03. Class in JavaScript + Introduction to React.js

Ric Huang / NTUEE

(EE 3035) Web Programming

Class in JavaScript

JavaScript Classes in ES-6

- JavaScript classes, introduced in ECMAScript 2015, are primarily syntactical sugars over JavaScript's existing prototype-based inheritance.
- The class syntax does not introduce a new object-oriented inheritance model to JavaScript.

Ref: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

Class Declarations

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

- Compared to —

```
function Rectangle(height, width) {
  this.height = height;
  this.width = width;
}
```

No class hoisting!!

```
const p = new Rectangle(); // ReferenceError!!

class Rectangle {}
```

Class Methods

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // Method
  calcArea() {
    return this.height * this.width;
  }
}

const a = new Rectangle(10, 20);
a.calcArea();  // 200
```

Compared to —

```
function Rectangle(height, width) {
  this.height = height;
  this.width = width;
  this.calcArea = function () {
    return this.height * this.width;
  }
}

const a = new Rectangle(10, 20);
a.calcArea();
```

With get/set methods

```
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 a = new Rectangle(10, 20);
a.calcArea();  // 200
a.area;        // 200
```

What's the difference between getter/setter

- "A difference between using a getter or setter and using a standard function is that getters/setters are automatically invoked on assignment. So it looks just like a normal property but behind the scenes you can have extra logic (or checks) to be run just before or after the assignment."

Ref: https://stackoverflow.com/questions/42342623/why-use-getters-and-setters-in-javascript

Static Methods

- 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.

Static Method Example

```
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
```

Be careful about "this", again!

- In the above example, since there is no caller object when “Point.distance(p1, p2)” is called, no this is defined!

Class Inheritance

- Using the keywords extends and super

```
class Animal {
  constructor(name) { this.name = name; }
  speak() { console.log(this.name + ' makes a noise.'); }
}
class Dog extends Animal {
  constructor(name) {
    super(name); // call the super class constructor and
                 // pass in the name parameter
  }
  // Overload parent class' method
  speak() { console.log(this.name + ' barks.'); }
}

let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.
```

Recall: Object Inheritance

- All objects in JavaScript inherit from at least one other object.
- The object being inherited from is known as the prototype, and the inherited properties can be found in the prototype object of the constructor.
  - Don’t get confused with the “inheritance” in class-based language(e.g. C++).
  - “class” in JS is just a syntactical sugar (supported in ES6, covered later). JavaScript remains prototype-based.

Object Inheritance vs. Class Inheritance

- myDog inherits "Animal" through the method "Object.create(new Animal("mitzie"))"
- myDog.name is an inherited properties and does NOT reside in "myDog"

```
function Animal(name) {
  this.name = name;
  this.speak = function () {
    console.log(this.name + ' makes a noise.');
  }
}
var myDog = Object.create(new Animal("mitzie"));
myDog.speak();              // "mitzie makes a noise"
console.log(myDog);         // { } <- no own property
console.log(myDog.name);// "mitzie" <- inherited
                            //              property
```

Object Inheritance vs. Class Inheritance

- console.log(myDog), you will see —

```
myDog
    Animal {}
        __proto__: Animal
            name: "mitzie"
            speak: ƒ ()
            __proto__: Object
```

- This is an object inheritance, NOT a function inheritance, so you cannot do —

```
var d = new myDog("someName");
```

Object Inheritance vs. Class Inheritance

- But if we change it to:

```
var Dog = Object.create(Animal);
```

- Dog is now a function
- However, cannot do "`var d = new Dog("Mitzie")`"

So, what does class inheritance do, exactly?

In the previous class inheritance example...

```
class Animal {
  constructor(name) { this.name = name; }
  speak() { console.log(this.name +
                        'makes a noise.'); }
}
class Dog extends Animal {
  constructor(name) {
    super(name); // call the super class
                 // constructor and pass
                 // in the name parameter
  }
  // Overload parent class' method
  speak() { console.log(this.name +
                          ' barks.'); }
}

let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.
```

- console.log(d), you will see —

```
d
  Dog {name: "Mitzie"}
    name: "Mitzie"
    __proto__: Animal
      constructor: class Dog
      speak: ƒ speak()
      __proto__: Object
```

- "name" is a property in "Dog", not in "Animal"
  => Different from previous object inheritance example

class inheritance is equivalent to the following function inheritance...

```
function Animal(name) {
  this.name = name;
  this.speak = function () {
    console.log(this.name + ' makes a noise.');
  }
}
function Dog(name) {
  Animal.call(this, name);
  this.speak = function() {
    console.log(this.name + ' barks.');
  }
}
Dog.prototype = Object.create(Animal.prototype);
let d = new Dog('Mitzie');
d.speak();
```

FYI: Function.prototype.call()

- The call() method of Function calls a function with a given **this** value and arguments provided individually.

```
function Dog(name) {
  Animal.call(this, name);
  …
}
```

- With "Dog.prototype = Object.create(Animal.prototype)"

```
d
  Dog {name: "Mitzie",
       speak: ƒ}
    name: "Mitzie"
    speak: ƒ ()
    __proto__: Animal
      __proto__:
        constructor:
              ƒ Animal(name)
        __proto__: Object
```

- Without "Dog.prototype = Object.create(Animal.prototype)"

```
d
  Dog {name: "Mitzie",
       speak: ƒ}
    name: "Mitzie"
    speak: ƒ ()
    __proto__:
      constructor:
              ƒ Dog(name)
      __proto__: Object
```

Calling super class's method

- Again, using super

```
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.`);}
}
let l = new Lion('Fuzzy');
l.speak();
// Fuzzy makes a noise.
// Fuzzy roars.
```

You've learned "class"
in JavaScript.
How can it help you?

Let’s construct a simple table

Ric's Score

Subject	Score
Math	100
Chinese	87

- Download pure HTML implementation from here.

Think: If there are going to be more columns and rows, the HTML file can become very long and thus hard to maintain.

Ideally, we would like to have a JS file like...

```
// Data source
const who = "Ric";
const columnIndex = ["Subject", "Score"];
const scoreCard = {
    name: `${who}`,
    records: [
        ["Math", 100],
        ["Chinese", 87]
    ],
};

// Define a class,
// and based on the data above,
// create the table content
```

How to define a class?

- What are the repeated parts in HTML?
- Make a proper class name
- What are the parameters for the constructor?
- What will be the properties for the class?

To define a class for the table...

