Week 9: 延續 React

謝心默

B06505017@ntu.edu.tw

作業請交到這邊

- https://classroom.github.com/a/YzD40PO_

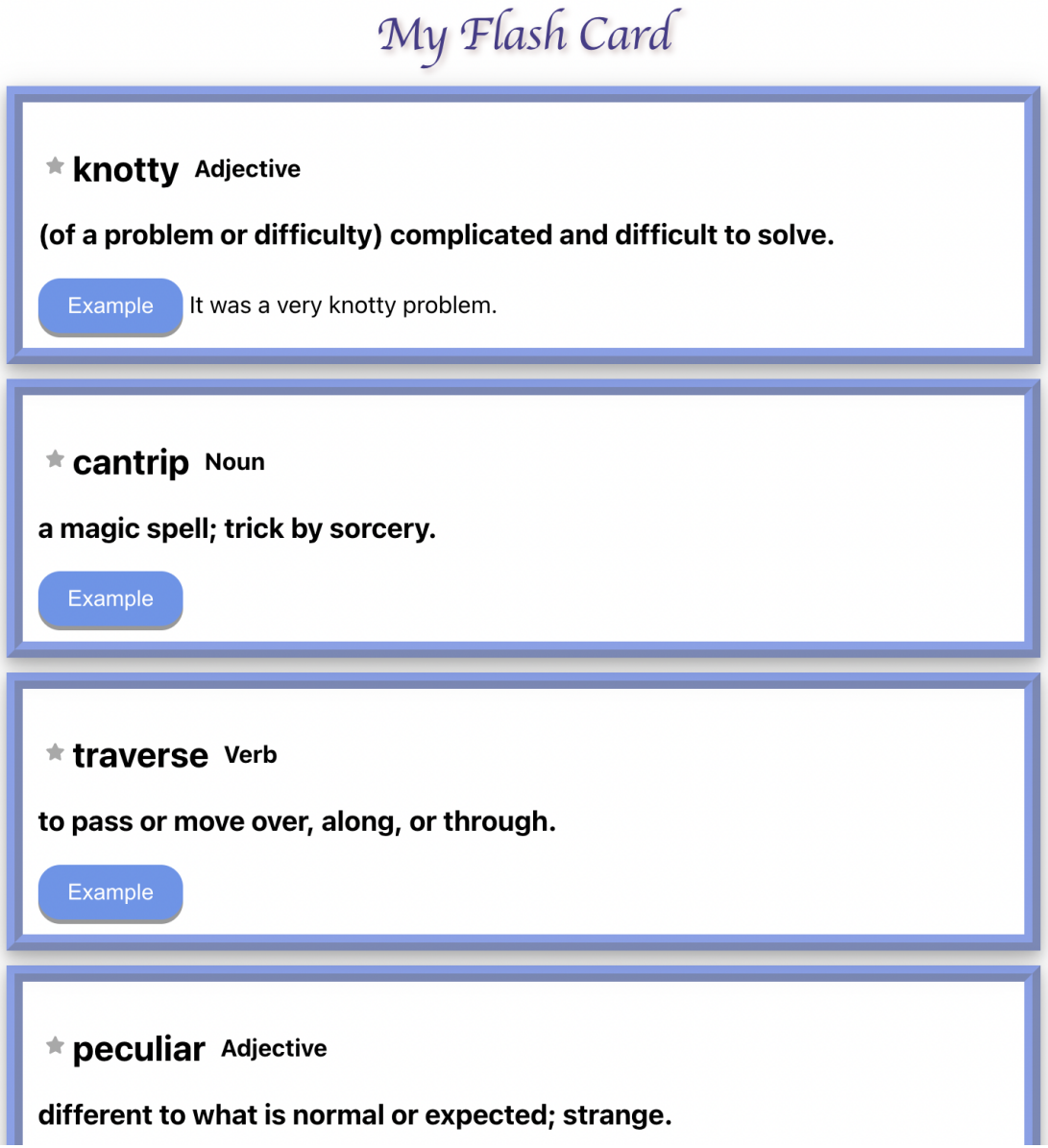
先幫大家再複習一下

- `root.render()` to update DOM
- React element 可以用 function 或 class 宣告
- 使用 `this.props` 以及 tag attribute 來傳遞參數
- `this.props` are read-only

