(function(contact) {
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.changeText}/>
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```

```
class Search extends React.Component {
  render() {
    return (<input type='text'
                   placeholder='search'
                   value={this.props.text}
                   onChange={this.handleTextChange}/>);
  }

  handleTextChange(event) {
    this
      .props
      .onTextChange(event.target.value);
  }
}
```

Here is an example of an outer component that will handle the state of the application and its interactivity.

Best Practice Example

Thus, best practices dictate that applications in React are best designed top-down with a single **stateful** component wrapping child **stateless** components.

Search is a stateless component.

```
class Search extends React.Component {
  render() {
    return (<input type='text'
                   placeholder='search'
                   value={this.props.text}
                   onChange={this.handleTextChange}/>);
  }

  handleTextChange(event) {
    this
      .props
      .onTextChange(event.target.value);
  }
}
```

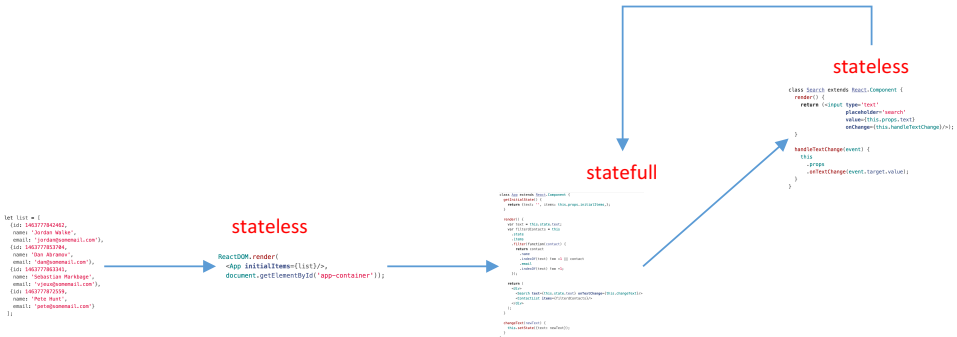
Best Practice Example

Thus, best practices dictate that applications in React are best designed top-down with a single **stateful** component wrapping child **stateless** components.

Search is a stateless component.

ContactItem is a stateless component.

ContactList is a stateless component.




```
class ContactItem extends React.Component {
    render() {
        return (
            <li className='contact-item'>
                <span> Name: {this.props.name} </span>
                <span> Email: {this.props.email} </span>
            </li>
        );
    }
}
class ContactList extends React.Component {
    render() {
        var contacts = this.props.items.map(function(item) {
            return (
                <ContactItem key={item.id}
                             name={item.name}
                             email={item.email} />
            );
        });

        return (
            <ul className='contact-list'>{contacts}</ul>
        );
    }
}
```

Best Practice Example

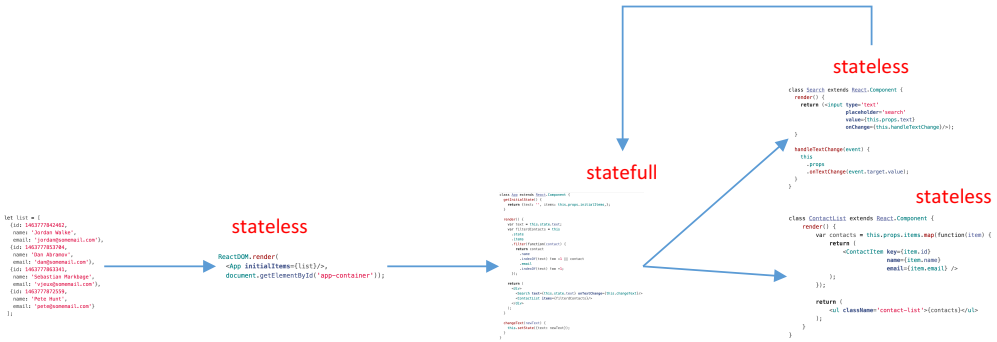
Thus, best practices dictate that applications in React are best designed top-down with a single **stateful** component wrapping child **stateless** components.

Search is a stateless component.

ContactItem is a stateless component.

ContactList is a stateless component.

```
class ContactItem extends React.Component {
    render() {
        return (
            <li className='contact-item'>
                <span> Name: {this.props.name} </span>
                <span> Email: {this.props.email} </span>
            </li>
        );
    }
}
```



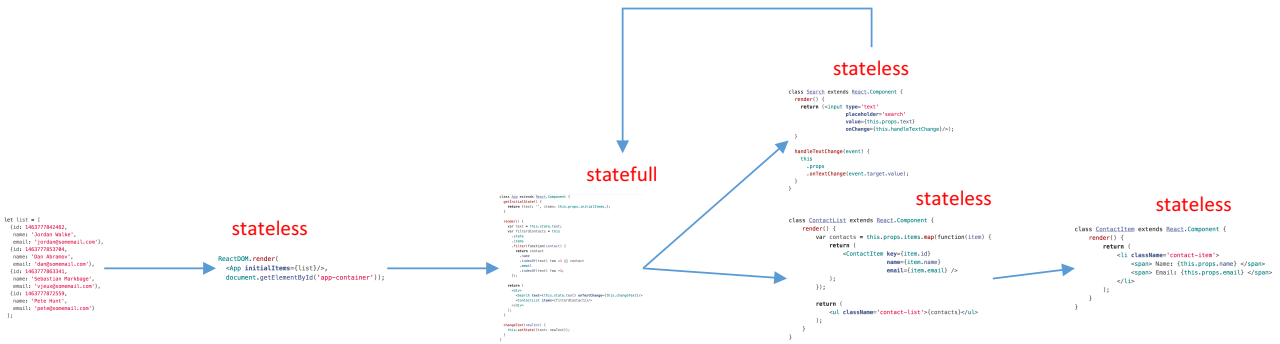
Best Practice Example

Thus, best practices dictate that applications in React are best designed top-down with a single **stateful** component wrapping child **stateless** components.

