COMPSCI 326
Web Programming

React State and Interactivity

Objectives

- Understand React State
- Understand React Interactivity

React Props

React has a one-way data flow beginning from the top-most component and working its way down through each of the sub-components.

The way we communicate data in this tree of components is by passing the data through **props**.

We can then use **props** to populate a UI component with the data it needs to render.

Facebook Comment

```
.
├── README.md
├── app.js
├── components
│   ├── comment.js
│   ├── commententry.js
│   ├── commentthread.js
│   ├── feed.js
│   ├── feeditem.js
│   ├── statusupdate.js
│   └── statusupdateentry.js
├── database.js
├── server.js
└── util.js
```

```
export default class Comment extends React.Component {
  render() {
    return (
      <div>
        <div className="media-left media-top">
          PIC
        </div>
        <div className="media-body">
          <a href="#">{this.props.author}</a> {this.props.children}
          <br /><a href="#">Like</a> · <a href="#">Reply</a> ·
            {this.props.postDate}
        </div>
      </div>
    )
  }
}
```

We pass along data using props.

```
<Comment author="Someone Else" postDate="20 hrs">
  hope everything is ok!</Comment>
```

Facebook Comment

```
.
├── README.md
├── app.js
├── components
│   ├── comment.js
│   ├── commententry.js
│   ├── commentthread.js
│   ├── feed.js
│   ├── feeditem.js
│   ├── statusupdate.js
│   └── statusupdateentry.js
├── database.js
├── server.js
└── util.js
```

```
export default class Comment extends React.Component {
  render() {
    return (
      <div>
        <div className="media-left media-top">
          PIC
        </div>
        <div className="media-body">
          <a href="#">{this.props.author}</a> {this.props.children}
          <br /><a href="#">Like</a> · <a href="#">Reply</a> ·
            {this.props.postDate}
        </div>
      </div>
    )
  }
}
```

We pass along data using props.

```
<Comment author="Someone Else" postDate="20 hrs">
  hope everything is ok!</Comment>
```

Facebook Comment

```
.
├── README.md
├── app.js
├── components
│   ├── comment.js
│   ├── commententry.js
│   ├── commentthread.js
│   ├── feed.js
│   ├── feeditem.js
│   ├── statusupdate.js
│   └── statusupdateentry.js
├── database.js
├── server.js
└── util.js
```

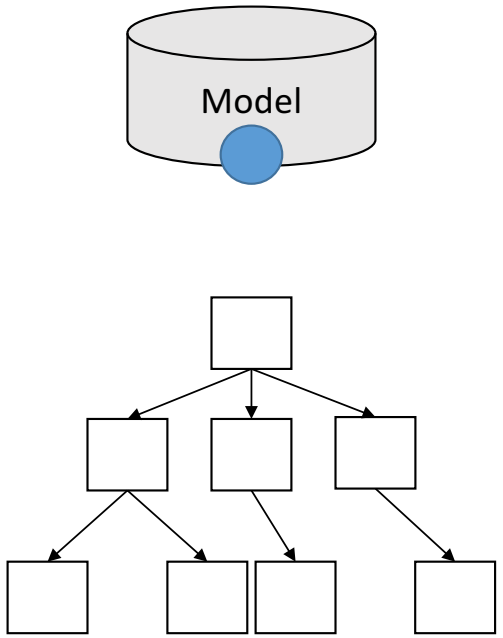
```
export default class Comment extends React.Component {
  render() {
    return (
      <div>
        <div className="media-left media-top">
          PIC
        </div>
        <div className="media-body">
          <a href="#">{this.props.author}</a> {this.props.children}
          <br /><a href="#">Like</a> · <a href="#">Reply</a> ·
            {this.props.postDate}
        </div>
      </div>
    )
  }
}
```

We pass along data using props.

```
<Comment author="Someone Else" postDate="20 hrs">
  hope everything is ok!</Comment>
```

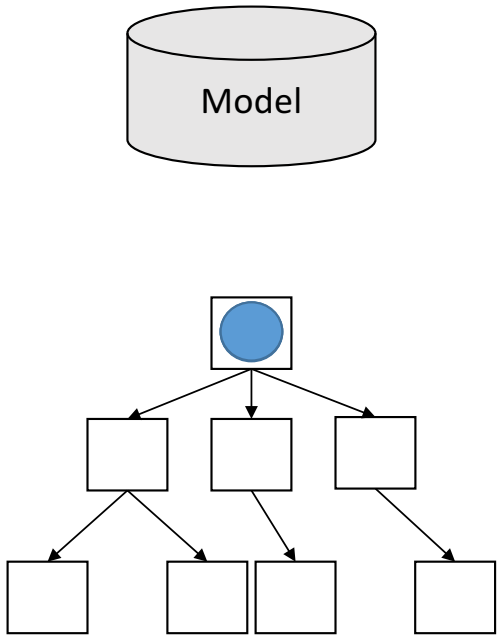
Changing Data

- Data is pushed through the react component tree from the model.



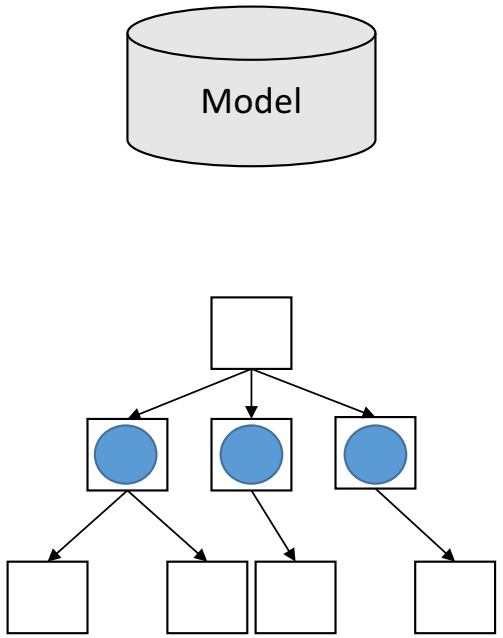
Changing Data

- Data is pushed through the react component tree from the model.



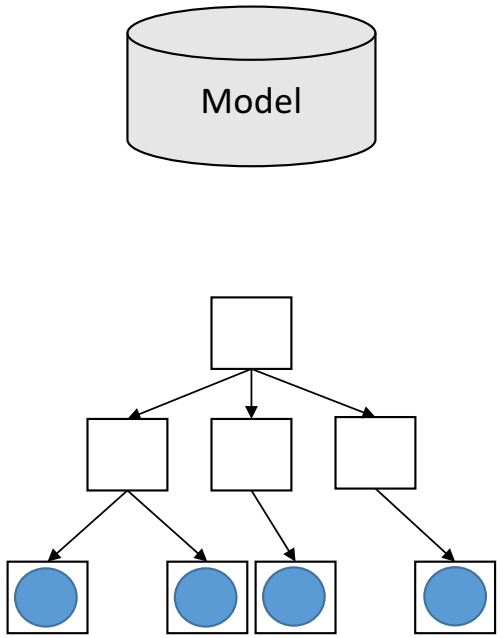
Changing Data

- Data is pushed through the react component tree from the model.



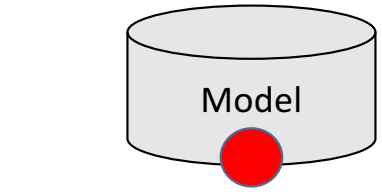
Changing Data

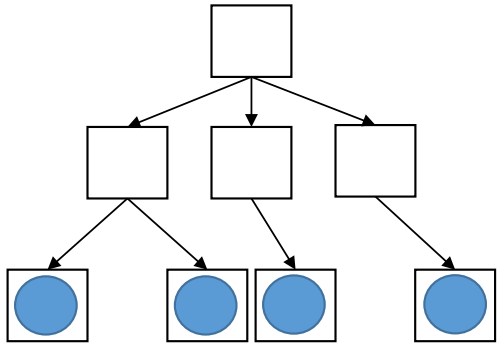
- Data is pushed through the react component tree from the model.



Changing Data

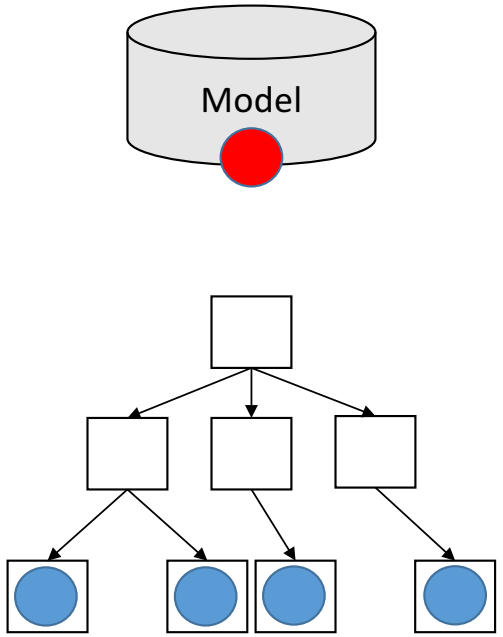
- Data is pushed through the react component tree from the model.
- What if our model changes?