再來講講上次的作業吧

課堂 / 回家作業

- 請用 React 實作一個 flashcard app
- 需要有標題、字卡、單字、詞性、意思
- 點星星會變色
- 點例句按鈕會顯示例句
- 盡量讓它長得好看
- 請先用 js 原生修改 DOM 的方式實作（先不要加入 State）
- 請先不要用外面的套件（Icon 可以）



Steps

- 把 github 上面的 `/lab/flashcard` pull 下來，複製到自己的 repo
- 在自己的 repo 跑一下 yarn

```
$ npm install yarn -g
$ yarn
$ yarn start
```

- 在 `/src` 裡面會有 `containers` 和 `components`，也可以自己新增
- 統一在 `/public/styles.css` 加入 style
  - **也可以直接用我的 `styles.css`**

Layout


<Header />
My Flash Card
<Card />
<Button />
<h1 />
<h4 />
knotty
Adjective
<h3 />
(of a problem or difficulty) complicated and difficult to solve
<Button />
Example
It was a very knotty problem.
<Content />
cantrip Noun
a magic spell; trick by sorcery.
Example
traverse Verb
to pass or move over, along, or through.
Example
peculiar Adjective
different to what is normal or expected; strange.

<FlashCard />

```
import Header from "../components/Header";
import Content from "../components/Content";

const vocabularies = [ /* ...*/ ];

function FlashCard() {
  return (
    <div>
      <Header title={"My Flash Card"} />
      <Content vocabularies={vocabularies} />
    </div>
  );
}

export default FlashCard;
```

<Header />

```
import PropTypes from "prop-types";

export default function Header({ title }) {
  return <header className="title">{title}</header>;
}

Header.propTypes = {
  title: PropTypes.string.isRequired,
};
```

<Content>

- 有很多個單字，總不能放一堆 `<Card />` 吧？！

```
export default function Content({ vocabularies }) {
  return (
    <div className="content">
      <Card
        word={vocabularies[0].word}
        part_of_speech={vocabularies[0].part_of_speech}
        definition={vocabularies[0].definition}
      />
      <Card
        word={vocabularies[1].word}
        part_of_speech={vocabularies[1].part_of_speech}
        definition={vocabularies[1].definition}
      />
      {
        /*
        .
        .
        .
        */
      }
    </div>
  );
}
```

還記得之前學過的 `map()` 嗎？

- 把 array 裡面每個 element 都「映射」成新的 element
- **`map((element) => { /* … */ })`**

```
const numbers = [1, 4, 9];
const doubles = numbers.map((num) => num * 2); // [2, 8, 18]
```

- **`map((element, index) => { /* … */ })`**

```
const strings = ["a", "b", "c"];
const strings_with_index = strings.map((s, i) => `${i}_${s}`); // ["0_a", "1_b", "2_c"]
```

`<Content>` 就可以寫成這樣：

```
export default function Content({ vocabularies }) {
  return (
    <div className="content">
      {vocabularies.map((v, i) => (
        <div className="card" key={i}> // 這個 key 很重要！！
          <Card
            word={v.word}
            part_of_speech={v.part_of_speech}
            definition={v.definition}
          />
        </div>
      ))}
    </div>
  );
}
```

接下來看 `<Card />`

```
export default function Card({
  id,
  handleStarClick,
  word,
  part_of_speech,
  definition,
  handleExampleClick,
}) {
  return (
    <>
      <div className="vocabulary">
        { /* star button here */ }
        <h2 className="word">{word}</h2>
        <h4>{part_of_speech}</h4>
      </div>
      <h3 className="definition">{definition}</h3>
      { /* example button here */ }
    </>
  );
}
```

有兩個會用到 `<Button>` 的地方

1. 按下星星會加入最愛（變色）
2. 按下 Example 會顯示例句

- **會需要長得不太一樣 ➡️ `props` 傳入 `className`**
- **按下去的行為不一樣 ➡️ `props` 傳入 `onClick` function**

但，一個需要文字、另一個需要 Icon 耶？

還記得上週講到的 `props.children` 嗎？

```
export default function Button(props) {
  return (
    <button className={props.className} onClick={props.onClick}>
      {props.text}
      {props.children}
    </button>
  );
}
```

