# React on ES6+

**Classes**

By far the most outwardly visible change to how we write React components using ES6+ comes about when we choose to use the [**class definition syntax**](https://babeljs.io/docs/learn-es2015/#classes). Instead of using the React.createClass method to define a component, we can define a bonafide ES6 class that extendsReact.Component:

class Photo extends React.Component {

render() {

return <img alt={this.props.caption} src={this.props.src} />;

}

}

Right away, you’ll notice a subtle difference – a more terse syntax is available to you when defining classes:

// The ES5 way

var Photo = React.createClass({

handleDoubleTap: function(e) { … },

render: function() { … },

});

// The ES6+ way

class Photo extends React.Component {

handleDoubleTap(e) { … }

render() { … }

}

Notably, we’ve dropped two parentheses and a trailing semicolon, and for each method declared we omit a colon, a function keyword, and a comma.

All of the lifecycle methods but one can be defined as you would expect when using the new class syntax. The class’ constructor now assumes the role previously filled by componentWillMount:

// The ES5 way

var EmbedModal = React.createClass({

componentWillMount: function() { … },

});

// The ES6+ way

class EmbedModal extends React.Component {

constructor(props) {

super(props);

// Operations usually carried out in componentWillMount go here

}

}

**Property initializers**

In the ES6+ class world, prop types and defaults live as static properties on the class itself. These, as well as the component’s initial state, can be defined using ES7 [**property initializers**](https://gist.github.com/jeffmo/054df782c05639da2adb):

// The ES5 way

var Video = React.createClass({

getDefaultProps: function() {

return {

autoPlay: false,

maxLoops: 10,

};

},

getInitialState: function() {

return {

loopsRemaining: this.props.maxLoops,

};

},

propTypes: {

autoPlay: React.PropTypes.bool.isRequired,

maxLoops: React.PropTypes.number.isRequired,

posterFrameSrc: React.PropTypes.string.isRequired,

videoSrc: React.PropTypes.string.isRequired,

},

});

// The ES6+ way

class Video extends React.Component {

static defaultProps = {

autoPlay: false,

maxLoops: 10,

}

static propTypes = {

autoPlay: React.PropTypes.bool.isRequired,

maxLoops: React.PropTypes.number.isRequired,

posterFrameSrc: React.PropTypes.string.isRequired,

videoSrc: React.PropTypes.string.isRequired,

}

state = {

loopsRemaining: this.props.maxLoops,

}

}

ES7 property initializers operate inside the class’ constructor, where thisrefers to the instance of the class under construction, so the initial state can still be made to depend on this.props. Notably, we no longer have to define prop defaults and the initial state object in terms of a getter function.

**Arrow functions**

The React.createClass method used to perform some extra binding work on your component’s instance methods to make sure that, inside them, thethis keyword would refer to the instance of the component in question.

// Autobinding, brought to you by React.createClass

var PostInfo = React.createClass({

handleOptionsButtonClick: function(e) {

// Here, 'this' refers to the component instance.

this.setState({showOptionsModal: true});

},

});

Since we don’t involve the React.createClass method when we define components using the ES6+ class syntax, it would seem that we need to manually bind instance methods wherever we want this behavior:

// Manually bind, wherever you need to

class PostInfo extends React.Component {

constructor(props) {

super(props);

// Manually bind this method to the component instance...

this.handleOptionsButtonClick = this.handleOptionsButtonClick.bind(this);

}

handleOptionsButtonClick(e) {

// ...to ensure that 'this' refers to the component instance here.

this.setState({showOptionsModal: true});

}

}

Luckily, by combining two ES6+ features – [**arrow functions**](https://babeljs.io/docs/learn-es2015/#arrows) and property initializers – opt-in binding to the component instance becomes a breeze:

class PostInfo extends React.Component {

handleOptionsButtonClick = (e) => {

this.setState({showOptionsModal: true});

}

}

The body of ES6 arrow functions share the same lexical this as the code that surrounds them, which gets us the desired result because of the way that ES7 property initializers are scoped. [Peek under the hood](https://babeljs.io/repl/#?experimental=true&evaluate=true&loose=false&spec=false&code=class%20PostInfo%20extends%20React.Component%20%7B%0A%09handleOptionsButtonClick%20%3D%20(e)%20%3D%3E%20%7B%0A%20%20%20%20this.setState(%7BshowOptionsModal%3A%20true%7D)%3B%0A%20%20%7D%0A%7D) to see why this works.

**Dynamic property names & template strings**

One of the [**enhancements to object literals**](https://babeljs.io/docs/learn-es2015/#enhanced-object-literals) includes the ability to assign to a derived property name. We might have originally done something like this to set a piece of state:

var Form = React.createClass({

onChange: function(inputName, e) {

var stateToSet = {};

stateToSet[inputName + 'Value'] = e.target.value;

this.setState(stateToSet);

},

});

Now, we have the ability to construct objects whose property names are determined by a JavaScript expression at runtime. Here, we use a[**template string**](https://babeljs.io/docs/learn-es2015/#template-strings) to determine which property to set on state:

class Form extends React.Component {

onChange(inputName, e) {

this.setState({

[`${inputName}Value`]: e.target.value,

});

}

}

**Destructuring & spread attributes**

Often when composing components, we might want to pass down *most* of a parent component’s props to a child component, but not all of them. In combining ES6+ **[destructuring](https://babeljs.io/docs/learn-es2015/" \l "destructuring)** with JSX [**spread attributes**](https://facebook.github.io/react/docs/jsx-spread.html), this becomes possible without ceremony:

class AutoloadingPostsGrid extends React.Component {

render() {

var {

className,

...others, // contains all properties of this.props except for className

} = this.props;

return (

<div className={className}>

<PostsGrid {...others} />

<button onClick={this.handleLoadMoreClick}>Load more</button>

</div>

);

}

}

We can combine JSX spread attributes with regular attributes too, taking advantage of a simple precedence rule to implement overrides and defaults. This element will acquire the className “override” even if there exists a className property in this.props:

<div {...this.props} className="override">

…

</div>

This element will regularly have the className “base” unless there exists a className property in this.props to override it:

<div className="base" {...this.props}>

…

</div>