Search is a stateless component.

ContactItem is a stateless component.

ContactList is a stateless component.



Best Practice Example

```
let list = [
  {id: 1463777842462,
   name: 'Jordan Walke',
   email: 'jordan@somemail.com'},
  {id: 1463777853704,
   name: 'Dan Abramov',
   email: 'dan@somemail.com'},
  {id: 1463777863341,
   name: 'Sebastian Markbage',
   email: 'vjeux@somemail.com'},
  {id: 1463777872559,
   name: 'Pete Hunt',
   email: 'pete@somemail.com'}
 ];
```

stateless

```
ReactDOM.render(
  <App initialItems={list}/>,
  document.getElementById('app-container'));
```

statefull

```
class App extends React.Component {
  getInitialState() {
    return {text: '', items: this.props.initialItems,};
  }

  render() {
    var text = this.state.text;
    var filterdContacts = this
      .state
      .items
      .filter(function(contact) {
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.changeText}/>
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```

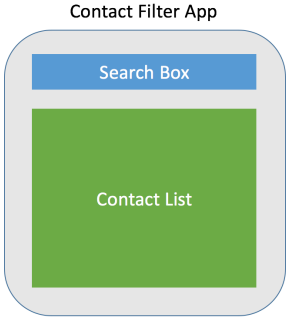
stateless

```
class Search extends React.Component {
  render() {
    return (<input type='text'
                   placeholder='search'
                   value={this.props.text}
                   onChange={this.handleTextChange}/>);
  }

  handleTextChange(event) {
    this
      .props
      .onTextChange(event.target.value);
  }
}
```

stateless

```
class ContactList extends React.Component {
    render() {
        var contacts = this.props.items.map(function(item) {
            return (
                <ContactItem key={item.id}
                             name={item.name}
                             email={item.email} />
            );
        });

        return (
            <ul className='contact-list'>{contacts}</ul>
        );
    }
}
```

Contact Filter App



stateless

```
class ContactItem extends React.Component {
    render() {
        return (
            <li className='contact-item'>
                <span> Name: {this.props.name} </span>
                <span> Email: {this.props.email} </span>
            </li>
        );
    }
}
```

Best Practice Example

```
let list = [
  {id: 1463777842462,
   name: 'Jordan Walke',
   email: 'jordan@somemail.com'},
  {id: 1463777853704,
   name: 'Dan Abramov',
   email: 'dan@somemail.com'},
  {id: 1463777863341,
   name: 'Sebastian Markbage',
   email: 'vjeux@somemail.com'},
  {id: 1463777872559,
   name: 'Pete Hunt',
   email: 'pete@somemail.com'}
];
```

stateless

```
ReactDOM.render(
  <App initialItems={list}/>,
  document.getElementById('app-container'));
```

statefull

```
class App extends React.Component {
  getInitialState() {
    return {text: '', items: this.props.initialItems,};
  }

  render() {
    var text = this.state.text;
    var filterdContacts = this
      .state
      .items
      .filter(function(contact) {
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.changeText}/>
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```

stateless

```
class Search extends React.Component {
  render() {
    return (<input type='text'
                   placeholder='search'
                   value={this.props.text}
                   onChange={this.handleTextChange}/>);
  }

  handleTextChange(event) {
    this
      .props
      .onTextChange(event.target.value);
  }
}
```

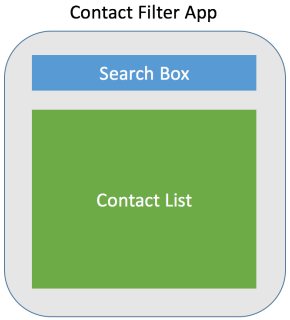
stateless

```
class ContactList extends React.Component {
    render() {
        var contacts = this.props.items.map(function(item) {
            return (
                <ContactItem key={item.id}
                             name={item.name}
                             email={item.email} />
            );
        });

        return (
            <ul className='contact-list'>{contacts}</ul>
        );
    }
}
```

stateless

```
class ContactItem extends React.Component {
    render() {
        return (
            <li className='contact-item'>
                <span> Name: {this.props.name} </span>
                <span> Email: {this.props.email} </span>
            </li>
        );
    }
}
```

These arrows represent the **data flow** through the application.