這樣就可以在底下傳入 text or icon

```
<Button
  className={"example-button"}
  text={"Example"}
  onClick={handleExampleClick}
/>
```

```
<Button className={"icon-button"}>
  <FaStar
    icon="fa-star"
    className="fa-star"
    id={`star-${id}`}
    onClick={handleStarClick}
  />
</Button>
```

> `handleStarClick` 和 `handleExampleClick` 要定義在哪裡？

Containers v.s. Coponents

- Containers: 整個 DOM 架構的基底，存著 state/props 以及一些主要的**程式邏輯**
  - 通常是有狀態 (state) 的，因為它們提供資料
- Components: 功能或 style 會重複使用的元件，很少有自己的狀態 (state)， 也比較沒有複雜的邏輯

  > 當有自己的狀態時，通常是 UI 狀態而不是資料結構

放在 `<FlashCard />` 再一層一層往下傳

- `<FlashCard />` ➡️ `<Content />` ➡️ `<Card>` ➡️ `<Button />`

```
function FlashCard() {
  const handleStarClick = (event) => {};
  const handleExampleClick = (event) => {};

  return (
    <div>
      <Header title={"My Flash Card"} />
      <Content
        vocabularies={vocabularies}
        handleStarClick={handleStarClick}
        handleExampleClick={handleExampleClick}
      />
    </div>
  );
}
```

還是不知道怎麼 handle ...

- hint1: `console.log` 一下 `event.currentTarget`
- hint2: how about `event.currentTarget.parentNode`?
- hint3: 想一下怎麼動態 append 元件到某個元件後面？
- hint4: 瘋狂按 example 的話會怎麼樣？

Q: 什麼時候適合拆 component ？

1. 當需要重複利用一個 component 時
2. 當想要把一個獨立的 component 封裝起來，不受其他 components 影響時
3. 當一個 component 太大，造成不相關邏輯強烈耦合，或者難以閱讀時

> 太瑣碎的 components 也可能會造成維護上的不易

終於進入今天的正題！ State！

如果我們想要動態修改 component，讓網頁有更多互動性呢？

- 按下確認按鈕後跳轉頁面
- 記錄貼文按讚數量
- 打開下拉選單

➡️➡️➡️ `state`: 讓 component 擁有自己的狀態

- e.g. 時鐘顯示的時間、 menu 的開合、後端回傳的 API 結果

更精確地來說，何時需要使用 state？

1. component 需要使用某些資料/狀態，且根據這些資料，畫面會顯示不同的結果
2. 資料/狀態不會被上層的 component 所需要
3. 資料/狀態不需要由上層的 component 提供

`state` v.s. `props`

- `state` 是 component 自己創建的資料
  - 擁有 `state` 的 component 會自行管理 / 修改 `state` 的內容
- `props` 是上層 component 傳入的資料
  - 接收 `props` 的 component 不可自行修改其內容

要完整學會 state，我們需要了解三個觀念

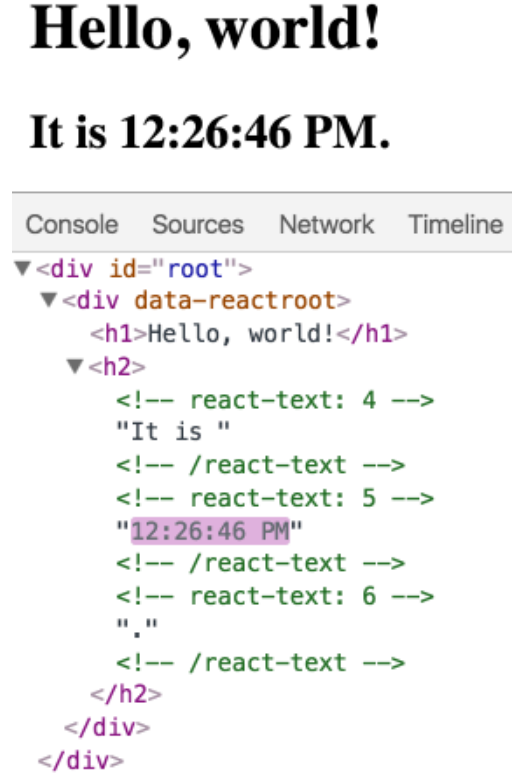
1. `this.state` & `setState()`
2. Component lifecycle
3. Event handling

