Features from ES6, a recent version of JavaScript. In this tutorial, we’re using arrow functions, classes, var, let, and const statements.

**Functions =>**

var materials = [

'Hydrogen',

'Helium',

'Lithium',

'Beryllium'

];

console.log(materials.map(material => material.length));

// expected output: Array [8, 6, 7, 9]

materials.map(function(material) {

return material.length;

}); // [8, 6, 7, 9]

materials.map((material) => {

return material.length;

}); // [8, 6, 7, 9]

materials.map(({length}) => length); // [8, 6, 7, 9]

var func = x => x \* x;

// concise body syntax, implied "return"

var func = (x, y) => { return x + y; };

// with block body, explicit "return" needed

**Defining classes**

Classes are in fact "special functions", and just as you can define function expressions and function declarations, the class syntax has two components: class expressions and class declarations.

### Class declarations

One way to define a class is using a **class declaration**. To declare a class, you use the class keyword with the name of the class ("Rectangle" here).

class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

}

#### Hoisting

An important difference between **function declarations** and **class declarations** is that function declarations are hoisted and class declarations are not. You first need to declare your class and then access it, otherwise code like the following will throw a ReferenceError:

var p = new Rectangle(); // ReferenceError

class Rectangle {}

### Class expressions

A **class expression** is another way to define a class. Class expressions can be named or unnamed. The name given to a named class expression is local to the class's body. (it can be retrieved through the class's (not an instance's) .name property, though)

// unnamed

var Rectangle = class {

constructor(height, width) {

this.height = height;

this.width = width;

}

};

// named

var Rectangle = class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

};

**Note:** Class **expressions** also suffer from the same hoisting issues mentioned for Class **declarations**.

## Class body and method definitions

The body of a class is the part that is in curly brackets {}. This is where you define class members, such as methods or constructor.

### Strict mode

The bodies of class declarations and class expressions are executed in strict mode i.e. constructor, static and prototype methods, getter and setter functions are executed in strict mode.

### Constructor

The constructor method is a special method for creating and initializing an object created with a class. There can only be one special method with the name "constructor" in a class. A SyntaxError will be thrown if the class contains more than one occurrence of a constructor method.

A constructor can use the super keyword to call the constructor of the super class.

### Prototype methods

See also method definitions.

class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

// Getter

get area() {

return this.calcArea();

}

// Method

calcArea() {

return this.height \* this.width;

}

}

const square = new Rectangle(10, 10);

console.log(square.area); // 100

### Static methods

The static keyword defines a static method for a class. Static methods are called without instantiating their class and **cannot**be called through a class instance. Static methods are often used to create utility functions for an application.

class Point {

constructor(x, y) {

this.x = x;

this.y = y;

}

static distance(a, b) {

const dx = a.x - b.x;

const dy = a.y - b.y;

return Math.hypot(dx, dy);

}

}

const p1 = new Point(5, 5);

const p2 = new Point(10, 10);

console.log(Point.distance(p1, p2)); // 7.0710678118654755

### Boxing with prototype and static methods

When a static or prototype method is called without an object valued "this", then the "this" value will be **undefined** inside the called function. Autoboxing will not happen. The behavior will be the same even if we write the code in non-strict mode because all the functions, methods, constructor, getters or setters are executed in strict mode. So if we do not specify this value then the this value will be **undefined.**

class Animal {

speak() {

return this;

}

static eat() {

return this;

}

}

let obj = new Animal();

obj.speak(); // Animal {}

let speak = obj.speak;

speak(); // undefined

Animal.eat() // class Animal

let eat = Animal.eat;

eat(); // undefined

If we write the above code using traditional function based classes, then autoboxing will happen based on the "this" value for which the function was called.

function Animal() { }

Animal.prototype.speak = function() {

return this;

}

Animal.eat = function() {

return this;

}

let obj = new Animal();

let speak = obj.speak;

speak(); // global object

let eat = Animal.eat;

eat(); // global object

## Sub classing with extends

The extends keyword is used in class declarations or class expressions to create a class as a child of another class.

class Animal {

constructor(name) {

this.name = name;

}

speak() {

console.log(this.name + ' makes a noise.');

}

}

class Dog extends Animal {

speak() {

console.log(this.name + ' barks.');

}

}

var d = new Dog('Mitzie');

d.speak(); // Mitzie barks.

If there is a constructor present in sub-class, it needs to first call super() before using "this".

One may also extend traditional function-based "classes":

function Animal (name) {

this.name = name;

}

Animal.prototype.speak = function () {

console.log(this.name + ' makes a noise.');

}

class Dog extends Animal {

speak() {

console.log(this.name + ' barks.');

}

}

var d = new Dog('Mitzie');

d.speak(); // Mitzie barks.

Note that classes cannot extend regular (non-constructible) objects. If you want to inherit from a regular object, you can instead use Object.setPrototypeOf():

var Animal = {

speak() {

console.log(this.name + ' makes a noise.');

}

};

class Dog {

constructor(name) {

this.name = name;

}

}

// If you do not do this you will get a TypeError when you invoke speak

Object.setPrototypeOf(Dog.prototype, Animal);

var d = new Dog('Mitzie');

d.speak(); // Mitzie makes a noise.

## Species

You might want to return Array objects in your derived array class MyArray. The species pattern lets you override default constructors.