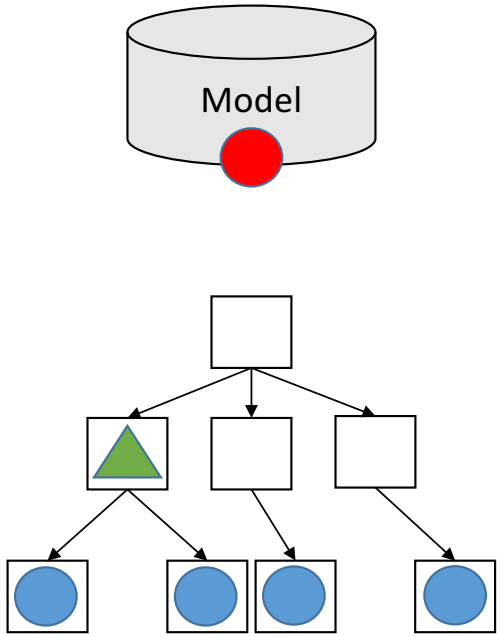
Changing Data

- Data is pushed through the react component tree from the model.
- What if our model changes?
- How do we re-render the React components to reflect those changes?



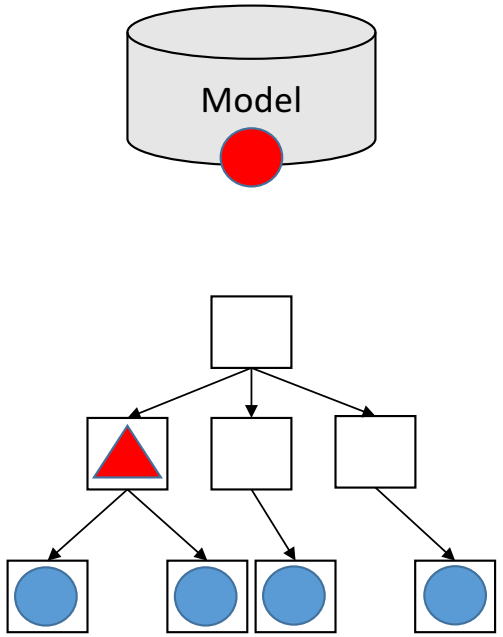
React State

- In addition to **props**, React components may also maintain **state**.



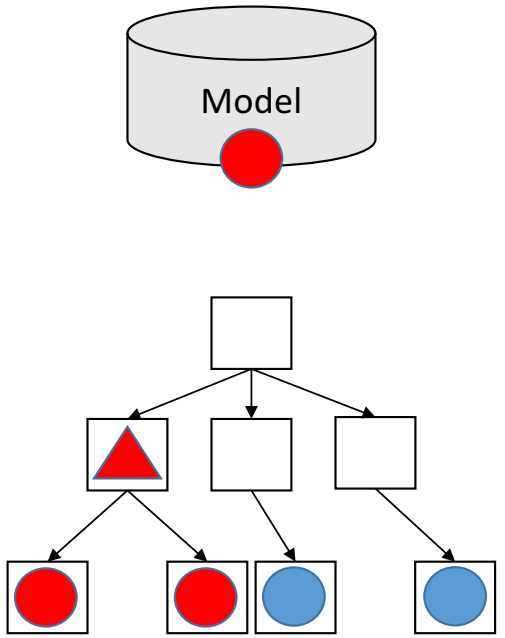
React State

- In addition to **props**, React components may also maintain **state**.
- If the model changes and the state in a component is updated to reflect that change, it will cause a React component to re-render.

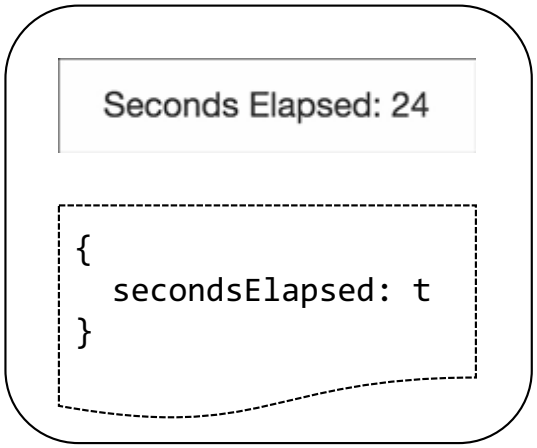


React State

- In addition to **props**, React components may also maintain **state**.
- If the model changes and the state in a component is updated to reflect that change, it will cause a React component to re-render.
- Which will cause each child component to be re-instantiated with new **prop** values.



A Timer Application



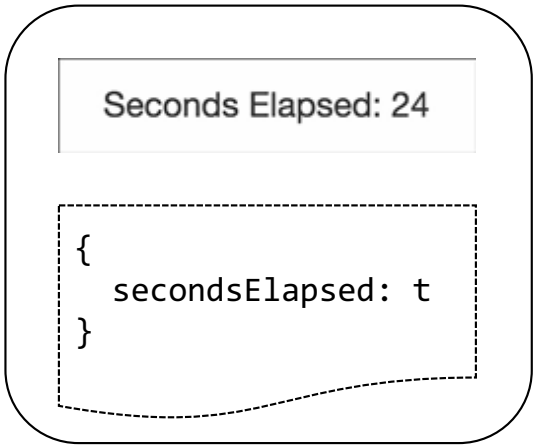
This is a simple Timer application.

After each second the component's view is updated to report the number of elapsed seconds.

The data associated with that view is stored as part of the component's **state**.

The **state** and the view are tied together by firing an event every second that updates the state of the Timer component.

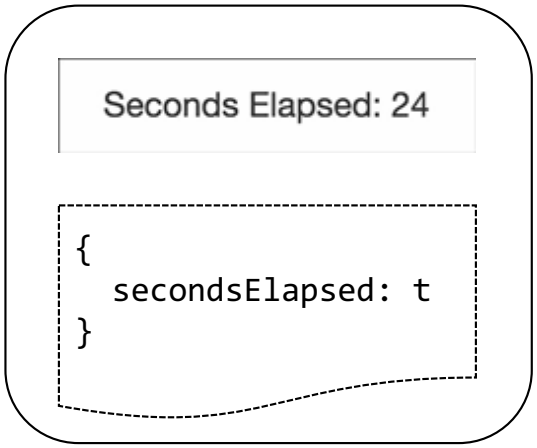
A Timer Application




```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

```
import React from 'react';

export default class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {secondsElapsed: 0};
  }

  tick() {
    this.setState((prevState) => ({
      secondsElapsed: prevState.secondsElapsed + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}
```

A Timer Application

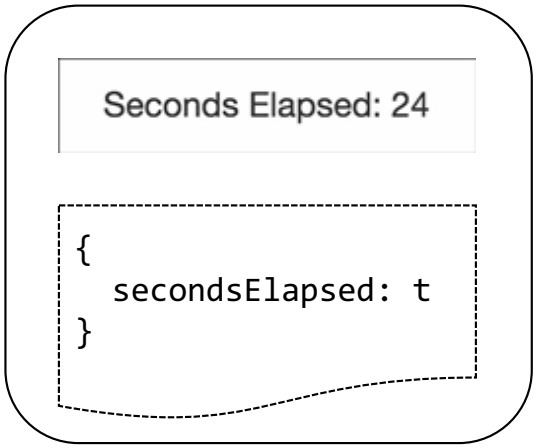



```
import React from 'react';

export default class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {secondsElapsed: 0};
  }

  tick() {
    this.setState((prevState) => ({
      secondsElapsed: prevState.secondsElapsed + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}
```

Like all of our other React components we begin by importing React and creating a new class that extends **React.Component**.

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

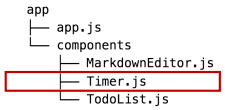
A Timer Application



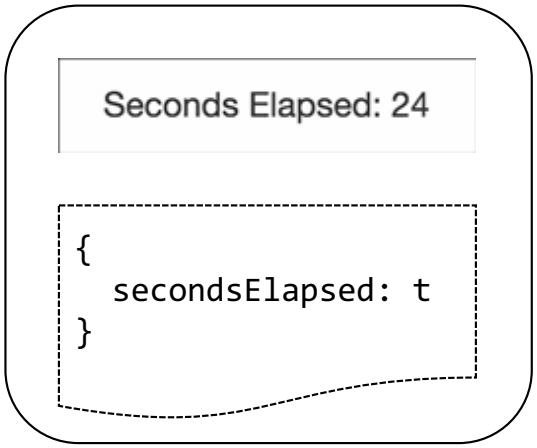