還記得上禮拜的時鐘嗎？

```
const root = ReactDOM.createRoot(document.getElementById("root"));

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

setInterval(tick, 1000);
```



我們今天來把它變成一個獨立的 component

`<Clock />`

先把 component 的部分拆出來

```
const root = ReactDOM.createRoot(document.getElementById('root'));

function Clock(props) { // <= here
  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  root.render(<Clock date={new Date()} />);
}

setInterval(tick, 1000);
```

`this.state`

- JavaScript object in class component
- 完全由創建的 component 維護 / 控制，不會被上層或任何其他的 components 改動
- 操作 state 只有兩個階段
    i. **初始化** state
    ii. **更新** state

初始化 state

- 時鐘的例子（這裡改成 class component）

```
class Clock extends React.Component {
  constructor(props) {
    super(props);

    this.state = { // object declaration
      date: new Date()
    };
  }
  render() { /** ... **/ }
}
```

用 `this.state.date` 拿到它


```
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> // <-- here
        </div>
      )
    }
  }
}

function tick() {
  root.render(<Clock />); // <--
}

setInterval(tick, 1000);
```

> 但時鐘不會動了？

❌ Note: 請記得 `state` 不能從上層傳入

```
// wrong
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>
     )
  }
}

root.render(<MyClass name="Momo" count={10} / >);
```

> output: `Momo is called 0 times!`

在講如何更新 state 之前，

需要先了解一下 React 的 component lifecycle

Component Lifecycle 生命週期

- 一個 React component 從一開始被定義、被 render 到畫面上、因為內容改變而被 re-render、到最後從 DOM 被移除，總過會經歷 3 個階段的 lifecycle

1. **Mounting**
    - component 即將被產生並 mount 到 DOM 上面
2. **Updating**
    - component 因為 `props` / `state` 的改變而 trigger virtual DOM 去 re-render 它
3. **Unmounting**
    - component 即將從 DOM 被移除