For example, when using methods such as map() that returns the default constructor, you want these methods to return a parent Array object, instead of the MyArray object. The Symbol.species symbol lets you do this:

class MyArray extends Array {

// Overwrite species to the parent Array constructor

static get [Symbol.species]() { return Array; }

}

var a = new MyArray(1,2,3);

var mapped = a.map(x => x \* x);

console.log(mapped instanceof MyArray); // false

console.log(mapped instanceof Array); // true

## Super class calls with super

The super keyword is used to call corresponding methods of super class.

class Cat {

constructor(name) {

this.name = name;

}

speak() {

console.log(this.name + ' makes a noise.');

}

}

class Lion extends Cat {

speak() {

super.speak();

console.log(this.name + ' roars.');

}

}

var l = new Lion('Fuzzy');

l.speak();

// Fuzzy makes a noise.

// Fuzzy roars.

Let

--

let x = 1;

if (x === 1) {

let x = 2;

console.log(x);

// expected output: 2

}

console.log(x);

// expected output: 1

Const

const number = 42;

try {

number = 99;

} catch(err) {

console.log(err);

// expected output: TypeError: invalid assignment to const `number'

// Note - error messages will vary depending on browser

}

console.log(number);

// expected output: 42

Components

In particular, we have three components:

* Square
* Board
* Game

The Square component renders a single <button>, the Board renders 9 squares, and the Game component renders a board with some placeholders that we’ll fill in later. None of the components are interactive at this point.

### Passing Data Through Props

Just to get our feet wet, let’s try passing some data from the Board component to the Square component.

In Board’s renderSquare method, change the code to pass a value prop to the Square:

class Board extends React.Component {

renderSquare(i) {

return <Square value={i} />;

}

Then change Square’s render method to show that value by replacing {/\* TODO \*/} with {this.props.value}:

class Square extends React.Component {

render() {

return (

<button className="square">

{this.props.value}

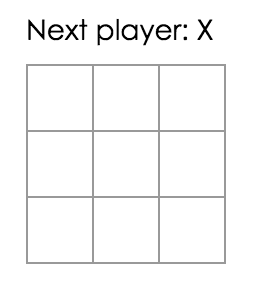
</button>

);

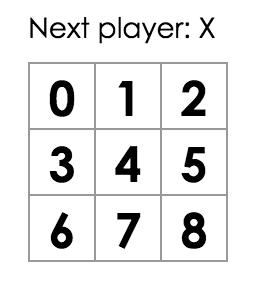
}

}

Before:

[](https://reactjs.org/static/tictac-empty-1566a4f8490d6b4b1ed36cd2c11fe4b6-a9336.png)

After: You should see a number in each square in the rendered output.

[](https://reactjs.org/static/tictac-numbers-685df774da6da48f451356f33f4be8b2-be875.png)

View the current code.

### An Interactive Component

Let’s make the Square component fill in an “X” when you click it. Try changing the button tag returned in the render() function of the Square like this:

class Square extends React.Component {

render() {

return (

<button className="square" onClick={() => alert('click')}>

{this.props.value}

</button>

);

}

}

If you click on a square now, you should get an alert in your browser.

This uses the new JavaScript arrow function syntax. Note that we’re passing a function as the onClick prop. Doing onClick={alert('click')} would alert immediately instead of when the button is clicked.

React components can have state by setting this.state in the constructor, which should be considered private to the component. Let’s store the current value of the square in state, and change it when the square is clicked.

First, add a constructor to the class to initialize the state:

class Square extends React.Component {

constructor(props) {

super(props);

this.state = {

value: null,

};

}

render() {

return (

<button className="square" onClick={() => alert('click')}>

{this.props.value}

</button>

);

}

}

Now change the Square render method to display the value from the current state, and to toggle it on click:

* Replace this.props.value with this.state.value inside the <button> tag.
* Replace the () => alert() event handler with () => this.setState({value: 'X'}).

Now the <button> tag looks like this:

class Square extends React.Component {

constructor(props) {

super(props);

this.state = {

value: null,

};

}

render() {

return (

<button className="square" onClick={() => this.setState({value: 'X'})}>

{this.state.value}

</button>

);

}

}

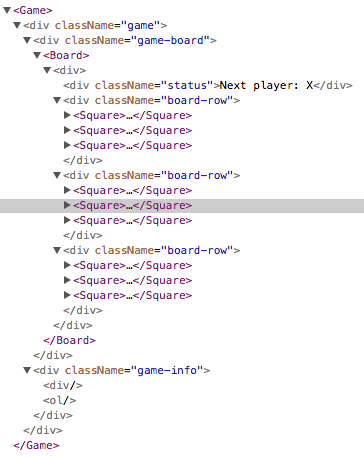
Whenever this.setState is called, an update to the component is scheduled, causing React to merge in the passed state update and rerender the component along with its descendants. When the component rerenders, this.state.value will be 'X' so you’ll see an X in the grid.

If you click on any square, an X should show up in it.

View the current code.

### Developer Tools

The React Devtools extension for Chrome and Firefox lets you inspect a React component tree in your browser devtools.

[](https://reactjs.org/static/devtools-878d91461c78d8f238e116477dfe0b46-6ca3b.png)

It lets you inspect the props and state of any of the components in your tree.

After installing it, you can right-click any element on the page, click “Inspect” to open the developer tools, and the React tab will appear as the last tab to the right.

**However, note there are a few extra steps to get it working with CodePen:**

1. Log in or register and confirm your email (required to prevent spam).
2. Click the “Fork” button.
3. Click “Change View” and then choose “Debug mode”.
4. In the new tab that opens, the devtools should now have a React tab.