State

Constructor(props)

The constructor() method is called before a React Component is mounted and is used to set up the initial state of the component. It is important to call super(props) at the beginning of the constructor() method or else the *this.props* attribute may not work correctly. The first argument to the constructor() method represents the properties that are passed into the component.

```
class Counter extends React.Component{
    constructor(props){
        super(props)
    }
    render(){
        return <div>Hello World!</div>
    }
}
```

Adding an initial state to Class Components

The initial state of a Class Component can be declared within the **constructor()** method. The state of the component must be declared as an object with attributes.

```
class Counter extends React.Component{
    constructor(props){
        super(props)
        this.state = {foo:123,bar:456}
    }
    render(){
        return <div>foo:{this.state.foo} bar:{this.state.bar}</div>
    }
}
```

Updating state

The set*State(updater,[callback])* method is used to update the state of the component. It takes in an updater object and updates the component state by shallowly merging the updater object's attributes with the previous component state. The method updates the state asynchronously, so a there is an option callback that will be called once the state has finished updating completely. In order to use the set*State()* method, it must be referenced by calling *this.setState()*.

The setState method will trigger the updating phase of the component lifecycle to start. This will cause the component to rerender unless the shouldComponentUpdate() function returns false.

Example:

```
this.setState({message:"new message"})
```

For example:

```
class Counter extends React.Component{
    constructor(props){
        super(props)
        //initial state set up
        this.state = {message:"initial message"}
    }
    componentDidMount(){
        //updating state
        this.setState({message:"new message"})
    }
    render(){
        return <div>Message:{this.state.message}</div>
    }
}
```

Updating state based on previous state

The set*State()* method does not immediately update the state of the component, it just puts the update in a queue to be processed later. React may batch multiple update requests together to make rendering more efficient. Due to this, special precautions must be made when updating the state based on the component's previous state.

For example, the following code will only increment the state value attribute by 1 even though it was called 4 times:

```
class Counter extends React.Component{
    constructor(props){
        super(props)
        //initial state set up
        this.state = {value:0}
    }
    componentDidMount(){
        //updating state
        this.setState({value:this.state.value+1})
        this.setState({value:this.state.value+1})
        this.setState({value:this.state.value+1})
        this.setState({value:this.state.value+1})
    }
    render(){
        return <div>Message:{this.state.message}</div>
    }
}
```

The set*State(updater,[callback])* method can take in an updater function as its first argument to update the state based on the previous state and properties. The return value of the updater function will be shallowly merged with the previous component state. The method updates the state asynchronously, so a there is an option callback that will be called once the state has finished updating completely.

Example:

```
this.setState((prevState, props) => {
    return {attribute:"value"}
})
```

Here is an example of how to update the state based on previous state:

```
class Counter extends React.Component{
    constructor(props){
        super(props)
        //initial state set up
        this.state = {message:"initial message"}
    }
    componentDidMount()
        //updating state
        this.setState((prevState, props) => {
            return {message: prevState.message + '!'}
        })
    }
    render(){
        return <div>Message:{this.state.message}</div>
    }
}
```

Using future state values

Since state updates asynchronously, you can not just expect the state values to update immediately after a setState() method call.

For example, the console log may not output the updated state:

```
//this.state.count is originally 0
this.setState({count:42})
console.log(this.state.count)
//outputs 0 still
```

In order to use a state after it has been updated, do all logic in the callback argument:

```
//this.state.count is originally 0
this.setState({count:42}, () = {
    console.log(this.state.count)
    //outputs 42
})
```

State is not mutable

State is read only so you should not try to manually change the values of the state attributes. If the state needs to be updated, the set*State()* method is the only way to change the state.

For example, don't do this:

```
//incorrect, state should not be mutated directly
this.state.message = "new message"
```