React Without ES6

Normally you would define a React component as a plain JavaScript class:

```
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

If you don't use ES6 yet, you may use the `create-react-class` module instead:

```
var createReactClass = require('create-react-class');
var Greeting = createReactClass({
  render: function() {
    return <h1>Hello, {this.props.name}</h1>;
  }
});
```

The API of ES6 classes is similar to `createReactClass()` with a few exceptions.

Declaring Default Props

With functions and ES6 classes `defaultProps` is defined as a property on the component itself:

```
class Greeting extends React.Component {
  // ...
}

Greeting.defaultProps = {
  name: 'Mary'
};
```

With `createReactClass()`, you need to define `getDefaultProps()` as a function on the passed object:

```
var Greeting = createReactClass({
  getDefaultProps: function() {
    return {
      name: 'Mary'
    };
  },
```

```
  // ...

});
```

Setting the Initial State

In ES6 classes, you can define the initial state by assigning `this.state` in the constructor:

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
  }
  // ...
}
```

With `createReactClass()`, you have to provide a separate `getInitialState` method that returns the initial state:

```
var Counter = createReactClass({
  getInitialState: function() {
    return {count: this.props.initialCount};
  },
  // ...
});
```

Autobinding

In React components declared as ES6 classes, methods follow the same semantics as regular ES6 classes. This means that they don't automatically bind `this` to the instance. You'll have to explicitly use `.bind(this)` in the constructor:

```
class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: 'Hello!'};
    // This line is important!
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    alert(this.state.message);
  }

  render() {
```

```
    // Because `this.handleClick` is bound, we can use it as an event handler.
    return (
      <button onClick={this.handleClick}>
        Say hello
      </button>
    );
  }
}
```

With `createReactClass()`, this is not necessary because it binds all methods:

```
var SayHello = createReactClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },

  handleClick: function() {
    alert(this.state.message);
  },

  render: function() {
    return (
      <button onClick={this.handleClick}>
        Say hello
      </button>
    );
  }
});
```

This means writing ES6 classes comes with a little more boilerplate code for event handlers, but the upside is slightly better performance in large applications.
If the boilerplate code is too unattractive to you, you may enable the **experimental** Class Properties syntax proposal with Babel:

```
class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: 'Hello!'};
  }
  // WARNING: this syntax is experimental!
  // Using an arrow here binds the method:
  handleClick = () => {
    alert(this.state.message);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
```

```
        Say hello
      </button>
    );
  }
}
```

Please note that the syntax above is **experimental** and the syntax may change, or the proposal might not make it into the language.

If you'd rather play it safe, you have a few options:

- Bind methods in the constructor.
- Use arrow functions, e.g. `onClick={(e) => this.handleClick(e)}`.
- Keep using `createReactClass`.

Mixins

Note:
ES6 launched without any mixin support. Therefore, there is no support for mixins when you use React with ES6 classes.
We also found numerous issues in codebases using mixins, and don't recommend using them in the new code.

This section exists only for the reference.

Sometimes very different components may share some common functionality. These are sometimes called cross-cutting concerns. `createReactClass` lets you use a legacy `mixins`system for that.
One common use case is a component wanting to update itself on a time interval. It's easy to use `setInterval()`, but it's important to cancel your interval when you don't need it anymore to save memory. React provides lifecycle methods that let you know when a component is about to be created or destroyed. Let's create a simple mixin that uses these methods to provide an easy `setInterval()` function that will automatically get cleaned up when your component is destroyed.

```
var SetIntervalMixin = {
  componentWillMount: function() {
    this.intervals = [];
  },
  setInterval: function() {
    this.intervals.push(setInterval.apply(null, arguments));
  },
  componentWillUnmount: function() {
    this.intervals.forEach(clearInterval);
  }
};
```

```
var createReactClass = require('create-react-class');

var TickTock = createReactClass({
  mixins: [SetIntervalMixin], // Use the mixin
  getInitialState: function() {
    return {seconds: 0};
  },
  componentDidMount: function() {
    this.setInterval(this.tick, 1000); // Call a method on the mixin
  },
  tick: function() {
    this.setState({seconds: this.state.seconds + 1});
  },
  render: function() {
    return (
      <p>
        React has been running for {this.state.seconds} seconds.
      </p>
    );
  }
});

ReactDOM.render(
  <TickTock />,
  document.getElementById('example')
);
```

If a component is using multiple mixins and several mixins define the same lifecycle method (i.e. several mixins want to do some cleanup when the component is destroyed), all of the lifecycle methods are guaranteed to be called. Methods defined on mixins run in the order mixins were listed, followed by a method call on the component.