```
import React from 'react';

export default class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {secondsElapsed: 0};
  }

  tick() {
    this.setState((prevState) => ({
      secondsElapsed: prevState.secondsElapsed + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}
```

Like all of our other React components we begin by importing React and creating a new class that extends **React.Component**.

We define a constructor for initializing the props and setting **this.state** – a *special* variable to maintain the state of a component.



A Timer Application

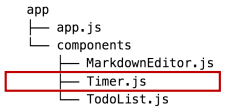



```
import React from 'react';

export default class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {secondsElapsed: 0};
  }

  tick() {
    this.setState((prevState) => ({
      secondsElapsed: prevState.secondsElapsed + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}
```

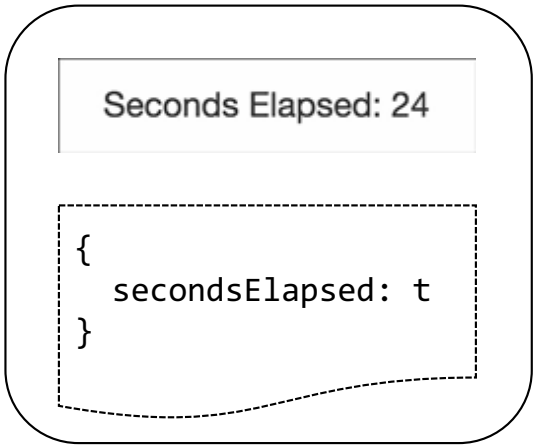
Like all of our other React components we begin by importing React and creating a new class that extends **React.Component**.

We define a constructor for initializing the props and setting **this.state** – a *special* variable to maintain the state of a component.

In this case, our component's initial state is an object with a single property **secondsElapsed** with an initial value of 0.



A Timer Application

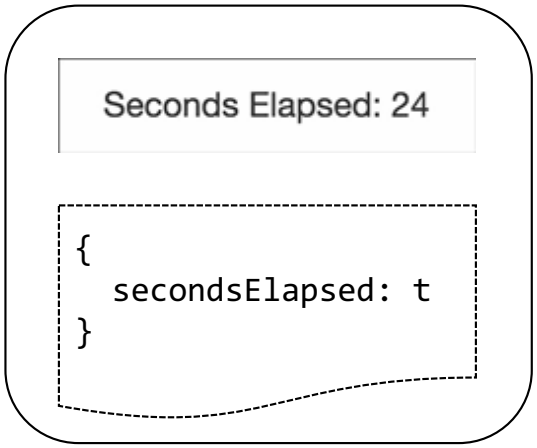



```
import React from 'react';

export default class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {secondsElapsed: 0};
  }

  tick() {
    this.setState((prevState) => ({
      secondsElapsed: prevState.secondsElapsed + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}
```

The **render** method is straightforward.

It simply returns a containing the number of seconds that have elapsed since the component was mounted in the view.

A Timer Application

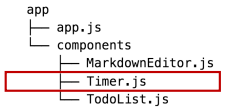



```
import React from 'react';

export default class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {secondsElapsed: 0};
  }

  tick() {
    this.setState((prevState) => ({
      secondsElapsed: prevState.secondsElapsed + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}
```

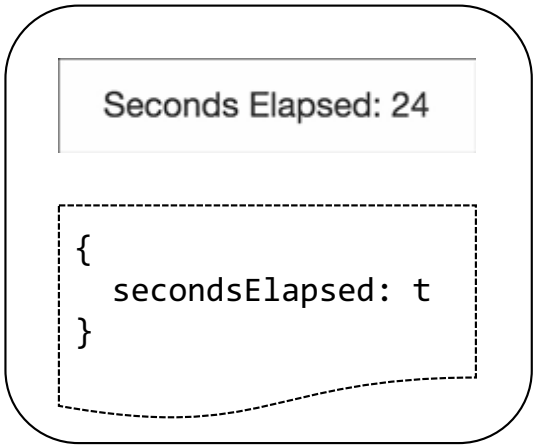
The **render** method is straightforward.

It simply returns a containing the number of seconds that have elapsed since the component was mounted in the view.

We use the **state** of the component to display the number of elapsed seconds.



A Timer Application




```
import React from 'react';

export default class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {secondsElapsed: 0};
  }

  tick() {
    this.setState((prevState) => ({
      secondsElapsed: prevState.secondsElapsed + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}
```

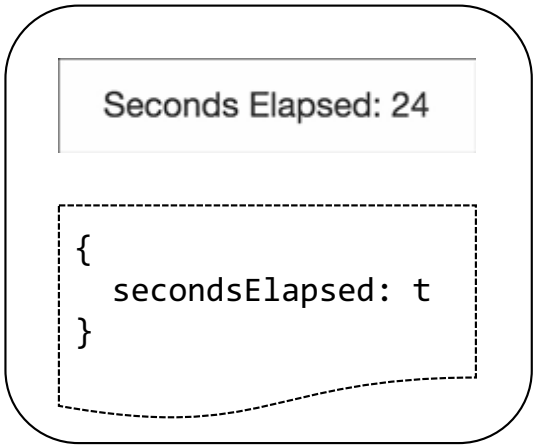
The **render** method is straightforward.

It simply returns a containing the number of seconds that have elapsed since the component was mounted in the view.

We use the **state** of the component to display the number of elapsed seconds.

But, how do we update the state of the component and update the corresponding view?

A Timer Application

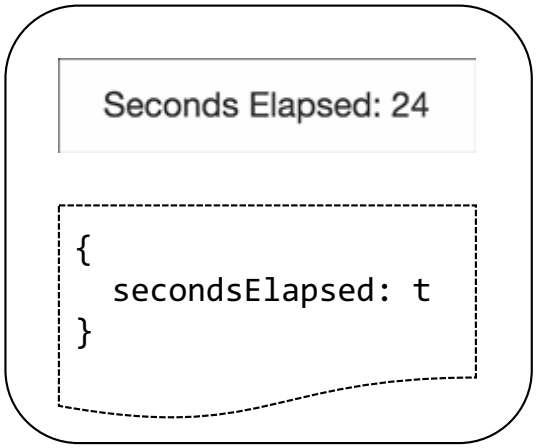



```
import React from 'react';

export default class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {secondsElapsed: 0};
  }

  tick() {
    this.setState((prevState) => ({
      secondsElapsed: prevState.secondsElapsed + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}
```

We define a **tick** method.

The **tick** method sets the **state** of the component to a new state whose **secondsElapsed** property is incremented by 1 from the previous value.

A Timer Application




```
import React from 'react';

export default class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {secondsElapsed: 0};
  }

  tick() {
    this.setState((prevState) => ({
      secondsElapsed: prevState.secondsElapsed + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}
```

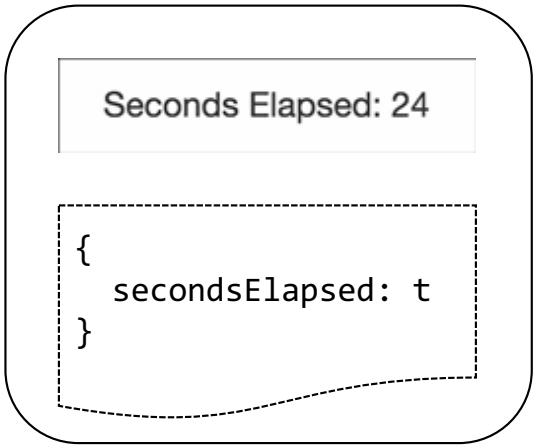
We define a **tick** method.

The **tick** method sets the **state** of the component to a new state whose **secondsElapsed** property is incremented by 1 from the previous value.

The **this.setState** method is special. It takes a function as its argument which accepts a single parameter – the previous state.

This function then returns the new state given the old state.

A Timer Application



But, how will this update the view rendered to the user?

```
import React from 're

export default class T
  constructor(props)
    super(props);
    this.state = {secondsEl
  }

  tick() {
    this.setState((prevState) => ({
      secondsElapsed: prevState.secondsElapsed + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}
```

We define a **tick** method.

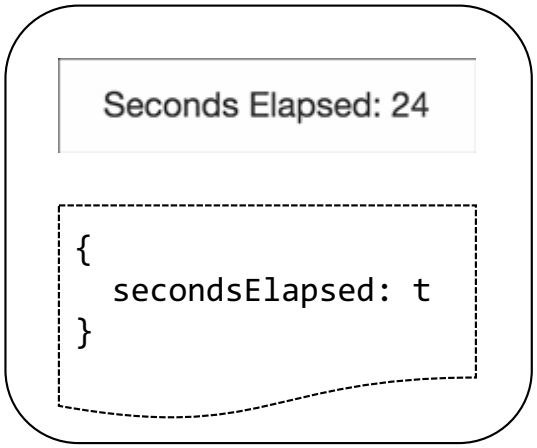
The **tick** method sets the **state** of the component to a new state whose **secondsElapsed** property is incremented by 1 from the previous value.

The **this.setState** method is special. It takes a function as its argument which accepts a single parameter – the previous state.

This function then returns the new state given the old state.

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

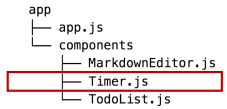
A Timer Application



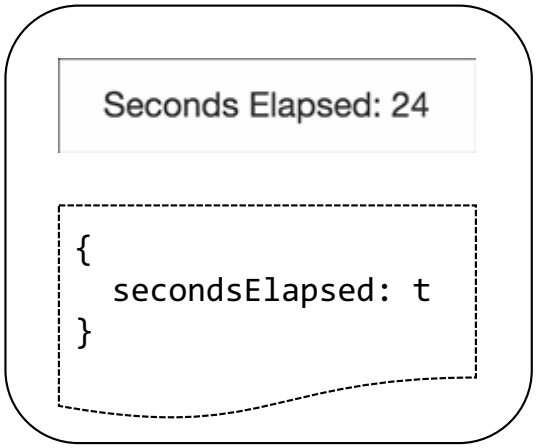
But, how will this update the view rendered to the user?