```
<caption> Ric's Score </caption>
<thead>
    <tr>
        <th>Subject</th>
        <th>Score</th>
    </tr>
</thead>
<tbody>
    <tr>
        <td>Math</td>
        <td>100</td>
    </tr>
    <tr>
        <td>Chinese</td>
        <td>87</td>
    </tr>
</tbody>
```

Repeated Part
=> **class Row**

Parameters for class Row's constructor

Some useful functions...

- document.createElement(tagName);
- someNode.textContent = textString;
- someNode.appendChild(childNode);

In-Class Practice!!

【How do "class" help you?】

Any better way to write the code and make it cleaner and more extensible (maybe more object-oriented)?

Introducing...

Preparations

- Install "create-react-app"
- (建議) node.js & npm 要先安裝
- sudo npm install -g create-react-app
- In case you encounter any problem, use CodePen instead for now…

Before learning React...

- 如果說，這三個星期教的 HTML, CSS, JavaScript 讓你學會怎麼去寫一個簡單的網頁程式，
  那麼，從這星期開始教的 React 以及後面眾多單元，
  將會真正帶你進入 "Web Applications" 的奇幻旅程！

- 不只是學寫程式，而要去 「了解整個技術生態系，
  從服務設計的思維出發，學會使用最佳的工具與資源，進行最有效率的設計與開發！」

從 React 看整個技術生態系

- JSX
- Node.js, npm, yarn
- Babel
- CommonJS 等模組化開發
- Webpack
- ESLint
- React Router
- Flux, Redux
- Jest
- React Native
- GraphQL/Relay

兩個月後回來，希望你對於左邊的所有技術都瞭若指掌了！

Ref: https://github.com/kdchang/reactjs101/blob/master/Ch01/react-ecosystem-introduction.md

React.js。Basic Introductions

- React is a JavaScript library
  - 目前由 Facebook 以及 reactjs.org 所維護
  - 前端。single-page application
- First developed by Jordan Walke of Facebook in 2011, and opened source in May 2013
- React Native was released and opened source in 2015, which enabled native Android and IOS development with React

Your First React Program

```
> create-react-app hello-world
> cd hello-world
> npm start or yarn start
```

What do you see?

Check these files

- public/index.html
  - Only a in <body>
- src/index.js

```
ReactDOM.render(
  <React.StrictMode> <App /> </React.StrictMode>,
  document.getElementById('root')
);
```

- src/App.js

```
function App() {
  return (...something looks like HTML);
}
```

React.js in a glance

public/index.html

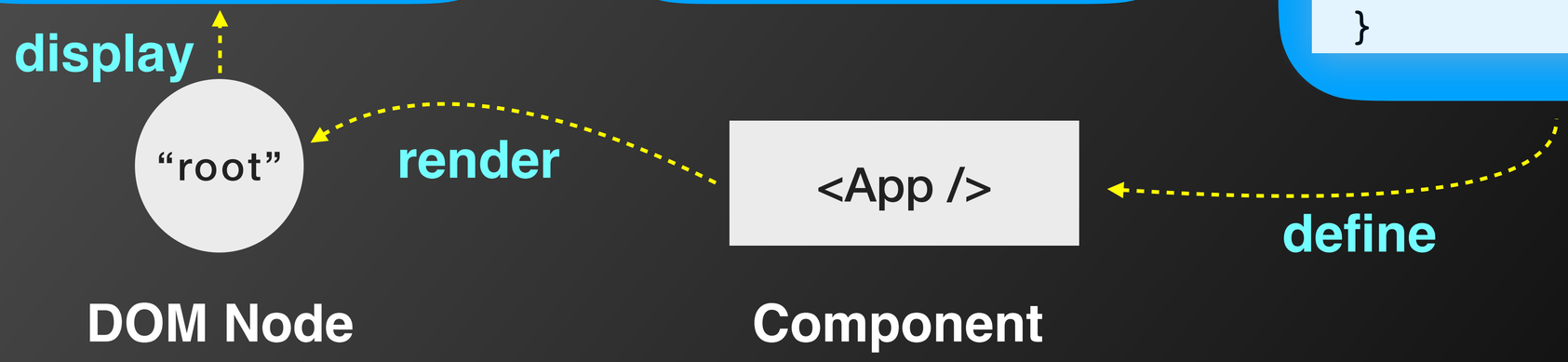
```
<body>
  <div id="root">
  </div>
</body>
```

src/index.js

```
ReactDOM.render(
  <App />,
  document.
  getElementById
  ('root')
);
```

src/App.js

```
function App() {
  return (
    <div
      className
      ="App">
      …
    </div>
  );
}
```



What did "create-react-app" do for you?

```
> npm init projectName
> npm install
// what you need to run React,
// including all the modules, webpack configure,
// Babel... etc
And prepare all scripts for you to run React Apps
```

- If you click to open "index.html" in browser, what did you see?

Edit the file "src/App.js" to print out "Hello, World!"

- Delete lines 7-20
- Change returned context to "**`<h1>Hello, world!</h1>`**"
- Save it, and you will see the webpage automatically reloaded!

Or, simply do —

- Change this line in "index.js":

```
ReactDOM.render(
    <App />, document.getElementById('root'));
```

- To —

```
// App.js won't be called
ReactDOM.render(
    <h1>Hello, world!</h1>,
    document.getElementById(‘root')
);
```

Try to explain
what you see!

React Basics

- React 是一種 component-based 的寫法 — 把網頁的 DOM 想成一個個的 components, 然後利用 JSX 的語法，把每個 component 寫成 React element, 就像是在 JavaScript 裡頭直接寫 HTML 一樣，然後利用 ReactDOM 的 render() method 把 React element 畫到 index.html 對應的節點上面:

```
// In JavaScript/React
ReactDOM.render(
    <h1>Hello, world!</h1>,
    document.getElementById('root')
);
```

```
<!-- In HTML -->
<body>
    <div id="root"></div>
</body>
```

Virtual DOM

- JavaScript 裡頭對於 DOM element 的產生與操作是很慢的；React 則使用了 Virtual DOM 的概念，去 monitor 頁面改變的地方，而當改變發生的時候，只重新 render 改變部分，因此，可以大幅提升畫面更新的速度

Virtual DOM Example

- Try this on index.js and watch it on console:

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is
        {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(element,
    document.getElementById('root'));
}
setInterval(tick, 1000);
```

Classes in React

- Hello-world 的範例也可以用 React 的 "class" 來改寫
- 原先：

```
function App() {
  return (...something looks like HTML);
}
```

- 改成：

```
import React from 'react';
class App extends React.Component {
  // 一定要有一個 render() method
  render() {
    // 回傳很像是 html 的 jsx
    return <h1>Hello, World</h1>;
  }
}
```

Classes in React

- 然後在 JS 的主程式：

```
ReactDOM.render(
    <App />, document.getElementById('root'));
```

Quick Summary — How React works?