Mounting 依序會呼叫到的 methods

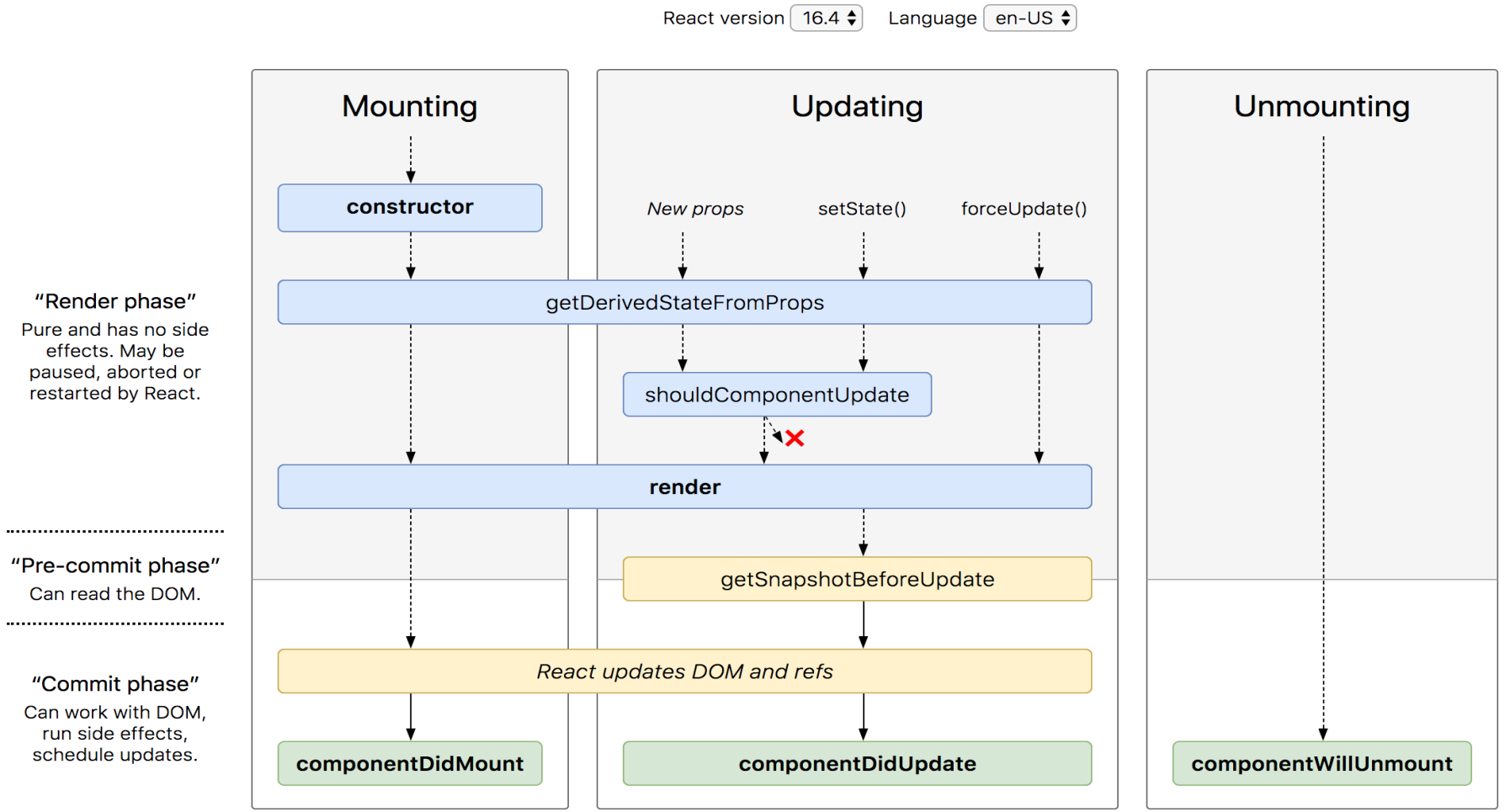
- `constructor()`
- `static getDerivedStateFromProps()`
  - 用初始接收到的props」去設定第一次render時的state
- `render()`
- `componentDidMount()`

Updating 依序會呼叫到的 methods

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

Unmounting 會呼叫到的 method

- `componentWillUnmount()`


React version 16.4
Language en-US
Mounting
Updating
Unmounting
constructor
New props
setState()
forceUpdate()
getDerivedStateFromProps
shouldComponentUpdate
render
getSnapshotBeforeUpdate
React updates DOM and refs
componentDidMount
componentDidUpdate
componentWillUnmount
"Render phase"
Pure and has no side effects. May be paused, aborted or restarted by React.
"Pre-commit phase"
Can read the DOM.
"Commit phase"
Can work with DOM, run side effects, schedule updates.

詳細可參考官網

知道這個可以幹嘛？？

- 用 *Mounting* 階段中的 methods 初始化 `props` / `state` 的值 or 做一些必要的設定
  - E.g. 在 `componentDidMount()` 設定時鐘的 timer `setInterval()`
  - E.g. 在 `componentDidMount()` 打 API
- 用 event handling 的方式更新 state 的值，並且在 *updating* 階段中 re-render component
  - E.g. 如果要打 API 執行，component 並不會等 response 回來才 render => 在讀取完資料前會轉圈圈/顯示 loading
- （某些情況下）當 component 要被從 DOM 移除時，須在 *unmounting* 階段把資源還給系統

更新 state: `setState()`

- 用 `setState()` 才會去通知 virtual DOM 重新呼叫 `render()` 來更新畫面
  - `this.state.date = new Date();` => 不會 re-render

> 自己不需要重新呼叫 `root.render()`

套用到我們的時鐘上

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

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

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

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

關於 `setState()`，你需要記住的幾件事

1. 不能直接更改 state 來更新

```
// ❌
this.state.comment = "Hello";

// ✅
this.setState({ comment: "Hello" });
```

2. state 的更新可能是 Asynchronous 的

- React 可能會把一段時間內的多個 `setState()` 批次（Batch）執行

```
// ❌✅
this.setState({
  counter: this.state.counter + this.props.increment
});

// ✅
this.setState((state, props) => ({ // function declaration
  counter: state.counter + props.increment
}));
```