```
import React from 're

export default class
  constructor(props)
    super(props);
    this.state = {secondsEl
  }

  tick() {
    this.setState((prevState) => ({
      secon lapsed: prevState.secondsElapsed + 1
    }));
  }

  componentD Mount() {
    this.int val = setInterval(() => this.tick(), 1000);
  }

  componentW lUnmount() {
    clearInt val(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}
```

It turns out that when **this.setState** is invoked, React will update the state and *automatically* call the **render** method to re-render the component.



A Timer Application



But, how will this update the view rendered to the user?

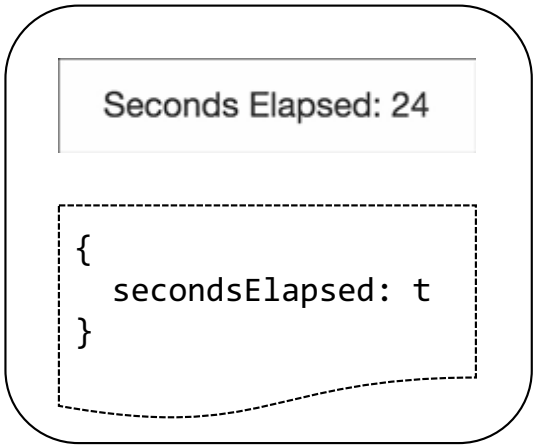
```
import React from 're

export default class
  constructor(props)
    super(props);
    this.state = {secondsE
  }

  tick() {
    this.setState((prevState) => ({
      secon lapsed: prevState.secondsElapsed + 1
    }));
  }

  componentD Mount() {
    this.int val = setInterval(() => this.tick(), 1000);
  }

  componentW lUnmount() {
    clearInt val(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}
```

It turns out that when **this.setState** is invoked, React will update the state and *automatically* call the **render** method to re-render the component.

So, how do we call **tick** every second?

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A Timer Application



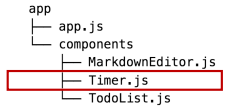
But, how will this update the view rendered to the user?

```
import React from 're

export default class
  constructor(props)
    super(props);
    this.state = {secondsE
  }

  tick() {
    this.setState((prevState) => ({
      secon lapsed: prevState.secondsElapsed + 1
    }));
  }

  componentD Mount() {
    this.int val = setInterval(() => this.tick(), 1000);
  }

  componentW lUnmount() {
    clearInt val(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}
```

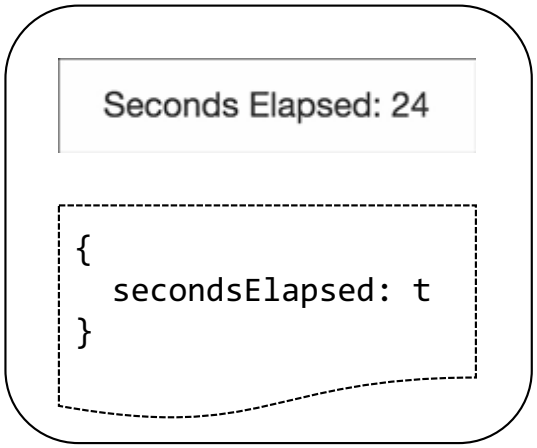
It turns out that when **this.setState** is invoked, React will update the state and *automatically* call the **render** method to re-render the component.

So, how do we call **tick** every second?

It turns out that there are other *special* functions associated with React components.



A Timer Application




```
import React from 'react';

export default class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {secondsElapsed: 0};
  }

  tick() {
    this.setState((prevState) => ({
      secondsElapsed: prevState.secondsElapsed + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}
```

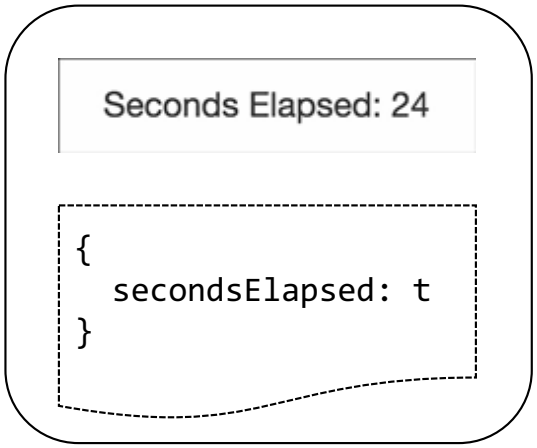
It turns out that when **this.setState** is invoked, React will update the state and *automatically* call the **render** method to re-render the component.

So, how do we call **tick** every second?

It turns out that there are other *special* functions associated with React components.

componentDidMount is one of these special functions.

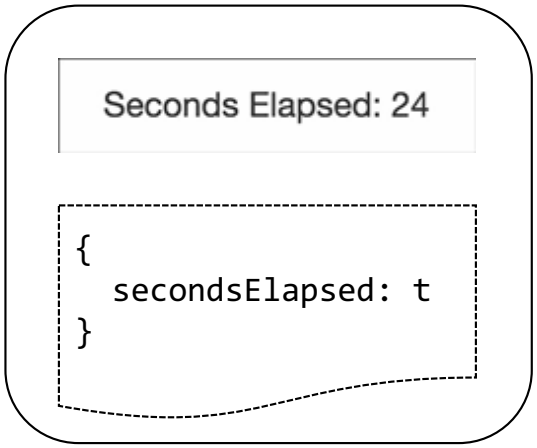
A Timer Application




```
import React from 'react';

export default class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {secondsElapsed: 0};
  }

  tick() {
    this.setState((prevState) => ({
      secondsElapsed: prevState.secondsElapsed + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}
```

componentDidMount is called automatically by React right after the component is mounted and rendered in the browser.

A Timer Application




```
import React from 'react';

export default class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {secondsElapsed: 0};
  }

  tick() {
    this.setState((prevState) => ({
      secondsElapsed: prevState.secondsElapsed + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}
```

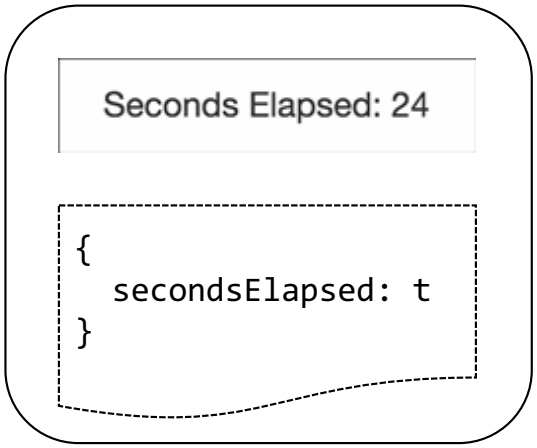
componentDidMount is called automatically by React right after the component is mounted and rendered in the browser.

In the Timer component we want to begin the timer right after it has been rendered the first time.

So, we use **setInterval** to call the **tick** method after each second.

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A Timer Application



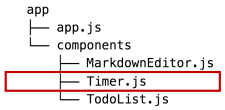

```
import React from 'react';

export default class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {secondsElapsed: 0};
  }

  tick() {
    this.setState((prevState) => ({
      secondsElapsed: prevState.secondsElapsed + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}
```

componentDidMount is called automatically by React right after the component is mounted and rendered in the browser.

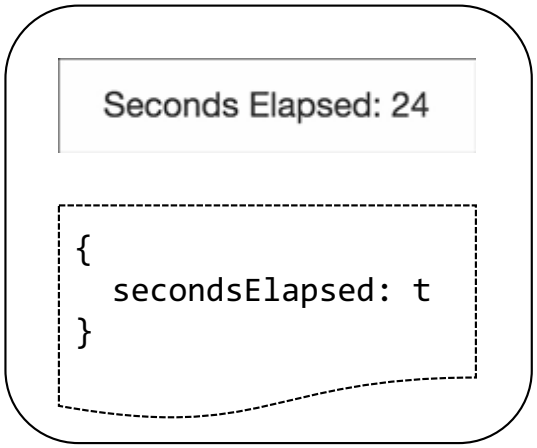
In the Timer component we want to begin the timer right after it has been rendered the first time.

So, we use **setInterval** to call the **tick** method after each second.

The return value of **setInterval** is an object that identifies that particular interval. So…



A Timer Application