- 用 HTML 把網頁的殼子畫出來
- 用 JSX 的語法定義 React elements (as DOM components)

```
// expression
<tag>text or { expression }</tag>
// or class
class ClassName extends React.Component {
  render() { ... return...; }
}
```

```
ReactDOM.render(
    <ClassName />, document.getElementById('root'));
```

ReactDom.render()

- 語法

```
ReactDOM.render(element, container[, callback]);
```

- 其中 element 除了可以直接把 JSX 語法寫在參數上面之外，也可以是一個 JavaScript 的 variable：

```
let e = <h1>Hello, world!</h1>;
ReactDOM.render(e,
                document.getElementById('root'));
```

- 而 container 則是一個 DOM node
- callback 顧名思義則是在 render() 完成後被呼叫

JSX

在 JS 檔中用像是在寫 HTML 的方式

產生 JavaScript 的 DOM Node

【JSX (JavaScript XML)】

An extension to the JavaScript language syntax to provide a way to structure component rendering using syntax familiar to many developers (i.e. HTML/XML).

JSX to create React Elements

- JSX 背後其實是用 React.createElement() 去做轉換的

```
// Input (JSX):
var app = <Nav color="blue" />;
// Output (JS):
var app = React.createElement(Nav, {color:"blue"});
```

```
// Input (JSX):
var app = <Nav color="blue">
          <Profile>click</Profile></Nav>;
// Output (JS):
var app = React.createElement(
  Nav,
  {color:"blue"},
  React.createElement(Profile, null, "click")
);
```

Embedding JS Expressions into JSX

- 可以在適當的地方用 {…} 插入任何 JavaScript 的 expressions:

```
const e1 = <h1> Hello, {iAmAFunction(pp)}! </h1>;
const e2 = <img src={user.avatarUrl}></img>;
const e3 = <p> 2 + 3 = { 2+3 } </p>
let e4;
if (someExp)
    e4 = <h1>Hello, {iAmAFunction(pp)}!</h1>
else e4 = <h1>Hello, world!</h1>
```

Specifying Tag Attributes with JSX

- 指定 JSX tag 裏頭 attribute 的值
- 請注意，JSX 裏頭 tag 的名稱為了不要跟 JavaScript 裏頭的保留字衝突，會換成別的名稱，且會變成 camelCase

```
// as "class" in HTML
<div className="foo" />
// as "for" in HTML
<label htmlFor="username">Username</label>
<MyButton disabled={false} onClick={() => {}} />
```

常犯錯誤

```
render() { // 不能回傳並列的 elements
  return (
    <div>Hello 1</div>
    <div>Hello 2</div>
  );
}
```

- 只能有一個 root element

```
render() { // 包進 div
  return (
    <div>
      <div>Hello 1</div>
      <div>Hello 2</div>
    </div>
  );
}
```

常犯錯誤

- 用 " " 把 { } 的 JS expression 括起來 => 會變字串
- 用 ( ) 把 { } 的 JS expression 括起來 => 會多生出 ( ) 符號，或者是文法錯誤！
- 或是忘記加 { }

```
// extra { }
<A disabled="{false}" onClick={() => {}} />
// extra ( ); output: (World!)
<A> ({ "World!" }) </A>
// Missing { }; output: ("World!")
<A> ("World!") </A>
// Can’t be a statement; ERROR: extra ';'!
<A> { "World!"; } </A>
```

Parameters (props) for React Components

- 如同 function/class 在定義一個 object 的 prototype 時可以傳入參數來定義其 properties; React 也可以利用 'props' 這個保留字，用來指定 React component 的 properties
- function 的用法： // 不建議去定義太複雜的 object

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

```
ReactDOM.render(<Welcome name="Ric" / >,
document.getElementById('root'));
```

React Components with class properties

- 用 class 裡頭的 this.props 來承接各種 component 的 properties (e.g. name), 然後在外部的 instantiation 利用 tag value assignment 來指定 property value
- 建議寫法：

```
import React, { Component } from 'react';
class Welcome extends Component {
  render()
  { return <h1>Hello, {this.props.name}</h1>; }
}

ReactDOM.render(<Welcome name="Ric" / >,
document.getElementById('root'));
```

搞得我好亂啊！到底是在寫 JS, JSX, 還是HTML...

!%#&^%@&$%@!

Rules of thumb

- 到底是 JSX, 還是 JS？ Browser 的 interpreter engine 會幫你分辨，基本上：
  - 這是 .js 檔，所有的語法要 follow JS
  - 但當某個 expression 被 HTML-like tag (e.g. <ScoreCard>) 包起來的時候，就進入了 JSX 的範疇，你要以 JSX 的語法來寫 (比較像 HTML/XML)，像是：
    - <tag>what to be shown</tag>
    - <tag>something { JS expression } </tag>
  - 多餘的 { }, " ", ( ) 都有可能變成 viewable 的一部分 (e.g. <tag>(extra braces)</tag>)

Let's do some Practice

Ric's Score

Subject	Score
Math	100
Chinese	87

- What's the DOM structure for this example?

Change public/index.html

- create-react-app for ScoreCard example
- Copy "styles.css" to public and modify <head /> in index.html accordingly
- Make a "root" ready for ReactDOM components

```
<body>
   <!-- will insert ReactDOM component here -->
   <!-- wrap the <table> with a <div>       -->
  <div id="root"></div>
</body>
```

Change src/index.js

- Make the record extensible, and can be applied to other students

```
import...;

const columnIndex = [ 'Subject', 'Score' ];
const scoreCard = {
  name: 'Ric',
  records: [
    [ 'Math', 100 ],
    [ 'Chinese', 87 ],
    [ 'English', 100 ],
    [ 'Science', 100 ],
    [ 'Social', 0 ]
  ]
};

ReactDOM.render(<App / >, document.getElementById('root'));
```

Change src/App.js

- Define class ScoreCard

```
// in App.js, change class App to ScoreCard
class ScoreCard extends Component {
  render() {
    return (
      <table>
      ...
      </table>
    );
  }
}
```

- The question is: How to "receive" data from the variables defined in index.js
  => Define properties for "this props" accordingly!

Link JSX to HTML

- Change "index.js"

```
import ScoreCard from './App';
ReactDOM.render(<ScoreCard scoreCard = {scoreCard}
                  columnIndex = {columnIndex} / >,
                  document.getElementById('root'));
```

- scoreCard is the attribute name of the class <ScoreCard>
- {scoreCard} refers to the variable in JS, so { } is needed
- Change "App.js"

```
class ScoreCard extends Component {
  render() {
    return (
      <table>
        <caption>{this.props.scoreCard.name}'s
                  Score</caption>
```

From data to ReactDOM element

- Objective —

```
  return (
  <table>
    <caption> Ric's Score </caption>
    <thead>
      <tr><th> Subject </th><th> Score </th></tr>
    </thead>
    <tbody>
      <tr><td> Math </td><td> 100 </td></tr>
      <tr><td> Chinese </td><td> 87 </td></tr>...
    </tbody>
  </table>
);
```

From data to ReactDOM element

- In index.js

```
const columnIndex = [ 'Subject', 'Score' ];
ReactDOM.render(<ScoreCard scoreCard = {scoreCard}
                columnIndex = {columnIndex} / >,
                document.getElementById('root'));
```

- In App.js

```
class ScoreCard extends Component {
  render() {
    return (
      <table>
        <caption>...</caption>
        <thead>
          <tr> { this.props.columnIndex.map
                 ( e => <th>{e}</th> ) } </tr>
        </thead>
```

What about
"scoreCard.records" ?