Best Practice Example

```
let list = [
  {id: 1463777842462,
   name: 'Jordan Walke',
   email: 'jordan@somemail.com'},
  {id: 1463777853704,
   name: 'Dan Abramov',
   email: 'dan@somemail.com'},
  {id: 1463777863341,
   name: 'Sebastian Markbage',
   email: 'vjeux@somemail.com'},
  {id: 1463777872559,
   name: 'Pete Hunt',
   email: 'pete@somemail.com'}
];
```

stateless

```
ReactDOM.render(
  <App initialItems={list}/>,
  document.getElementById('app-container'));
```

statefull

```
class App extends React.Component {
  getInitialState() {
    return {text: '', items: this.props.initialItems,};
  }

  render() {
    var text = this.state.text;
    var filterdContacts = this
      .state
      .items
      .filter(function(contact) {
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.changeText}/>
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```

stateless

```
class Search extends React.Component {
  render() {
    return (<input type='text'
                   placeholder='search'
                   value={this.props.text}
                   onChange={this.handleTextChange}/>);
  }

  handleTextChange(event) {
    this
      .props
      .onTextChange(event.target.value);
  }
}
```

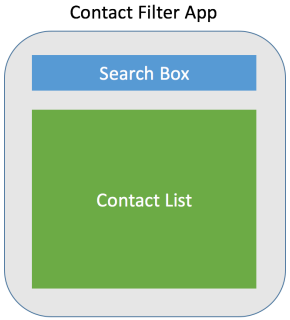
stateless

```
class ContactList extends React.Component {
    render() {
        var contacts = this.props.items.map(function(item) {
            return (
                <ContactItem key={item.id}
                             name={item.name}
                             email={item.email} />
            );
        });

        return (
            <ul className='contact-list'>{contacts}</ul>
        );
    }
}
```

stateless

```
class ContactItem extends React.Component {
    render() {
        return (
            <li className='contact-item'>
                <span> Name: {this.props.name} </span>
                <span> Email: {this.props.email} </span>
            </li>
        );
    }
}
```

This arrow represents the **control flow**.



Component Life-Cycle – What if?

Think a little about what a
React component does…

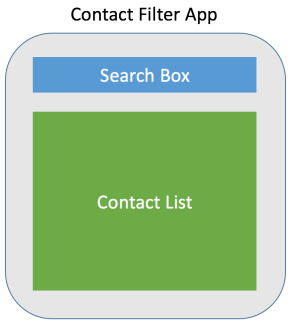
```
class App extends React.Component {
  getInitialState() {
    return {text: '', items: this.props.initialItems,};
  }

  render() {
    var text = this.state.text;
    var filterdContacts = this
      .state
      .items
      .filter(function(contact) {
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.changeText}/>
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```

Component Life-Cycle – What if?

Think a little about what a React component does…

Based on what we know, it describes what to render.

```
class App extends React.Component {
  getInitialState() {
    return {text: '', items: this.props.initialItems,};
  }

  render() {
    var text = this.state.text;
    var filterdContacts = this
      .state
      .items
      .filter(function(contact) {
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.changeText}/>
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```



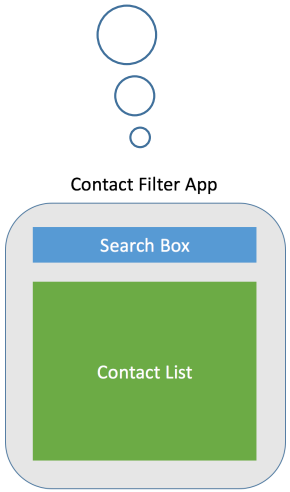
Component Life-Cycle – What if?

Think a little about what a React component does…

Based on what we know describes what to render.

What if we want to do something before or after the component has rendered?

```
class App extends React.Component {
  getInitialState() {
    return {text: '', items: this.props.initialItems,};
  }

  render() {
    var text = this.state.text;
    var filterdContacts = this
      .state
      .items
      .filter(function(contact) {
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.changeText}/>
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```

Component Life-Cycle – What if?

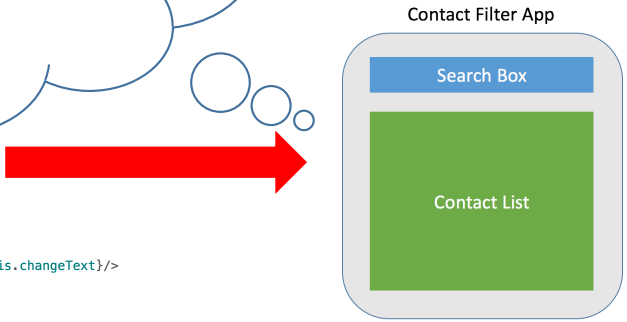
Think a little about what a React component does…

Based on what we know describes what to render.

```
class App exte
  getIniti
    retu
  }

  render()
    var te
    var f
      .sta
      .items
      .filter(funct
        return contact
          .name
          .indexOf(text) !== -1 || contact
          .email
          .indexOf(text) !== -1;
      });

    return (
      <div>
        <Search text={this.state.text} onTextChange={this.changeText}/>
        <ContactList items={filterdContacts}/>
      </div>
    );
  }

  changeText(newText) {
    this.setState({text: newText});
  }
}
```

What if we want to do something before or after the component has rendered?

What if we want to avoid rendering all together?