```
import React from 'react';

export default class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {secondsElapsed: 0};
  }

  tick() {
    this.setState((prevState) => ({
      secondsElapsed: prevState.secondsElapsed + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}
```

… The call to **tick** every second can be removed, or “cleared”.

When do we want this to happen?

In another special React method called **componentWillUnmount**.

Called right after the component is unmounted (removed from the DOM) and “destroyed”.

A ToDo List Application

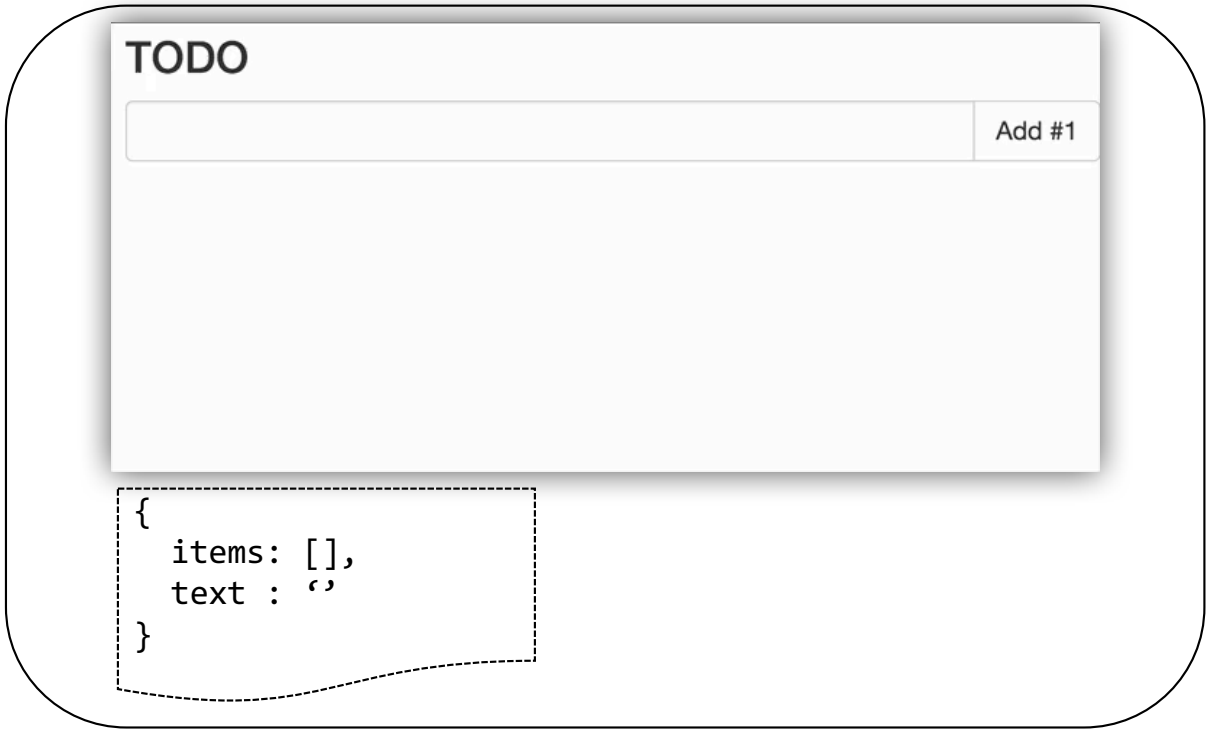
This is a simple ToDo application.

The ToDo application allows the user to type in a task into the ToDo list input text box.

The user can either hit *enter* or press the button “Add #n” to add the task to the list .

What is the **state** of this application?

A ToDo List Application



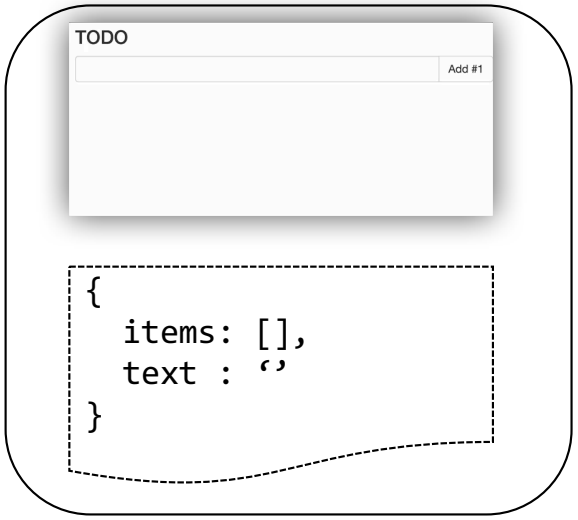
This is a simple ToDo application.

The ToDo application allows the user to type in a task into the ToDo list input text box.

The user can either hit *enter* or press the button “Add #n” to add the task to the list .

What is the **state** of this application?

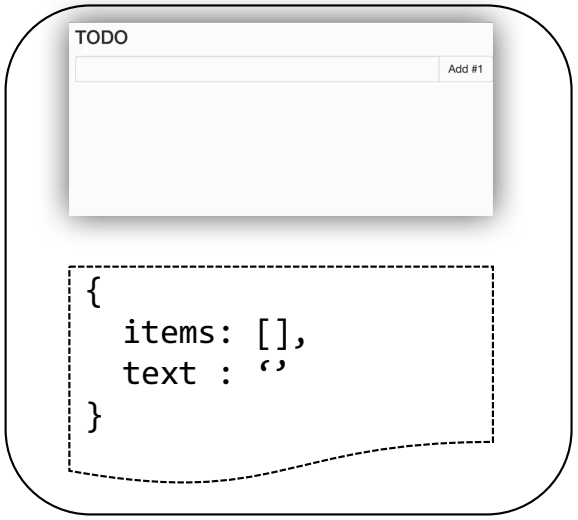
A ToDo List Application




```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} className="input-group">
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}


class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

A ToDo List Application



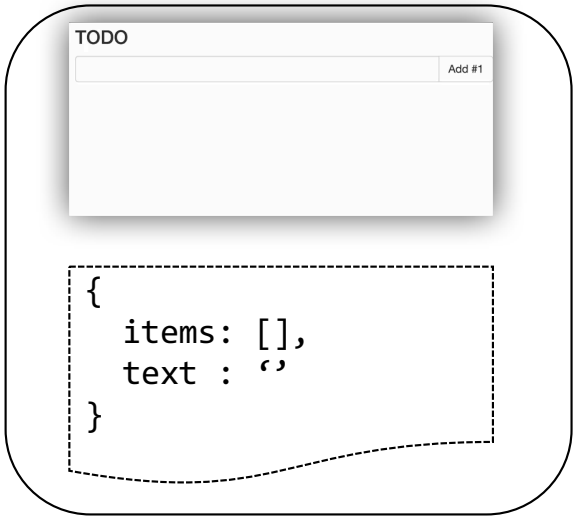

```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} className="input-group">
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

As usual, we extend **React.Component** to create a new component called **TodoApp**.

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application




```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items:
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} className="input-group">
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

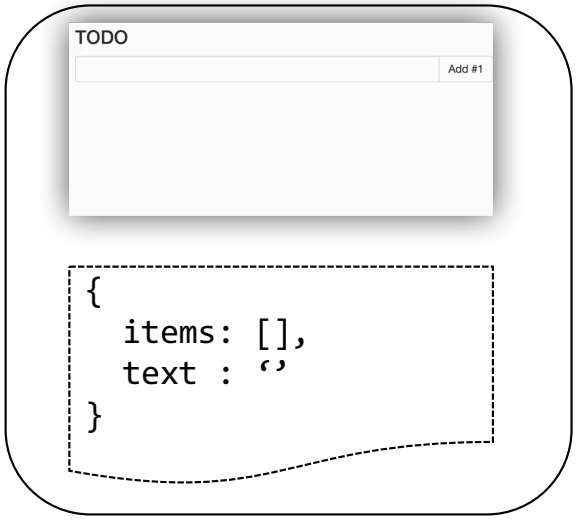
As usual, we extend **React.Component** to create a new component called **TodoApp**.

We call the super class constructor to initialize the **props** correctly.