Reference solution

- Create a local variable for each row…

```
// This will create:
// [ [ subjectNode, scoreNode]...], where
//      subjectNode = <td>{'subject'}</td>,
//      scoreNode   = <td>{ score }</td>
let records = this.props.scoreCard.records.map
    ( e => e.map( g => <td>{g}</td> ) );

// In the return part of render()…
    <tbody>{ records.map( e => <tr>{e}</tr>) }...
```

Reference solution

- Or simply do —

```
return (
  <table>
    ...
    <tbody>{ this.props.scoreCard.records.map(
      r => <tr> { r.map(e => <td>{e}</td>) }</tr>
    )}</tbody>
  </table>
);
```

Did you make it?

Before we continue...

- 我們的 code 裡頭大量的用到 "import", "export"...
- In index.js

```
import React from 'react';
import ScoreCard from ‘./App';
```

- In App.js

```
import React, {Component} from 'react';
class ScoreCard extends React.Component {
   ...
}
export default ScoreCard;
```

=> JS modules 之間的 function/class 分享，是 follow 一個叫做 "commonJS" 的規範

CommonJS 規範

- CommonJS 的誕生是為了要讓眾多的 JS modules 有一個共同的標準，得以彼此共生在 browser 以外的不同環境底下，建立應用生態系
- 主要包含了 模組規範、套件規範、I/O、File System、Promise 等
- Node.js 就是 CommonJS 的一個主要實踐者

CommonJS 規範

- CommonJS 是在 runtime 加載(require) modules

```
let { stat, exists, readFile } = require('fs');
const math = require('math');
```

- 然後就可以用了：

```
console.log(math.add(1, 2));
```

- 至於輸出模組，則用 "exports.functionName"

```
exports.incrementOne = function (num) {
  return math.add(num, 1);
};
```

CommonJS 規範。ES6

- ES6 則是強調「靜態時」就要決定模組的相依性
- By "export" & "import"

```
export var firstName = 'Michael';
export function multiply(x, y) { return x * y; }
       as MM;
export class MyClass extends React.Component...;
```

- from 後面的 path 可以是絕對或是相對位址; '.js' 可省

```
import { foo } from './myApp.js';
import { add, sub } from './myMath.js';
import { aVeryLongName as someName }
       from '../someFile.js'
```

"export default"

- 在前面的例子當中，使用者需要知道 import 進來的檔案裡頭原先的那些變數、function、class 的名字為何，需要跟原來檔案裡頭定義的名字一樣，才可以使用
  - 而且 import 時要記得加 { }
- "export default" 則讓我們可以不用管原來檔案裡頭這些 function/class 叫什麼名字，甚至是可以 anonymous

```
export default (a, b) => (a+b);
```

"export default"

- 不過既然 function/class 都可以 anonymous 了，所以：
  - export 的檔案就只能有一個 "export default" 的 function or class
  - 在 import 時的名字是屬於 import 那個檔案的 scope，且不可以加 { }
  - from 後面的檔案名稱可以把 .js 省略

```
export default (a, b) => (a+b); // myMath.js
```

```
import myAdd from myMath;  // myAdd 可以是隨便名字
```

比較這兩種寫法

- Specifically state that the class is extended from React.Component

```
import React from 'react'
class MyClass extends React.Component { ... }
```

- "React" is an "export default", while "Component" is a regular export

```
import React, {Component} from 'react'
class MyClass extends Component { ... }
```

Quick Review: React.js Basics

- Use "ReactDOM.render()" to update DOM

```
const element = ...; // some JSX expression
const node = document.getElementByID(...);
reactDOM.render(element, node);
```

- The "element" can be defined from a function or a class.

```
class MyElement { ... }
const element = <MyElement />;
reactDOM.render(element, node);
```

Use “/>” to close a tag
if no text content is involved.

不能用 new MyElement()

Quick Review: React.js Basics

- In index.js
- 使用 this.props 以及 tag attribute 來傳遞參數

```
const columnIndex = [ 'Subject', 'Score' ];
ReactDOM.render(<ScoreCard scoreCard = {scoreCard}
                columnIndex = {columnIndex} / >,
                document.getElementById('root'));
```

- In App.js

```
class ScoreCard extends Component {
  render() {
    return (
      <table>
        <caption>...</caption>
        <thead>
          <tr> { this.props.columnIndex.map
                 ( e => <th>{e}</th> ) } </tr>
        </thead>
```

一定要定義 “render()” 這個 method

只能 return 一個 tag

Note: "this.props" are read-only

- You cannot assign or change values to this.props

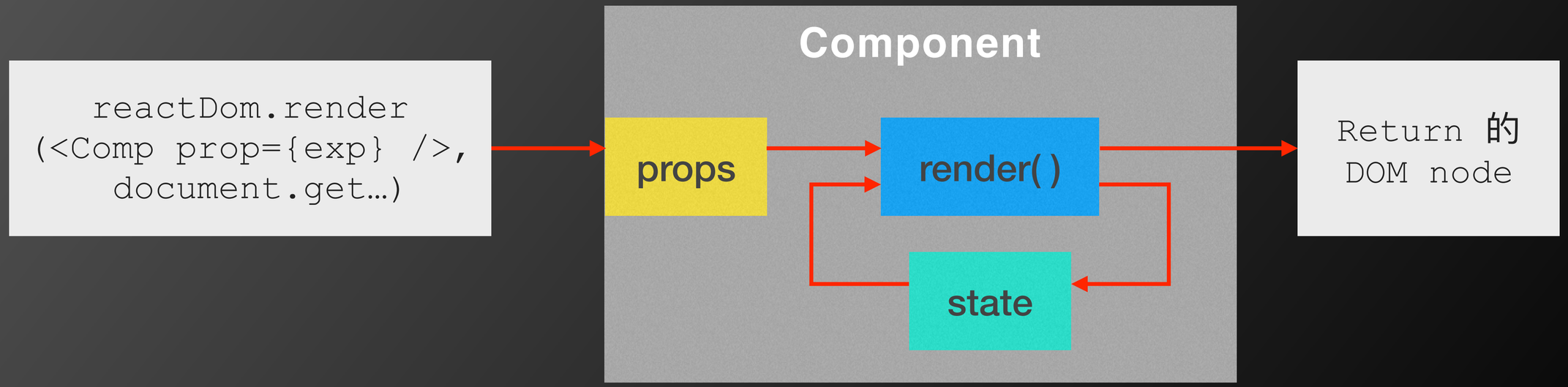
```
class Caption extends Component {
  Constructor(name} {
    this.props.name = name;
  }
  render() {
    return <caption> {this.props.name}'s
           Score </caption>;
    }
}
```

- "this.props" 是由 <Caption /> 的參數傳入時所設定，一但設定，就不能再改 (read-only)

What can we do if we want to change the React components dynamically according to some I/O events?