Component Life-Cycle Definition

- Looks like we need more control over the **stages** a component goes through.
- The process where all these stages are involved is called the **components life-cycle** and every React component goes through it.
- React provides several methods that notify us when a certain stage of the process occurs.
- These methods are called the **component's life-cycle methods** – these methods are invoked in a predictable order.

Component Life-Cycle: Initialization

Initialization

The initialization phase is where we define defaults and initial values **this.props** and **this.state**.

Component Life-Cycle: Initialization

```
import React from 'react';

export default class Counter extends React.Component {
  getDefaultProps() {
    return {title: 'Basic counter!!!'}
  }

  getInitialState() {
    return {count: 0}
  }

  render() {
    return (
      <div>
        <h1>{this.props.title}</h1>
        <div>{this.state.count}</div>
        <input type='button' value='+' onClick={this.handleIncrement}/>
        <input type='button' value='-' onClick={this.handleDecrement}/>
      </div>
    );
  }

  handleIncrement() {
    var newCount = this.state.count + 1;
    this.setState({count: newCount});
  }

  handleDecrement() {
    var newCount = this.state.count - 1;
    this.setState({count: newCount});
  }
}
```

The initialization phase is where we define defaults and initial values for **this.props** and **this.state**.

Component Life-Cycle: Initialization

```
import React from 'react';

export default class Counter extends React.Component {
  getDefaultProps() {
    return {title: 'Basic counter!!!'}
  }

  getInitialState() {
    return {count: 0}
  }

  render() {
    return (
      <div>
        <h1>{this.props.title}</h1>
        <div>{this.state.count}</div>
        <input type='button' value='+' onClick={this.handleIncrement}/>
        <input type='button' value='-' onClick={this.handleDecrement}/>
      </div>
    );
  }

  handleIncrement() {
    var newCount = this.state.count + 1;
    this.setState({count: newCount});
  }

  handleDecrement() {
    var newCount = this.state.count - 1;
    this.setState({count: newCount});
  }
}
```

The initialization phase is where we define defaults and initial values for **this.props** and **this.state**.

We do this by implementing two methods:

Component Life-Cycle: Initialization

```
import React from 'react';

export default class Counter extends React.Component {
  getDefaultProps() {
    return {title: 'Basic counter!!!'}
  }

  getInitialState() {
    return {count: 0}
  }

  render() {
    return (
      <div>
        <h1>{this.props.title}</h1>
        <div>{this.state.count}</div>
        <input type='button' value='+' onClick={this.handleIncrement}/>
        <input type='button' value='-' onClick={this.handleDecrement}/>
      </div>
    );
  }

  handleIncrement() {
    var newCount = this.state.count + 1;
    this.setState({count: newCount});
  }

  handleDecrement() {
    var newCount = this.state.count - 1;
    this.setState({count: newCount});
  }
}
```

The initialization phase is where we define defaults and initial values for **this.props** and **this.state**.

We do this by implementing two methods:

- **`getDefaultProps()`**
  This method is called once (when the class is created) and cached. It is also shared across instances of this component. This methods returns an object indicating which property values will be set on **this.props** - if the prop is not specified by the parent component.

Component Life-Cycle: Initialization

```
import React from 'react';

export default class Counter extends React.Component {
  getDefaultProps() {
    return {title: 'Basic counter!!!'}
  }

  getInitialState() {
    return {count: 0}
  }

  render() {
    return (
      <div>
        <h1>{this.props.title}</h1>
        <div>{this.state.count}</div>
        <input type='button' value='+' onClick={this.handleIncrement}/>
        <input type='button' value='-' onClick={this.handleDecrement}/>
      </div>
    );
  }

  handleIncrement() {
    var newCount = this.state.count + 1;
    this.setState({count: newCount});
  }

  handleDecrement() {
    var newCount = this.state.count - 1;
    this.setState({count: newCount});
  }
}
```

The initialization phase is where we define defaults and initial values for **this.props** and **this.state**.

We do this by implementing two methods:

- **`getDefaultProps()`**
  This method is called once (when the class is created) and cached. It is also shared across instances of this component. This methods returns an object indicating which property values will be set on **this.props** - if the prop is not specified by the parent component.

- **`getInitialState()`**
  This method is invoked once, right before the **mounting phase**. The return value of this method will be used as an initial value of **this.state** and must be an object.

Component Life-Cycle: Initialization

```
import React from 'react';

export default class Counter extends React.Component {
  getDefaultProps() {
    return {title: 'Basic counter!!!'}
  }

  getInitialState() {
    return {count: 0}
  }

  render() {
    return (
      <div>
        <h1>{this.props.title}</h1>
        <div>{this.state.count}</div>
        <input type='button' value='+' onClick={this.handleIncrement}/>
        <input type='button' value='-' onClick={this.handleDecrement}/>
      </div>
    );
  }

  handleIncrement() {
    var newCount = this.state.count + 1;
    this.setState({count: newCount});
  }

  handleDecrement() {
    var newCount = this.state.count - 1;
    this.setState({count: newCount});
  }
}
```

The rest of this component is implemented in a similar fashion as to other components we have seen at this point.