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application




```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} className="input-group">
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

As usual, we extend **React.Component** to create a new component called **TodoApp**.

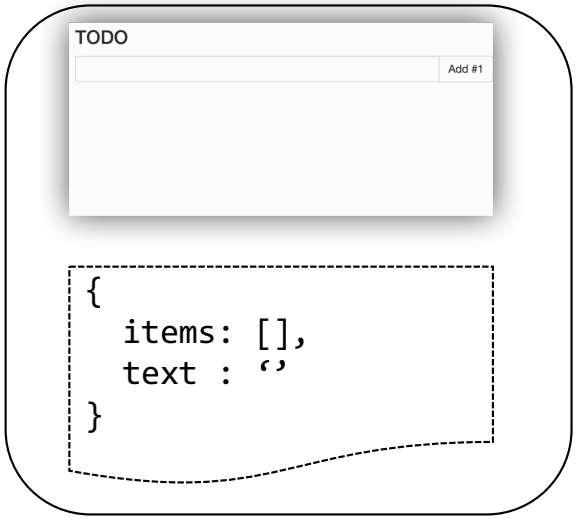
We call the super class constructor to initialize the **props** correctly.

We initialize the state of this component with the todo list items and the text contained in the input box.

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application



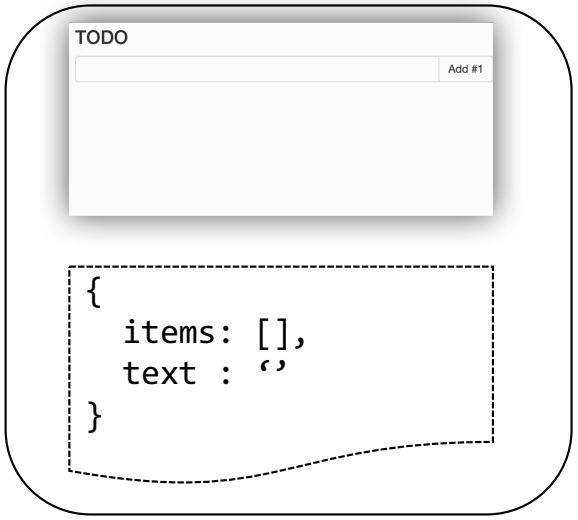

```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} className="input-group">
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

The **render** method consists of three important parts:

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application




```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} className="input-group">
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

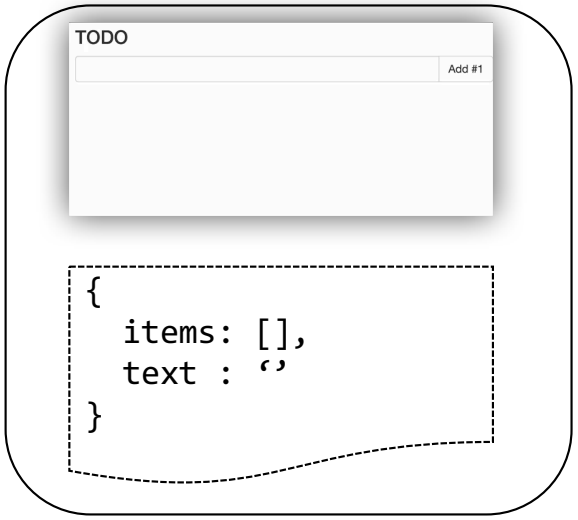
The **render** method consists of two important parts:

- The **TodoList** component
  A separate component for managing and rendering the list of tasks

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application




```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} className="input-group">
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

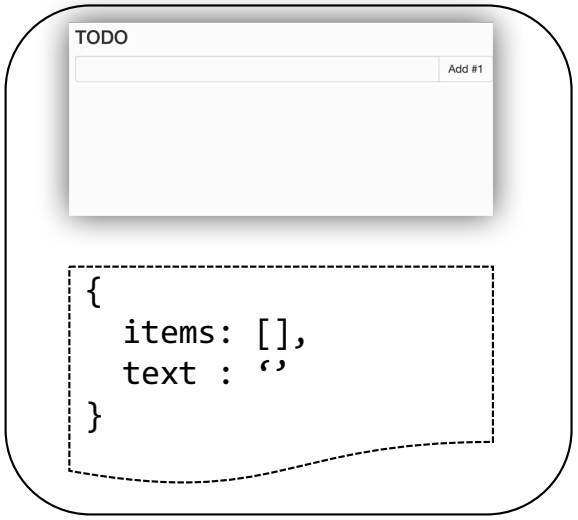
The **render** method consists of two important parts:

- The **TodoList** component
  A separate component for managing and rendering the list of tasks
- The form containing the input box the user types the task in to and the button we will use to add a task to the todo list.

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application



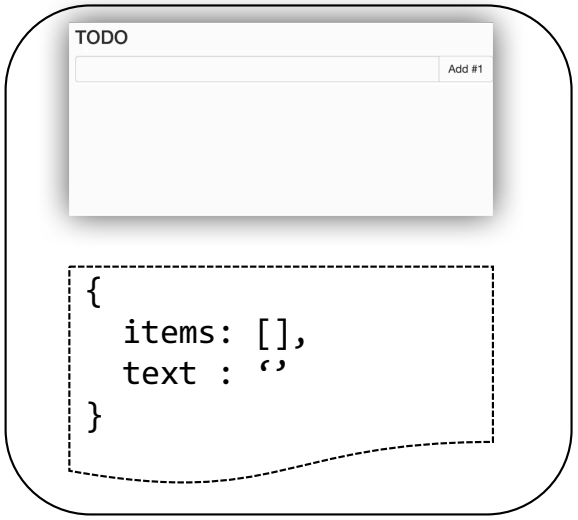

```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} className="input-group">
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

React provides support for each of the possible events that might occur on a DOM element as an attribute that can be assigned a function.

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application




```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} className="input-group">
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

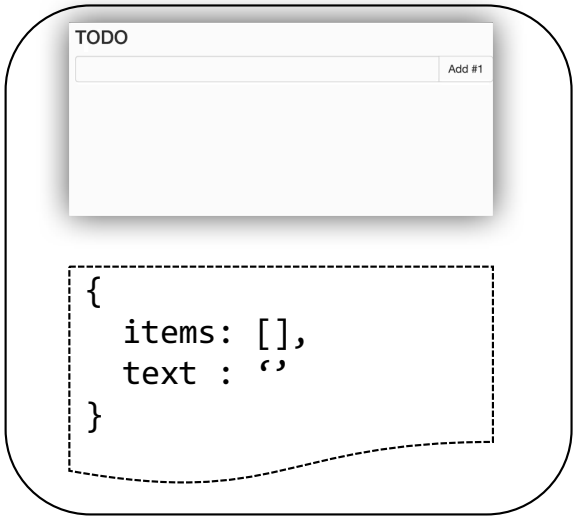
React provides support for each of the possible events that might occur on a DOM element as an attribute that can be assigned a function.

If the form receives a **submit** event (a button was pushed) we want to handle the submission.

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application




```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)}>
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

React provides support for each of the possible events that might occur on a DOM element as an attribute that can be assigned a function.

If the form receives a **submit** event (a button was pushed) we want to handle the submission.

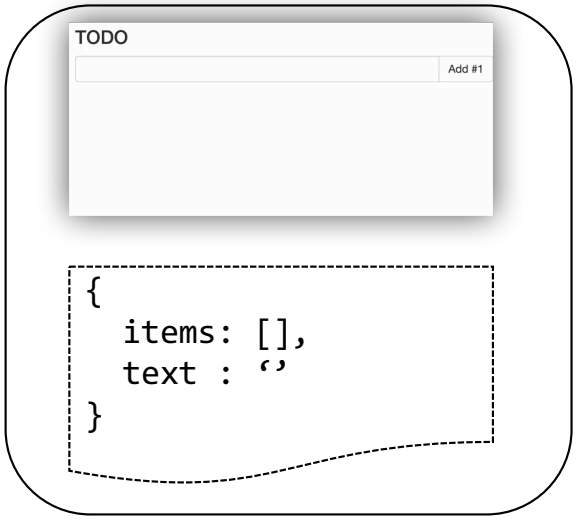
We indicate this by creating a new anonymous function that receives an event object *e* and calls the **handleSubmit** method defined by the TodoApp component.

Why do we need an anonymous => function?

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application




```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} classN
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

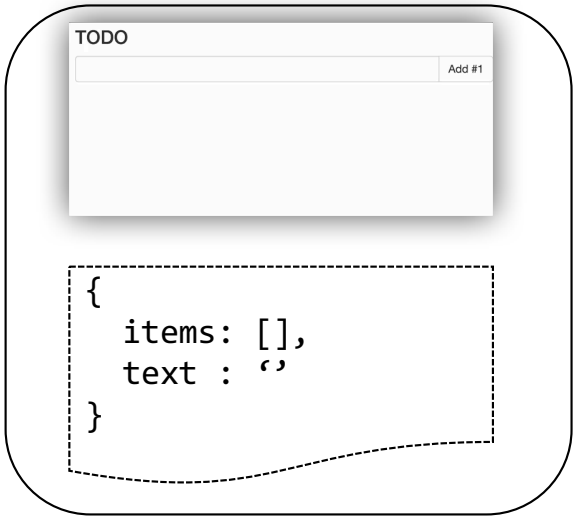
Likewise, If there is a change to the text input we want to update the internal state of the component to reflect what is currently displayed in the input box.