"this.state" in React Component

- 如果想要在 component 裡頭 "記住，進而可以變動" 一些資訊(e.g. 按讚數量, 目前計算結果...)，用來增加網頁的互動，則需要用 “state”



Understanding React “state"

- 要了解 React component 裡頭的 "state"，我們要同時學三件事情：

1. this.state
2. Component lifecycle
3. Event handling

"this.state"

- 語法

```
class MyClass extends React.Component {
  constructor(props) {
    super(props);
    this.state = {  // as an "object" declaration
      statePropName: statePropValue,
      ...
    };
  }
}
```

- e.g.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = { date: new Date() }; // initialize
  }
}
```

Why not “this.date”?

Referred by "this.state..."

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = { date: new Date() };
  }
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is
           {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
ReactDOM.render(<Clock / >,
document.getElementById('root'));
```

- 但... clock 不會動了... (fixed later)

當然，我們可以試著這樣做...

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = { date: new Date() };
  }
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
setInterval(
  () => ReactDOM.render(<Clock / >,
        document.getElementById('root')), 1000);
```

- 但這是不 work 的... Virtual DOM 並不會知道 state 被 update 而需要更新畫面

Note: "state" is private to the class

- You cannot pass in value to "state"

```
class MyClass extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 }
  }
  render() {
     return (
       <div> {this.props.name} is called
             {this.state.count} times!
       </div>
     );
  }
}
ReactDOM.render(<MyClass name="Ric" count={8} / >,
                document.getElementById('root'));
```

- Output: "Ric is called 0 times!" // No error, not '8'

- So, "this.state" is initialized in constructor. How do we update it?
- Or say, once the React component is rendered on the screen, how do we re-render it?
- To get a better understanding on how React state works, we should look at "component lifecycle" first!

Component Lifecycle

- 一個 React component 從一開始被定義、render 到螢幕、因為 "某些因素" 讓 virtual DOM 察覺到 component 的內容改變而 re-render 畫面、到最後這個 component 從 DOM 被拔掉，這個 component 總過會經歷過三個階段的 lifecycle —

1. Mounting: component 即將被生成並且插入到 DOM 上面 (i.e. displayed on page)
2. Updating: 因為 props or state 的改變而 trigger virtual DOM 去更新畫面
3. Unmounting: component 即將從 DOM 被移除

To be more specific。Mounting

- 當一個 component 的 instance 被建立且加入 DOM 中時 (i.e. display on page)，其生命週期將會依照下列的順序呼叫這些方法

```
constructor()
static getDerivedStateFromProps()
render()
componentDidMount()
```

- 下列方法已過時，你在寫新程式應避免使用：

```
UNSAFE_componentWillMount()
```

To be more specific。Updating

- 當 prop 或 state 有變化時，就會產生更新。當一個 component 被重新 render 時，其生命週期將會依照下列的順序呼叫這些方法：

```
static getDerivedStateFromProps()
shouldComponentUpdate()
render()
getSnapshotBeforeUpdate()
componentDidUpdate()
```

- 下列方法已過時，你在寫新程式應避免使用：

```
UNSAFE_componentWillUpdate()
UNSAFE_componentWillReceiveProps()
```

To be more specific。Unmounting

- 當一個 component 被從 DOM 中移除時，這個方法將會被呼叫：

```
componentWillUnmount()
```

Error Handling

- 當一個 component 在 render 的過程、生命週期、或在某個 child component 的 constructor 中發生錯誤時，這些方法會被呼叫：

```
static getDerivedStateFromError()
componentDidCatch()
```

Why should I care about the component lifecycle?

- 利用 "Mounting" 階段中的 methods 初始化 props/state 的值，並且做一些必要的設定
- 透過 event handling 更新 states 的值 (why not "props"?)，並且在 updating 階段中 re-render component
- (如有必要) 當 component 要被從 DOM 移除之時，在 unmounting 階段把一些資源回給系統

Use "setState()" to update state!!

- 用 setState() 才會去通知 virtual DOM 重新呼叫 render() 來更新畫面

- 如果這樣寫：

```
tick() {
  this.state.date = new Date();
}
```

- // 不會有 error message, 但 clock 不會動！

In the previous clock tick example...

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = { date: new Date() };
  }
  componentDidMount() {
    setInterval(() => this.updateTime(), 1000);
  }
  updateTime() {
    this.setState({ date: new Date() });
  }
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
ReactDOM.render(<Clock / >, document.getElementById('root'));
```

A Closer Look...

- 在 component mount/render 完畢之後設定 "this.updateTime()" 每秒會被呼叫一次

```
componentDidMount() {
    setInterval(() => this.updateTime(), 1000);
}
```

- 更新 state 的 value

```
updateTime() {
    this.setState({ date: new Date() });
}
```

- Note: "ReactDOM.render()" 不必重新被呼叫！！更新畫面是 virtual DOM 的事，你只要更新 state value 就好了！

Release System Resources

- 在前面的範例中，component 的更新是藉由 system clock tick, 所以如果 component 因故被從 DOM 拔掉，即使畫面不再顯示這個 component，它的 tick() 還是會被一直呼叫。
- 可以宣告一個變數把產生的 timer ID 存下來，然後再 unmounting phase 把它停掉

```
componentDidMount() {
    this.timerID = setInterval(() => this.tick(), 1000);
}
componentWillUnmount() {
    clearInterval(this.timerID);
}
```

Using React Event Handler

- 在前面的範例中，component 的更新是藉由 system clock tick, 但在一般的應用當中，常常是由 I/O events 來觸發畫面的更新
- Since React is just a JS library, 它的 event handling 基本上跟 JS 差不多...

Recall: Event handling in HTML/JS

1. addEventListener()

```
var targetElement = document.getElementById("target");
targetElement.addEventListener("click", function() {...});
```

2. GlobalEventHandlers

```
let log = document.getElementById('log');
log.onclick = inputChange;
function inputChange(e) {...}
```

3. As a tag attribute

```
<div class="myClass" onclick="clickHandler()">
function clickHandler() {...}
```

In React, we can usually do...

```
class MyButton extends React.Component {
  render() {
    return (
      <button onClick=
        {()=>console.log('this is:', this)}>
        Click me
      </button>
    );
  }
}
ReactDOM.render(<MyButton / >,
  document.getElementById('root'));
```

Or, create a class method...

```
class MyButton extends React.Component {
  handleClick = () => {
    console.log('this is:', this);
  }
  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
ReactDOM.render(<MyButton / >,
  document.getElementById('root'));
```