Component Life-Cycle: Initialization

```
import React from 'react';

export default class Counter extends React.Component {
  getDefaultProps() {
    return {title: 'Basic counter!!!'}
  }

  getInitialState() {
    return {count: 0}
  }

  render() {
    return (
      <div>
        <h1>{this.props.title}</h1>
        <div>{this.state.count}</div>
        <input type='button' value='+' onClick={this.handleIncrement}/>
        <input type='button' value='-' onClick={this.handleDecrement}/>
      </div>
    );
  }

  handleIncrement() {
    var newCount = this.state.count + 1;
    this.setState({count: newCount});
  }

  handleDecrement() {
    var newCount = this.state.count - 1;
    this.setState({count: newCount});
  }
}
```

Here is a refactoring of the original to further explore how initialization can be used to improve the design of components.

```
import React from 'react';

export default class Counter extends React.Component {
  getDefaultProps() {
    return { title: 'Basic counter!!!', step: 1 }
  }

  getInitialState() {
    return {count: (this.props.initialCount || 0)}
  }

  render() {
    let step = this.props.step;
    return (
      <div>
        <h1>{this.props.title}</h1>
        <div>{this.state.count}</div>
        <input type='button' value='+'
               onClick={() => this.updateCounter(step)}/>
        <input type='button' value='-'
               onClick={() => this.updateCounter(-step)}/>
      </div>
    );
  }

  updateCounter(amount) {
    let newCount = this.state.count + amount;
    this.setState({ count : newCount });
  }
}
```

Component Life-Cycle: Initialization

```
import React from 'react';

export default class Counter extends React.Component {
  getDefaultProps() {
    return {title: 'Basic counter!!!'}
  }

  getInitialState() {
    return {count: 0}
  }

  render() {
    return (
      <div>
        <h1>{this.props.title}</h1>
        <div>{this.state.count}</div>
        <input type='button' value='+' onClick={this.handleIncrement}/>
        <input type='button' value='-' onClick={this.handleDecrement}/>
      </div>
    );
  }

  handleIncrement() {
    var newCount = this.state.count + 1;
    this.setState({count: newCount});
  }

  handleDecrement() {
    var newCount = this.state.count - 1;
    this.setState({count: newCount});
  }
}
```

Here is a refactoring of the original to further explore how initialization can be used to improve the design of components.

```
import React from 'react';

export default class Counter extends React.Component {
  getDefaultProps() {
    return { title: 'Basic counter!!!', step: 1 }
  }

  getInitialState() {
    return {count: (this.props.initialCount || 0)}
  }

  render() {
    let step = this.props.step;
    return (
      <div>
        <h1>{this.props.title}</h1>
        <div>{this.state.count}</div>
        <input type='button' value='+'
               onClick={() => this.updateCounter(step)}/>
        <input type='button' value='-'
               onClick={() => this.updateCounter(-step)}/>
      </div>
    );
  }

  updateCounter(amount) {
    let newCount = this.state.count + amount;
    this.setState({ count : newCount });
  }
}
```

Component Life-Cycle: Initialization

```
import React from 'react';

export default class Counter extends React.Component {
  getDefaultProps() {
    return {title: 'Basic counter!!!'}
  }

  getInitialState() {
    return {count: 0}
  }

  render() {
    return (
      <div>
        <h1>{this.props.title}</h1>
        <div>{this.state.count}</div>
        <input type='button' value='+' onClick={this.handleIncrement}/>
        <input type='button' value='-' onClick={this.handleDecrement}/>
      </div>
    );
  }

  handleIncrement() {
    var newCount = this.state.count + 1;
    this.setState({count: newCount});
  }

  handleDecrement() {
    var newCount = this.state.count - 1;
    this.setState({count: newCount});
  }
}
```

Here is a refactoring of the original to further explore how initialization can be used to improve the design of components.

```
import React from 'react';

export default class Counter extends React.Component {
  getDefaultProps() {
    return { title: 'Basic counter!!!', step: 1 }
  }

  getInitialState() {
    return {count: (this.props.initialCount || 0)}
  }

  render() {
    let step = this.props.step;
    return (
      <div>
        <h1>{this.props.title}</h1>
        <div>{this.state.count}</div>
        <input type='button' value='+'
               onClick={() => this.updateCounter(step)}/>
        <input type='button' value='-'
               onClick={() => this.updateCounter(-step)}/>
      </div>
    );
  }

  updateCounter(amount) {
    let newCount = this.state.count + amount;
    this.setState({ count : newCount });
  }
}
```

Component Life-Cycle: Initialization

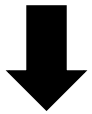
```
import React from 'react';

export default class Counter extends React.Component {
  getDefaultProps() {
    return {title: 'Basic counter!!!'}
  }

  getInitialState() {
    return {count: 0}
  }

  render() {
    return (
      <div>
        <h1>{this.props.title}</h1>
        <div>{this.state.count}</div>
        <input type='button' value='+' onClick={this.handleIncrement}/>
        <input type='button' value='-' onClick={this.handleDecrement}/>
      </div>
    );
  }

  handleIncrement() {
    var newCount = this.state.count + 1;
    this.setState({count: newCount});
  }

  handleDecrement() {
    var newCount = this.state.count - 1;
    this.setState({count: newCount});
  }
}
```

Here is a refactoring of the original to further explore how initialization can be used to improve the design of components.

```
import React from 'react';

export default class Counter extends React.Component {
  getDefaultProps() {
    return { title: 'Basic counter!!!', step: 1 }
  }

  getInitialState() {
    return {count: (this.props.initialCount || 0)}
  }

  render() {
    let step = this.props.step;
    return (
      <div>
        <h1>{this.props.title}</h1>
        <div>{this.state.count}</div>
        <input type='button' value='+'
               onClick={() => this.updateCounter(step)}/>
        <input type='button' value='-'
               onClick={() => this.updateCounter(-step)}/>
      </div>
    );
  }

  updateCounter(amount) {
    let newCount = this.state.count + amount;
    this.setState({ count : newCount });
  }
}
```

Component Life-Cycle: Initialization

Initialization

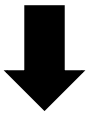


Mounting

Mounting is the process that occurs when a component is being inserted into the DOM.