We do this by capturing a **change** event on the input text box. We once again create an anonymous => function that calls the **handleChange** method of the TodoApp component.

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application




```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} classN
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

NOTE: You are not invoking these functions *here*.

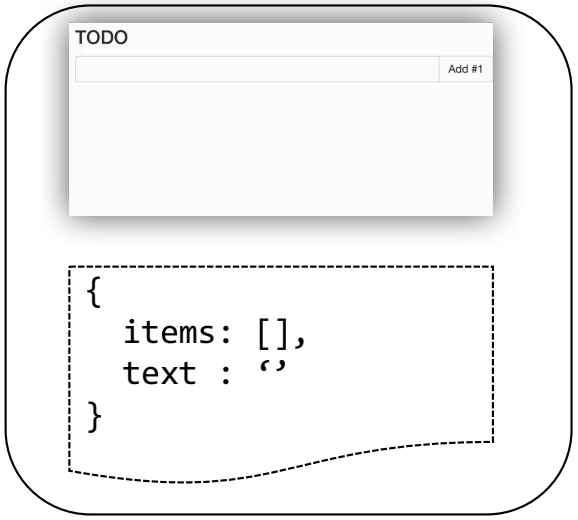
This code creates two new functions that are to be called when each of the corresponding events occurs.

This is the standard way to associate behavior and functionality of an application to parts of a UI.

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application




```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} classN
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

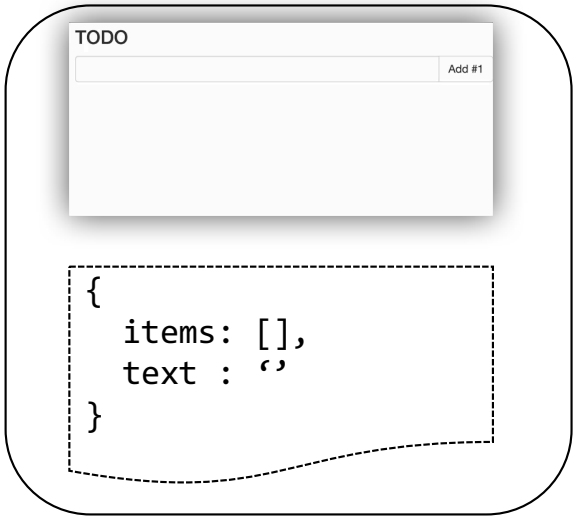
We also update the view to reflect the current **state** of the TodoApp component.

In this case, we set the value of the input box to be the value of the state’s *text* property.

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application




```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} className="input-group">
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

We also update the view to reflect the current **state** of the TodoApp component.

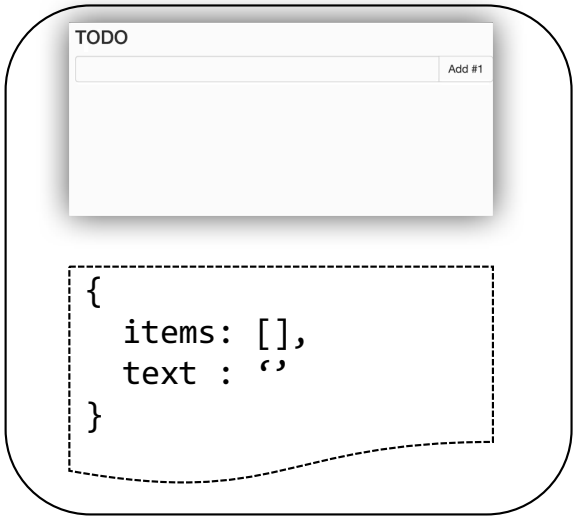
In this case, we set the value of the input box to be the value of the state’s *text* property.

We also set the button’s text value to indicate what the next item number it is you are adding.

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application




```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} classN
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

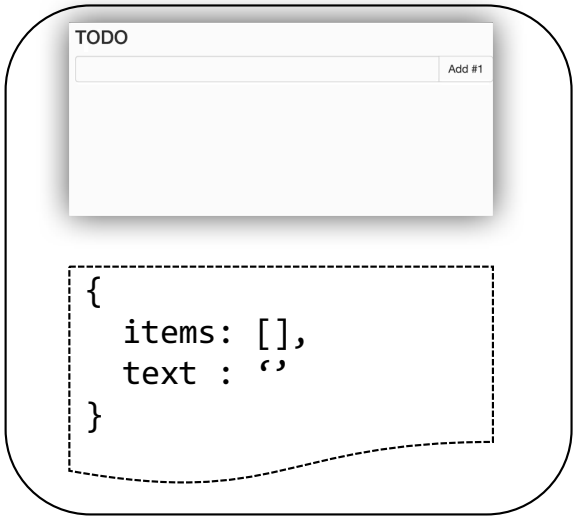
So…

What do these functions do?

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application




```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} className="input-group">
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

So…

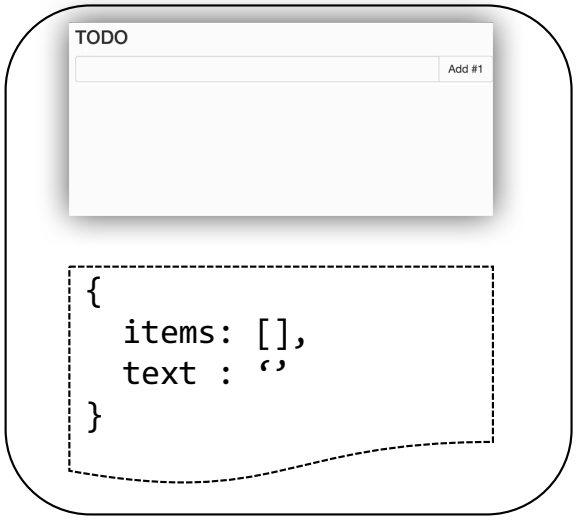
What do these functions do?

handleChange sets the state to be the current value of the DOM element (e.target) that received the event.

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application




```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} className="input-group">
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

So…

What do these functions do?

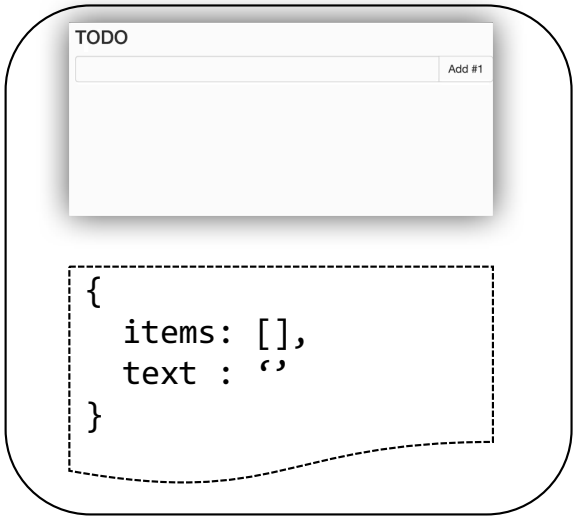
handleChange sets the state to be the current value of the DOM element (e.target) that received the event.

Because we set the state of the component it causes the **render** method to be invoked to re-render the component.

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application




```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} className="input-group">
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

So…

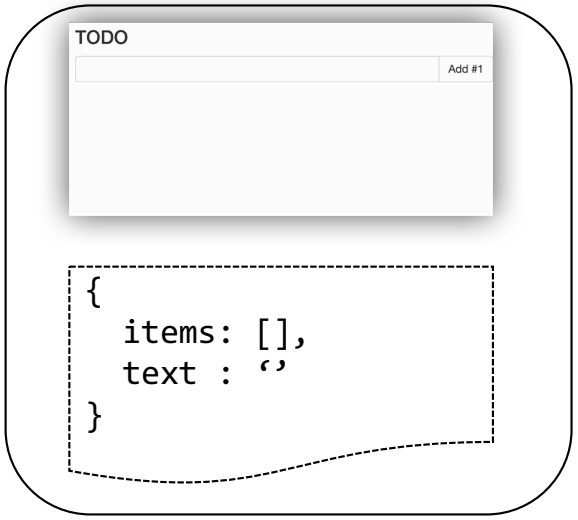
What do these functions do?

handleSubmit will deal with adding a new item to the todo list application.

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application




```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} className="input-group">
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

So…

What do these functions do?

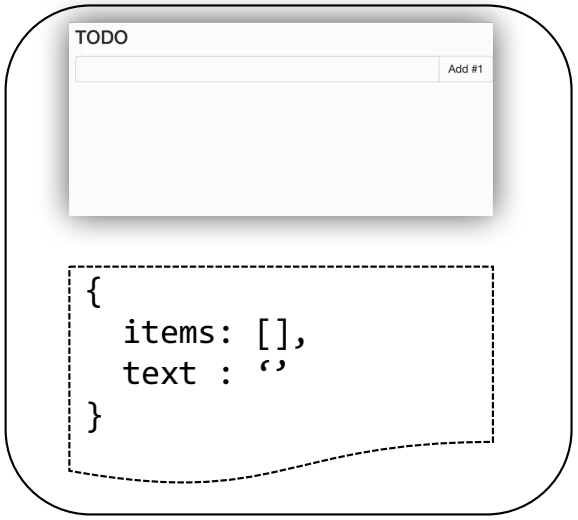
handleSubmit will deal with adding a new item to the todo list application.