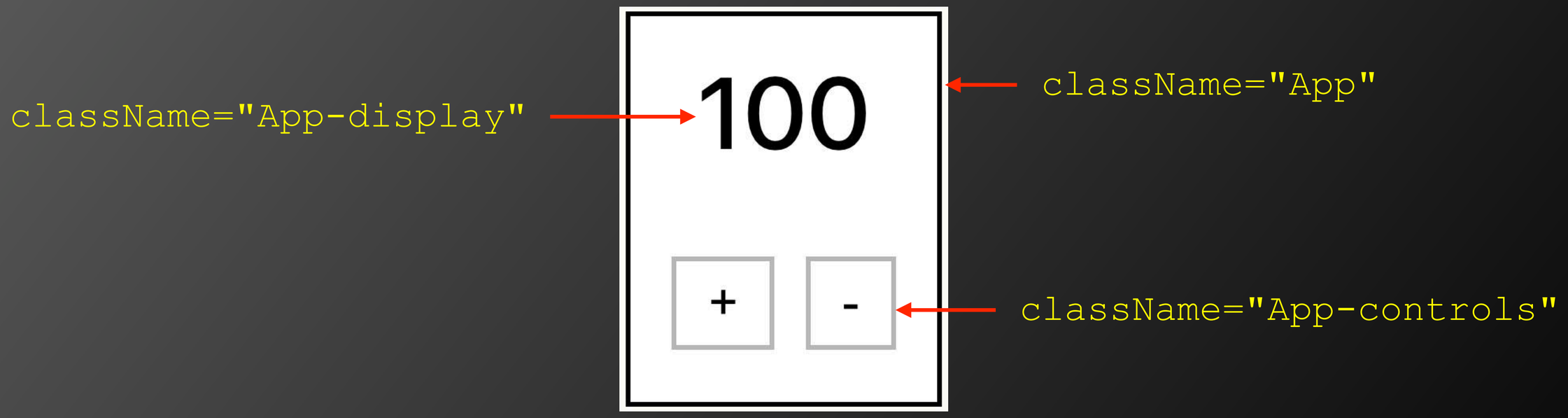
Recall: Specifying Tag Attributes with JSX

- JSX 裏頭 tag 的名稱為了不要跟 JavaScript 裏頭的保留字衝突，會換成別的名稱，且會變成 camelCase

```
// as "class" in HTML
<div className="foo" />
// as "for" in HTML
<label htmlFor="username">Username</label>
<MyButton disabled={false} onClick={() => {}} />
```

In-class Practice

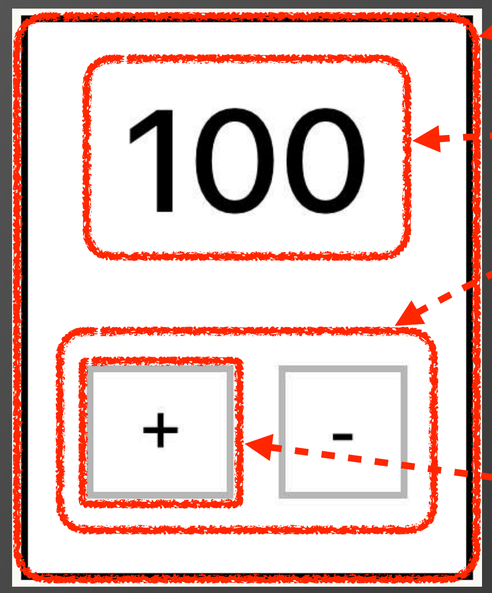
- 用 React 實作一個可以 加/減 的計數器

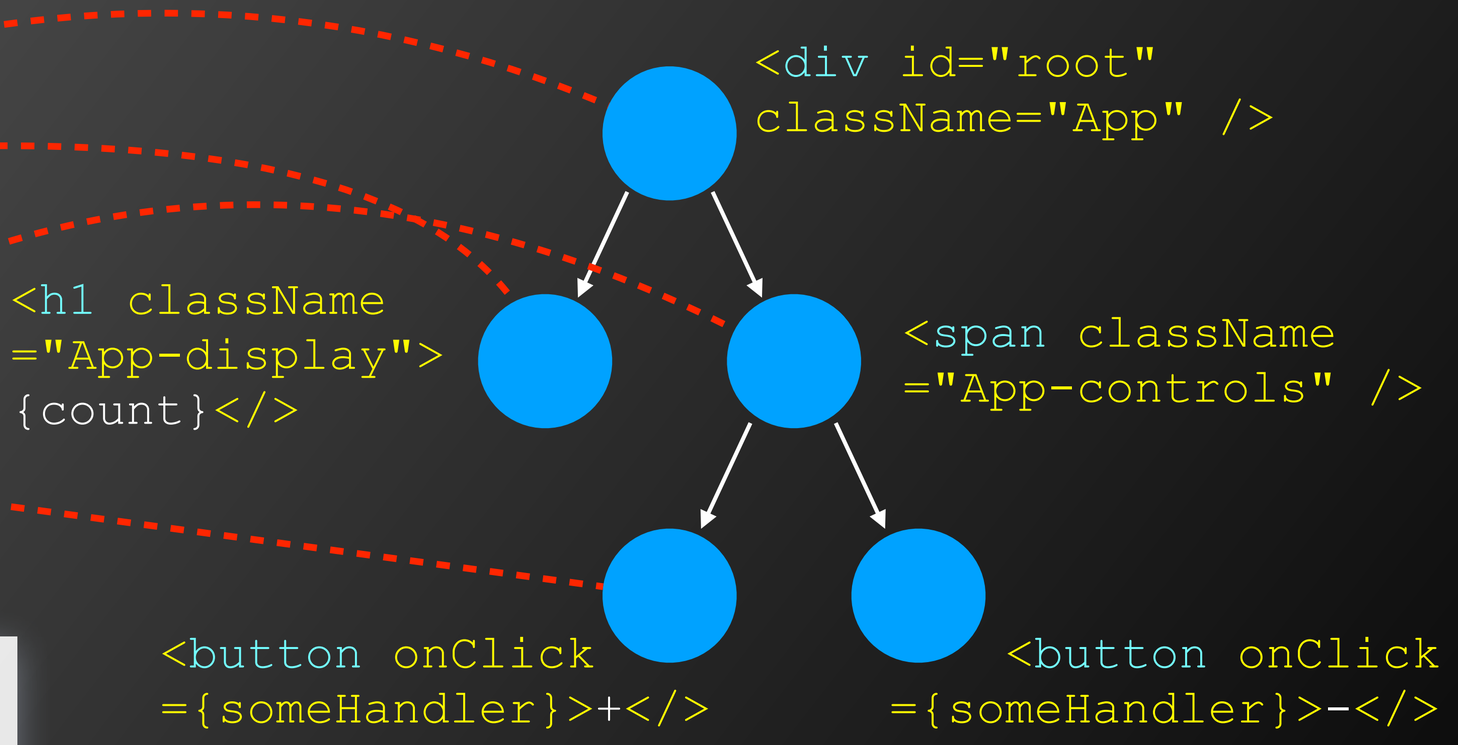


- 從 Ceiba 下載 "App.css" 以及 "index.css" 取代原來的檔案
- 初始值：100, 按 '+' 則 +1, 按 '-' 則 -1
- 至少 create 一個 class "Counter"