Component Life-Cycle: Initialization

Initialization



Mounting

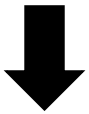
Mounting is the process that occurs when a component is being inserted into the DOM. There are two methods for this:

componentWillMount(): This method is invoked once and immediately before the rendering process, hence before React inserts the component into the DOM. Calling this.setState in this method has not effect.

componentDidMount(): This method is invoked once and immediately after React inserts the component into the DOM.

Component Life-Cycle: Initialization

Initialization



Mounting

Mounting is the process that occurs when a component is being inserted into the DOM. There are two methods for this:

componentWillMount(): This method is invoked once and immediately before the rendering process, hence before React inserts the component into the DOM. Calling this.setState in this method has not effect.

This method is useful because we are assured that the updated DOM is fully constructed.

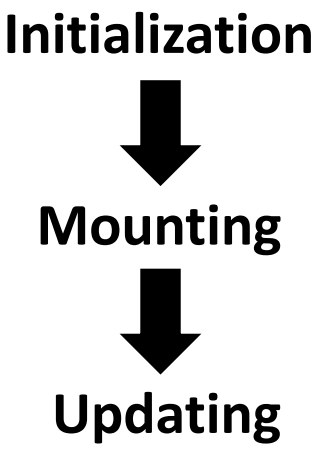
This method is most useful to use 3rd party libraries that need to access the DOM or for **fetching data** from a remote server.

componentDidMount(): This method is invoked once and immediately after React inserts the component into the DOM.

Component Life-Cycle: Mounting

```
import React from 'react';
import Counter from './Counter';
import Axios from './Axios'; // this is just a placeholder

export default class Container extends React.Component {
  getInitialState() {
       return { data: null, fetching: false, error: null };
   }

   render() {
       if (this.props.fetching) {
           return <div>Loading...</div>;
       }

       if (this.props.error) {
           return (
               <div className='error'>
                   {this.state.error.message}
               </div>
           );
       }

       let data = this.state.data;

       return <Counter initialCount={data.initialCount} step={data.step} />
   }

   componentDidMount() {
       this.setState({fetching: true});

       Axios.get(this.props.url).then(function(res) {
           this.setState({data: res.data, fetching: false});
       }).catch(function(res) {
           this.setState({error: res.data, fetching: false});
       });
   }
}
```

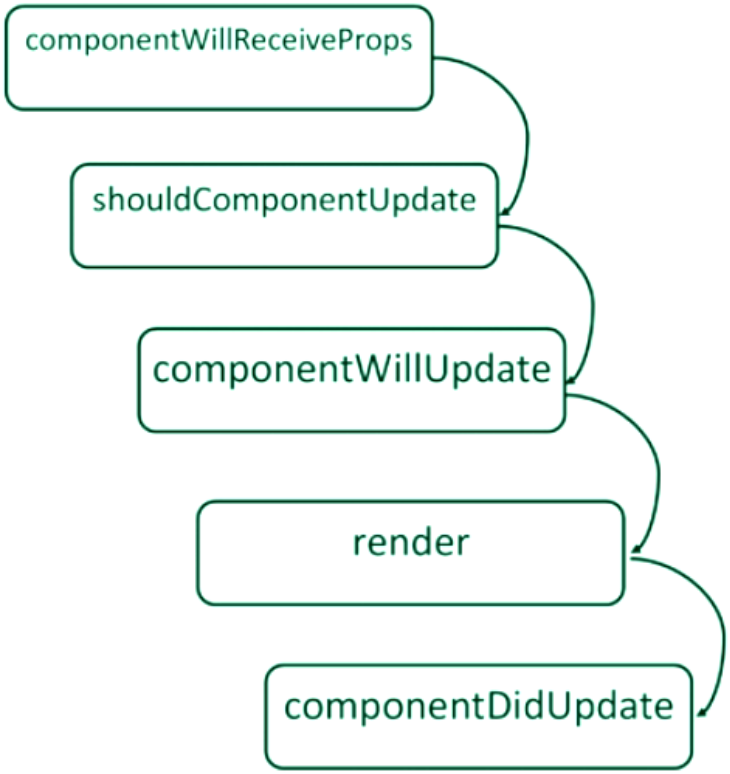
Component Life-Cycle: Updating



Updating is the process for updating a component. It will allow us to execute code relative to when a component's state or properties get updated.

There are methods that are part of the updating phas that are called in a particular order.

Component Life-Cycle: Updating



These methods are executed when a component receives new props from a parent (container) component.