3. 新的 state 會與舊的 state 合併

- 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
      });
}); }
```

4. 單向資料流（Unidirectional data flow）

- 一個 component 也無法知道另一個 component 是有狀態的（**Stateful**）還是無狀態的（**Stateless**）

  > 對於 React 來說，component 是否有 state 只是實作細節而已

- 這代表一個 stateless 的 component 中可以有一個 stateful 的 component，反之亦然

```
// inside <MyComponent>
<FormattedDate date={this.state.date} />
<FormattedDate date={new Date()} />

function FormattedDate(props) { // 這個 component 不會知道 props 是從哪來的
  return <h2>It is {props.date.toLocaleTimeString()}.</h2>;
}
```

> 所以，我們通常會把 state 往上提，如果 child component 會 depend on parent component state 的值，則在 child component 宣告的地方把 parent 的 state 放在 props 傳給 child

Umounting

`<Clock />` component 的更新是被 system clock tick 驅動的，所以如果它因故從 DOM 被移除，即使畫面上不再顯示這個 component，它的 `tick()` 還是會被一直呼叫

➡️➡️ 宣告一個變數把 `setInterval()` 回傳的 timer ID 存下來，然後在 *Umounting* phase 把它還給系統

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

componentWillUnmount() {
  clearInterval(this.timerID);
}
```

最後，我們的 <Clock /> 會長這樣

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

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

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

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

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Event handling

在一般的網頁中，常常是由 I/O events 來觸發畫面的更新

- React 的 event handling 基本上跟 JavaScript 差不多

複習一下 JavaScript 的 event handling

1. `addEventListener()`

```
const target = document.getElementById("target");
target.addEventListener("click", function() {/* ... */});
```

2. GlobalEventHandlers

```
let target = document.getElementById("target");
target.onclick = inputChange;

function inputChange(e) {/* ... */}
```

3. As a tag attribute

```
<div class="myClass" onclick="clickHandler()">

function clickHandler() {/* ... */}
```

Event handling in React

1. JSX 裏頭 tag 的名稱為了不要跟 JavaScript 的保留字衝突，會變成 **camelCase**，且可能會換成別的名稱
2. JavaScript 傳字串，React 傳 function

```
// HTML
<button onclick="activateLasers()">
  Activate Lasers
</button>

// React
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

有三種寫法

1. 在 `constructor` 中用 `bind` 包裹 event handler

```
class MyButton extends React.Component {
  constructor(props) {
    super(props);

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    console.log('this is:', this);
    // output: {props: {…}, context: {…}, refs: {…}, updater: {…}, state: {…}, …}
  }

  render() {
    return (
      // This syntax ensures `this` is bound within handleClick
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

2. 使用 public class fields 語法 (for CRA, need Babel)

```
class MyButton extends React.Component {
  handleClick = () => { // 一定要用 error function!!
    console.log('this is:', this);
  }

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

> arrow function 裡頭的 this refers to the caller's scope

3. 在 event expression 中用 arrow function / bind 包裹 event handler (較不推薦)

```
class MyButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }
  render() {
    return (
      // This syntax ensures `this` is bound within handleClick
      <button onClick={() => this.handleClick()}>
        Click me
      </button>
    );
  }
}
```

> 每次 render 時都會額外產生一次新的 `() => this.handleClick()` function，可能產生額外 re-render 的問題

另外，其實也可以直接這樣寫（不推薦）

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

> 但超過一行可讀性就會變差，而且還是會有前面說到 re-render 的問題

前面的語法是不是已經讓大家有點頭昏眼花了...

（有沒有覺得 class component 寫起來好麻煩 > <

沒關係，React 聽到了！

Hook

- React 16.8 中增加的新功能
- 原本只能用 `class` 實作 stateful component

➡️➡️➡️ 現在不必寫 class 就能使用 state 以及其他 React 的功能！

> 讓你可以從 function component 「**hook into**」 React state 與 Lifecycle

useState()

- `const [當前的值, 更新值的function] = useState(初始值);`
  - 更新值的 function 通常命名為 `setXxxx`