First, we call the **preventDefault**() method on the event object. **Why?**

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application




```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} className="input-group">
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

So…

What do these functions do?

handleSubmit will deal with adding a new item to the todo list application.

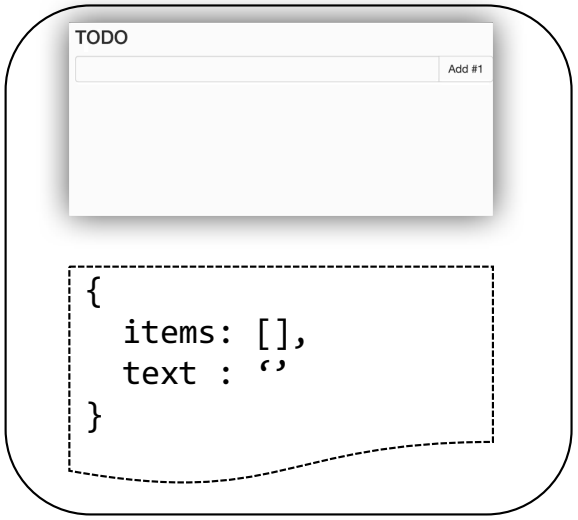
First, we call the **preventDefault**() method on the event object. **Why?**

We then create the new todo list item as a JavaScript object literal.

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application




```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} className="input-group">
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    )
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

So…

What do these functions do?

handleSubmit will deal with adding a new item to the todo list application.

First, we call the **preventDefault**() method on the event object. **Why?**

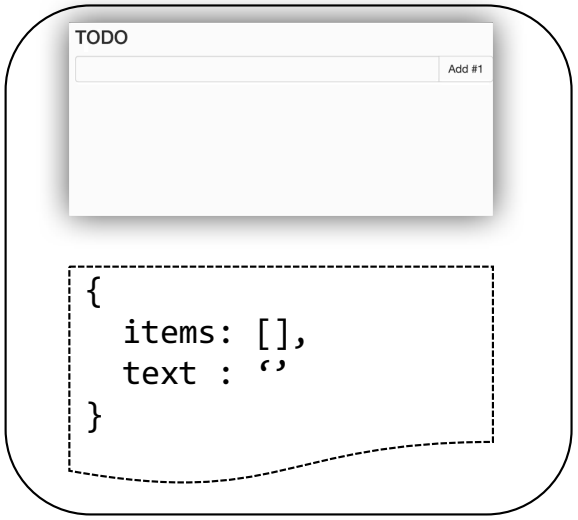
We then create the new todo list item as a JavaScript object literal.

Lastly, we set the state which indirectly causes **render** to be invoked to update the view.

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application




```jsx
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={(e) => this.handleSubmit(e)} className="input-group">
          <input onChange={(e) => this.handleChange(e)}
            className="form-control" value={this.state.text} />
          <span className="input-group-btn">
            <button className="btn btn-default">
              {'Add #' + (this.state.items.length + 1)}
            </button>
          </span>
        </form>
      </div>
    );
  }

  handleChange(e) {
    this.setState({text: e.target.value});
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

The **render** method consists of two important parts:

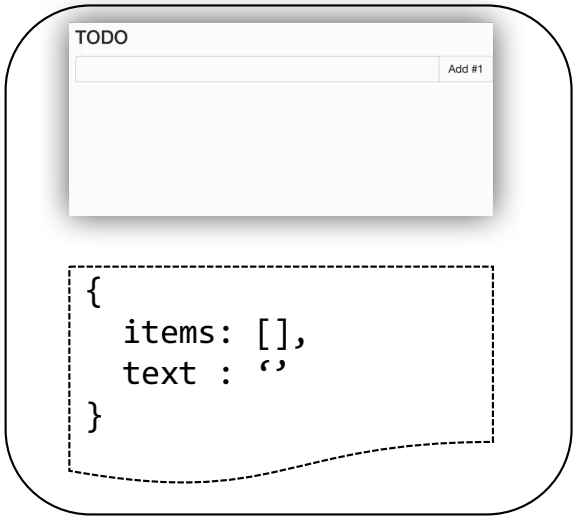
- The **TodoList** component
  A separate component for managing and rendering the list of tasks

  So, what about this line?

```jsx
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A ToDo List Application




```
import React from 'react';

export default class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: [], text: ''};
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
```

```
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

```
    );
  }

  handle
    this
  }

  handleSubmit(e) {
    e.preventDefault();
    var newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState((prevState) => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}
```

The **render** method consists of two important parts:

- The **TodoList** component
  A separate component for managing and rendering the list of tasks

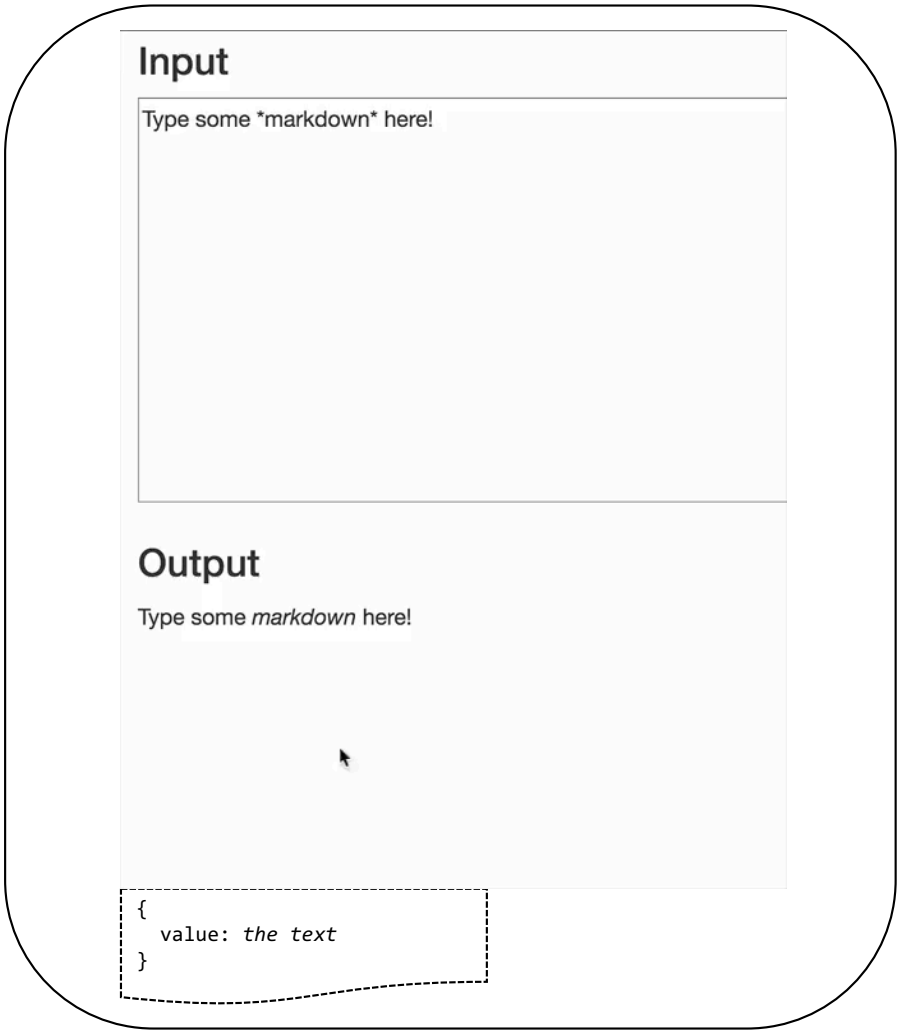
  So, what about this line?

- The TodoList component receives the list of items and iterates over them to produce the list of items currently in the todo list.

  Why do we not use React’s Children Map function?

```
app
├── app.js
└── components
    ├── MarkdownEditor.js
    ├── Timer.js
    └── TodoList.js
```

A Markdown Editor Application




```
import React from 'react';
import Markdown from 'react-remarkable';

export default class MarkdownEditor extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 'Type some *markdown* here!'};
  }

  handleChange() {
    this.setState({value: this.refs.textarea.value});
  }

  render() {
    return (
      <div className="MarkdownEditor">
        <h3>Input</h3>
        <textarea
          onChange={() => this.handleChange()}
          cols="100"
          ref="textarea"
          defaultValue={this.state.value} />
        <h3>Output</h3>
        <Markdown source={this.state.value}/>
      </div>
    );
  }
}
```