Initialization

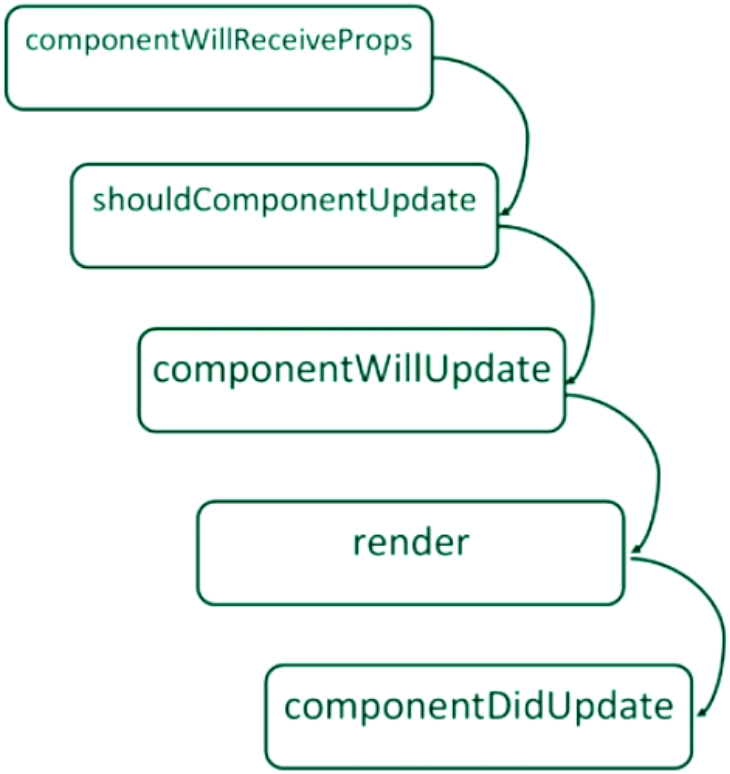
↓

Mounting

↓

Updating

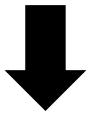
Component Life-Cycle: Updating



These methods are executed when a component receives new props from a parent (container) component.

componentWillReceiveProps is invoked when a component is receiving new props.

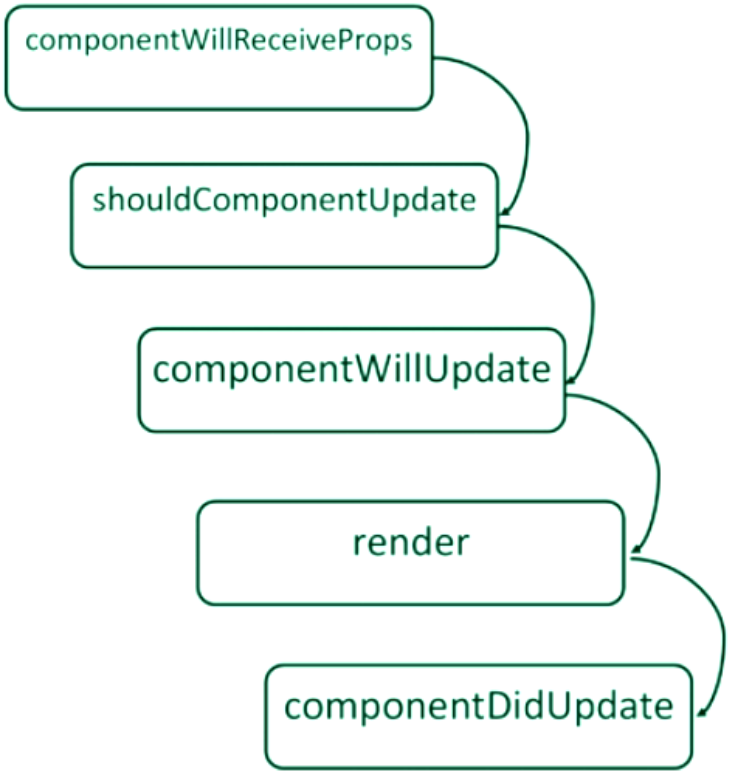
Initialization



Mounting

Updating

Component Life-Cycle: Updating

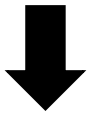


These methods are executed when a component receives new props from a parent (container) component.

componentWillReceiveProps is invoked when a component is receiving new props.

shouldComponentUpdate allows us to decide whether the next component's state should trigger a re-render. This returns a boolean and is used for optimization.