```
import { useState } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

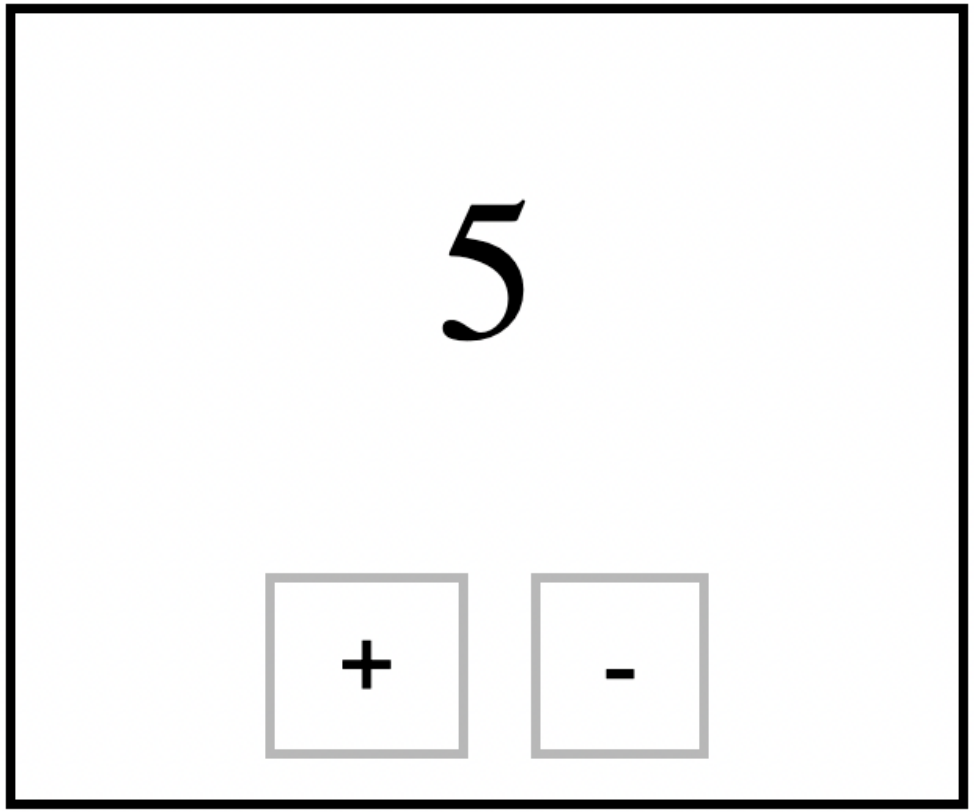
> **基本上可以不用管 this!!**

一個 function component 可以有多個 `useState()`

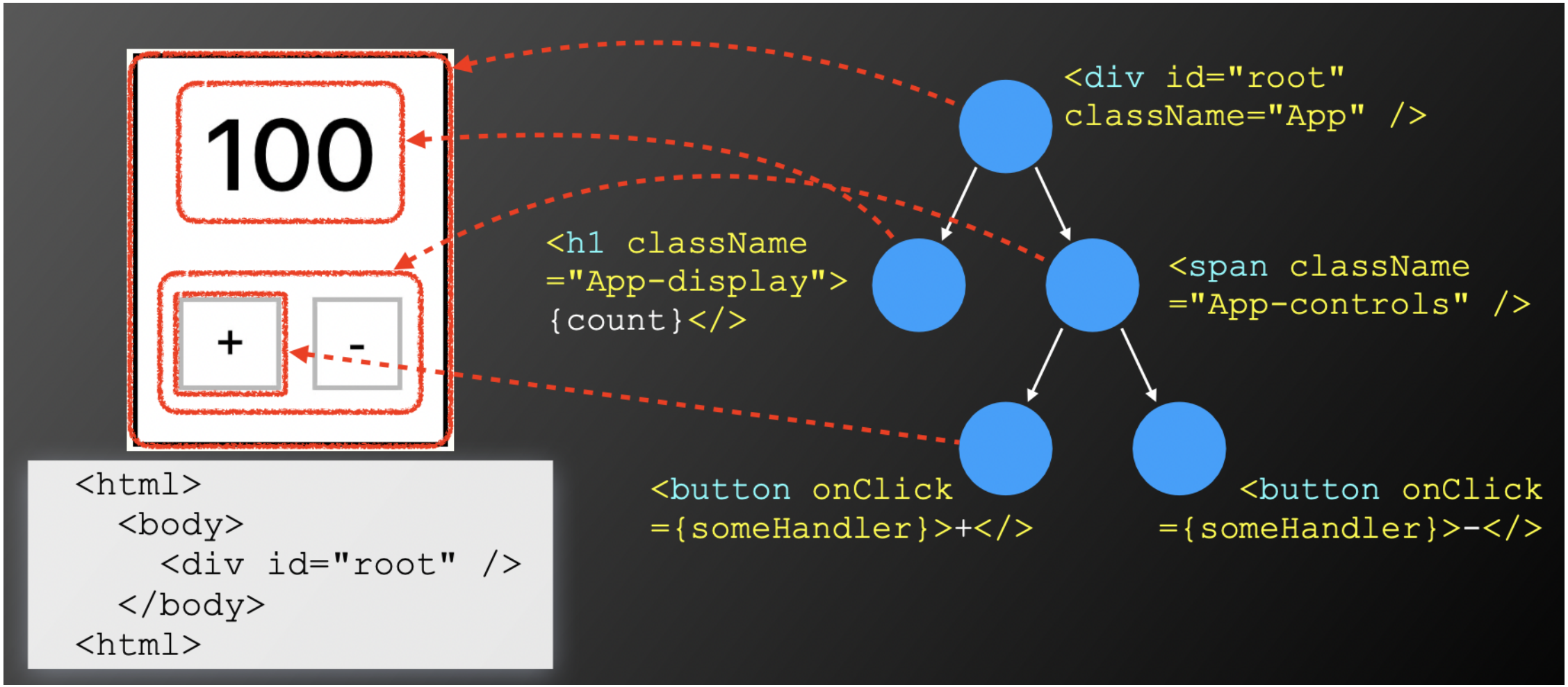
```
function ExampleWithManyStates() {
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
  // ...
}
```

來練習一下吧

- 用 React 實作一個可以加/減的計數器



第一步：畫出 DOM Structure



第二步：在 `src/Counter.js` 定義一個 top level component `<Counter />`，並且用它來產生 DOM

> 先定義靜態畫面就好

```
function Counter() {
  return (
    <div className="counter">
      <h1 className="counter-display">5</h1>
      <span className="counter-controls">
        <button>+</button>
        <button>-</button>
      </span>
    </div>
  );
}

export default Counter;
```

第三步：在 `src/index.js` 把 `<Counter />` 和 `public/index.html` 串起來

```
import Counter from "./containers/Counter";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <React.StrictMode>
    <Counter />
  </React.StrictMode>
);
```

> 這時候可以先 `$ yarn start` 看看

加入 state

```
import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(5);

  return (
    <div className="counter">
      <h1 className="counter-display">{count}</h1>