防雷頁…

1. 先寫好 index.html, 並且規劃好 DOM structure

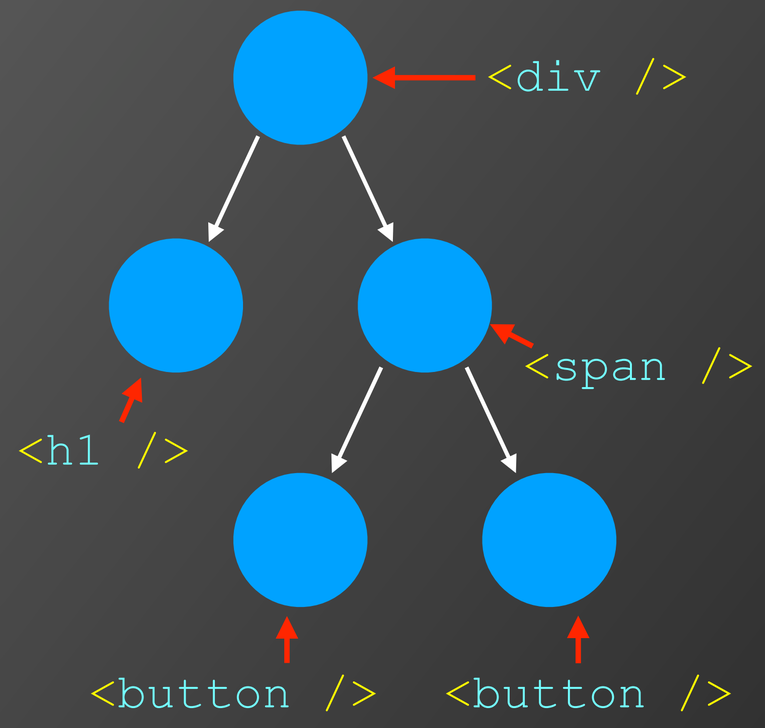





```
<html>
  <body>
    <div id="root" />
  </body>
<html>
```

2. 用 JSX 產生靜態頁面

寫好 src/App.js, 定義一個 top level class, 並且用它的 render() 來產生 DOM




```
import React, {Component} from 'react';
class Counter extends Component {
  render() {
    return (
      <div className="App">
        <h1 className="App-display">100</h1>
        <span className="App-controls">
          <button>+</button>
          <button>-</button>
        </span>
      </div>
    );
  }
}
export default Counter;
```

JavaScript / React class

HTML/CSS class

先寫成靜態的值，
有畫面再說

3. 把 JSX 與 HTML/CSS 串起來

用一個 src/index.js 來把 class Counter 與 HTML 串起來

先 "yarn start"，看看是否有正常看到畫面

public/index.html

```
<html>
  <body>
    <div id="root" />
  </body>
<html>
```

src/index.js

```
import Counter from './App';
ReactDOM.render(
 <Counter />,
  document.getElementById('root')
);
```

src/App.js

```
class Counter extends Component {
  render() {
    return (
      <div className="App">
        ...
      </div>
    );
  }
}
```

4. 定義一個 Counter 的 state { count: 100 }

由於 {counter} 的值會 depends on 動作之前一刻的值 (i.e. stateful), 所以它應該要是 class Counter 的一個 state value

```
class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 100 };
  }
  render() {
    return (
      <div className="App">
        <h1 className="App-display">{this.state.count}</h1>
        …
      </div>
    );
  }
}
```

this.state 是一個物件，所以用 { } 來初始化其值

<div…> 進入 JSX 的語法範圍

- HTML tag 的內文
- 由於內文是從 "變數" 來動態決定，所以用 { } 來表示 JS 的 expression

5. 定義 handler functions for button onClick events

先來個錯誤示範...

```
class Counter extends Component {
  render() {
    return (
      <div className="App">
        <h1 className="App-display”>{this.state.count}</h1>
        <span className="App-controls">
          <button onClick={()=>this.state.count++}>+</button>
          <button onClick={()=>this.state.count--}>-</button>
        </span>
      </div>
    );
  }
}
```

Recall: state 的更新要用 setState(), 不能直接設定！！

5. 定義 handler functions for button onClick events

再來個錯誤示範...

```
class Counter extends Component {
  render() {
    return (
      <div className="App">
        <h1 className="App-display”>{this.state.count}</h1>
        <span className="App-controls">
          <button onClick=
              {()=>this.setState(this.state.count + 1)}>+</button>
          <button onClick=
              {()=>this.setState(this.state.count - 1)}>-</button>
        </span>
      </div>
    );
  }
}
```

Recall: state 是個 object！！

5. 定義 handler functions for button onClick events

錯誤示範 #3…

```
class Counter extends Component {
  render() {
    return (
      <div className="App">
        <h1 className="App-display”>{this.state.count}</h1>
        <span className="App-controls">
          <button onClick={()=>this.setState
                  ({count: this.state.count + 1})}>+</button>
          <button onClick={()=>this.setState
                  ({count: this.state.count - 1})}>-</button>
        </span>
      </div>
    );
  }
}
```

咦！可以 work 啊！錯在哪裡？？

5. 定義 handler functions for button onClick events

根據 React 的官方說法，State Updates May Be Asynchronous

```
// Wrong: state 的 value 可能沒有被 update 到
this.setState({
  counter: this.state.counter + this.props.increment
});
```

```
// Correct: 這樣才會拿 previous state 的值來 update
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

State Updates May Be Asynchronous

Try this…

```
class Counter extends Component {
  handlePlus2 = () => {
    this.setState({count: this.state.count + 1});
    this.setState({count: this.state.count + 1});
  }
  render() {
    return (
      <div className="App">
        <h1 className="App-display">{this.state.count}</h1>
        <span className="App-controls">
          <button onClick={this.handlePlus2}>+2</button>
          <button onClick={()=>this.setState
                 ({count: this.state.count + 1})}>+</button>
          <button onClick={()=>this.setState
                 ({count: this.state.count - 1})}>-</button>
        </span>
      </div>
    );
  }
}
```

State Updates May Be Asynchronous

Note: JS 的 statements 是 non-blocking 的依序執行

```
handlePlus2 = () => {
    this.setState({count: this.state.count + 1});
    this.setState({count: this.state.count + 1});
    console.log("在這邊 console.log 試試看");
}
```

先 evaluate,
兩者都會變成
{count: 101}

而所謂的 asynchronous 的執行，就是把 async functions (通常是 batch 的方式)丟出去給這個執行 async functions 的 engine, 執行完畢之後，再用 callback 通知主程式

```
// 所以 React engine 收到的就是 –
setState({count: 101});
setState({count: 101});
```

5. 定義 handler functions for button onClick events

```
class Counter extends Component {
  handleInc = () => this.setState
                    (state => ({ count: state.count + 1 }));
  handleDec = () => this.setState
                    (state => ({ count: state.count - 1 }));
  render() {
    return (
      <div className="App">
        <h1 className="App-display”>{this.state.count}</h1>
        <span className="App-controls">
          <button onClick={this.handleInc}>
              +</button>
          <button onClick={this.handleDec}>
              -</button>
        </span>
      </div>
    );
  }
}
```

Closer look on the onClick's handler

```
<button onClick={this.handleInc}>
```

- 因為要綁定一個 function, 所以用 { } 進入 JS 的 expression, 且不能寫成 this.handleInc(), 否則就變成先呼叫 function 後的 return 值了

```
handleInc = () => this.setState
            (state => ({ count: state.count + 1 }));
```

- 用 arrow function, 因為要 return 一個 function 給 handleInc

一定要用 arrow function 嗎？

Try this...

- 這樣寫會給 “Failed to compile” (Why?)