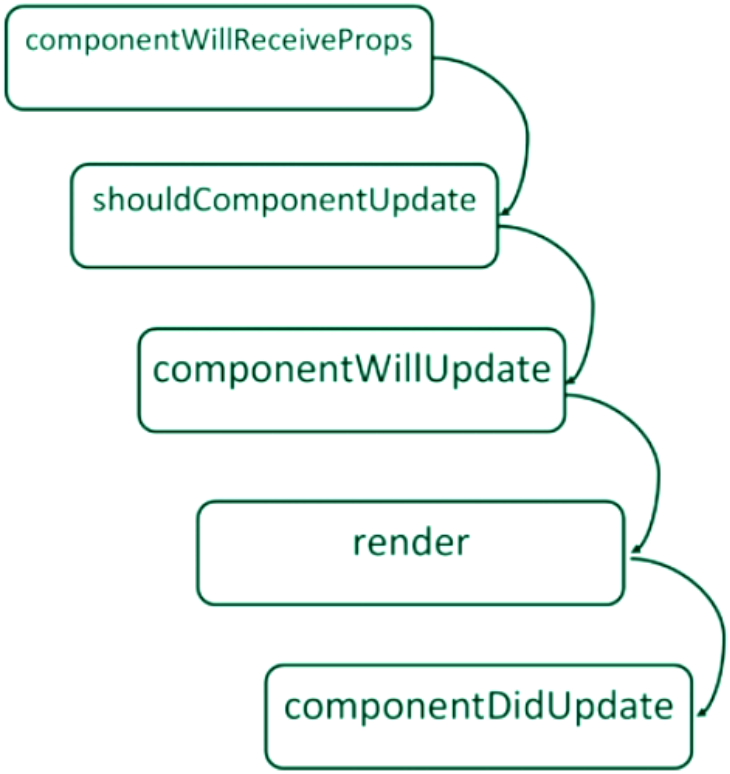
Initialization



Mounting

Updating

Component Life-Cycle: Updating

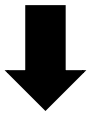


These methods are executed when a component receives new props from a parent (container) component.

componentWillUpdate is called immediately before rendering, when new props or state are being received.

We can use this as an opportunity to perform preparation before an update occurs, however, it is not allowed to use **this.setState**.

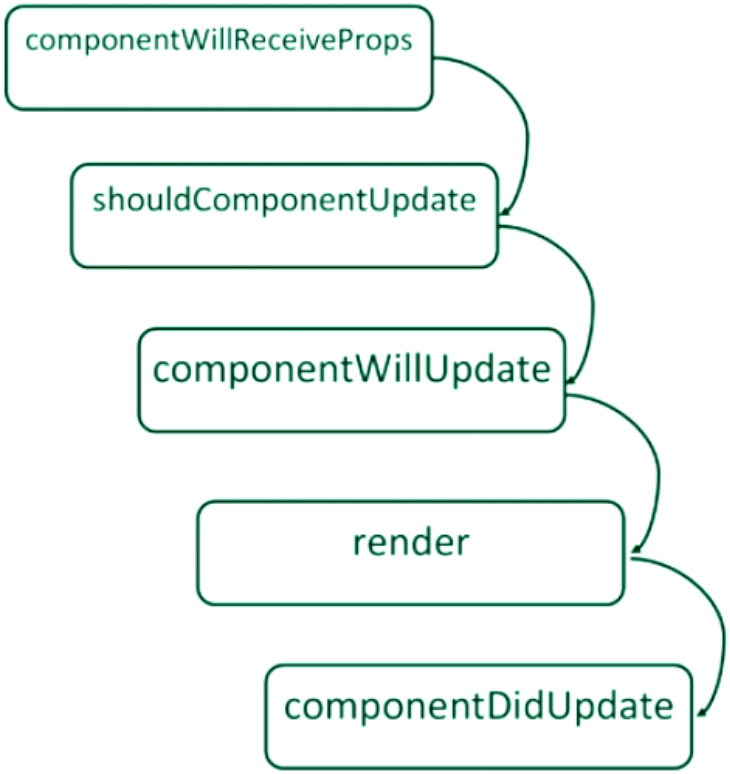
Initialization



Mounting

Updating

Component Life-Cycle: Updating



These methods are executed when a component receives new props from a parent (container) component.

componentDidUpdate is called immediately after React updates the DOM. We can use this method to perform any action post-rendering.

This method gets two arguments:

1. prevProps: the previous props
2. prevState: the previous state

Initialization

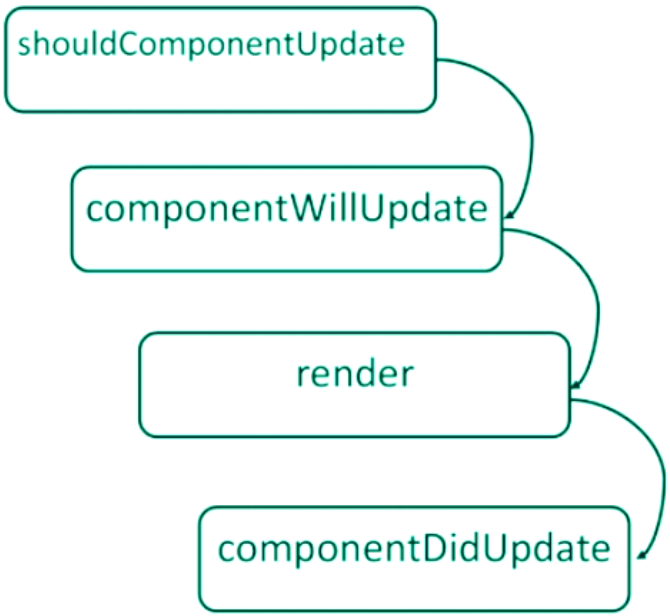
↓

Mounting

↓

Updating

Component Life-Cycle: Updating



These methods are executed when a component receives new state when the **this.setState** method is called.

Initialization

↓

Mounting

↓

Updating