```
function handleInc() {
    this.setState
        (state => ({ count: state.count + 1 })); }
```

- 這樣寫 compile 會過，但按了 ‘+’ 號後會有 “TypeError: Cannot read property 'setState' of undefined” 的 error (Why?)

```
handleInc
= function() {
    this.setState
        (state => ({ count: state.count + 1 })); }
```

Recall: ‘this’ refers to the function scope

this and bind()...

- 如果堅持要用前頁 function 的寫法，一個解決的辦法是在 caller 把 this bind() 起來！

```
<button onClick={this.handleInc.bind(this)}>
```

- 實在是 awkward… 好險現在有 arrow function
  => arrow function 裡頭的 this refers to the caller's scope

Closer look on the onClick's handler

```
handleInc = () => this.setState
            (state => ({ count: state.count + 1 }));
```

為什麼不用 'this'?

- this.setState() 吃的參數是一個 "stateUpdateFunction" (所以要用 arrow function), 而這個 function 吃一個參數 (i.e. local variable)，setState 會把 current this.state assign 給它
- 所以，也可以寫成：

```
handleInc = () => this.setState
            (s => ({ count: s.count + 1 }));
```

但，"count" 不能改成別的名字！ (why?)

And remember...

- “props” is pure. It should be read-only.
  - It's value is assigned when passed through tag attribute and should remain unchanged afterwards.
- “state” is private. You should use “this.setState()” to update state's value.
  - Otherwise, it won't trigger updates on VDOM.

6. 用 functional component 把 logic 跟 component 分開

```
// in "App.js"
import React, { Component } from 'react'
import Button from '../components/Button'
...
    <Button text="+" onClick={this.handleInc} />
    <Button text="-" onClick={this.handleInc} />
```

```
// 加一個 “src/components/Button.js”
import React from 'react'
export default ({ onClick, text }) => {
  return <button onClick={onClick}>{text}
         </button>;
}
```

Closer look at the functional component...

- Recall: 當上層的邏輯 (e.g. containers/App.js) 呼叫下層的 components 時，是用 JSX 與 tag attributes 打包成 object 傳給 component 的 props.

```
class Welcome extends Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
const element = <Welcome name="Ric" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

{name: "Ric"}

Closer look at the functional component...

- 如果改寫成 function...

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
const element = <Welcome name="Ric" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

- 事實上，props 只是 local variable (function argument), 你要把它改成 'p' 也是可以的 (但寫成 props 才會符合一些 linter 的規則)

Closer look at the functional component...

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

- 改寫成 functional component…

```
// components/Welcome.js
export default
(props) => return <h1>Hello, {props.name}</h1>;
```

- 應用 destructuring assignment 的概念，簡化成：

```
// components/Welcome.js
export default
({name}) => return <h1>Hello, {name}</h1>;
```

Functional (Dummy) Components

- 在前面的例子，整個畫面的主要邏輯都是寫在 Counter 裡面，而 Button 的角色只是個 component，也沒有自己的 state. 因此，建議可以將這兩種 components 分開，放在底下兩個子目錄：

1. Containers: 如：Counter, 存著 state/props 以及一些主要的邏輯
2. Components: 如：Button, 沒有自己的 states, 也沒有什麼複雜的邏輯，建議改寫成 Functional Component

Button as a Functional Component

- In Components/Button.js

```
import React from 'react'
export default ({ onClick, text }) => {
  return <button onClick={onClick}>{text}
         </button>;
}
```

- In Containers/App.js

```
import React, { Component } from 'react'
import Button from '../components/Button'
```

試試看在下面寫一個
input box 來設定
counter 的值

【關於 state 一些注意事項】

https://reactjs.org/docs/state-and-lifecycle.html#using-state-correctly

1. Do Not Modify State Directly

```
// Wrong: this won't re-render the component
this.state.comment = 'Hello';
```

```
// Correct: use "setState()"
this.setState({comment: 'Hello'});
```

- The only place where you can assign this.state is the constructor.

2. State Updates May Be Asynchronous

- React may batch multiple setState() calls into a single update for performance.
- Because this.props and this.state may be updated asynchronously, you should not rely on their values for calculating the next state.

```
// Wrong: state 的 value 可能沒有被 update 到
this.setState({
  counter: this.state.counter + this.props.increment
});
```

```
// Correct: 這樣會拿 previous state 的值來 update
this.setState((state, props) => ({
 counter: state.counter + props.increment
}));
```

3. State Updates are Merged

- 你可以針對 state object 裡頭不同的 properties 分開來 update

```
componentDidMount() {
    fetchPosts().then(response => {
        this.setState({
            posts: response.posts
        });
    });
    fetchComments().then(response => {
        this.setState({
            comments: response.comments
        });
    });
}
```

- 上述兩個 update 可以被獨立呼叫，不會互相影響

4. Data Flows Down

- State is local and encapsulated. 任何一個 component 不會知道它的 parent or child components 是 stateful or stateless. 它也無法去讀取它 parent or child component states 的值
  - 就像 Button 無法去讀 Counter 的值
- 所以，我們通常會把 state 往上 (in terms of DOM structure) 提，然後如果 child component 的 view 會 depend on parent component state 的值，則在 child component 宣告的地方把 parent state 的某個 prop 傳給 child props.

```
// In some parent component's function
   <ChildComponent someProp={this.state.someProp} / >
```

React Reference Readings

- From official reactjs.org website
  - [Getting started] (https://reactjs.org/docs/getting-started.html)
  - [A short tutorial] (https://reactjs.org/tutorial/tutorial.html)
- [Thinking in React] (https://reactjs.org/docs/thinking-in-react.html)

React Fundamentals: 不錯且都很短的教學影片

- Hello World - First Component
- The Render Method
- Introduction to Properties
- State Basics
- Owner Ownee Relationship
- Using Refs to Access Components
- Accessing Child Properties
- Component Lifecycle - Mounting Basics
- Component Lifecycle - Mounting Usage
- Component Lifecycle - Updating

感謝聆聽！