      <span className="counter-controls">
        <button>+</button>
        <button>-</button>
      </span>
    </div>
  );
}
```

加入 handler function

```
function Counter() {
  const [count, setCount] = useState(5);

  const handleInc = () => {
    setCount(count + 1);
  };

  const handleDec = () => {
    setCount(count - 1);
  };

  return (
    <div className="counter">
      <h1 className="counter-display">{count}</h1>
      <span className="counter-controls">
        <button onClick={handleInc}>+</button>
        <button onClick={handleDec}>-</button>
      </span>
    </div>
  );
}
```

拆分 Component

- 把 `<button>` 拆出來

```
// Button.js
export default function Button({ onClick, text }) {
  return <button onClick={onClick}>{text}</button>;
}
```

```
// Counter.js
// ...
  <span className="counter-controls">
    <Button text="+" onClick={handleInc} />
    <Button text="-" onClick={handleDec} />
  </span>
```

再加一點點東西

- 加入一個 reset 的 button
- 加入一個 +2 的 button，並在 handler 裡使用兩次 `setCount()`

> 有沒有發現哪裡怪怪的？

直覺上來說可能會寫成這樣

```
// ❌
const handleInc = () => {
  setCount(count + 1);
  setCount(count + 1);
};
```

但是，請記得 state update 是 asynchronous 的！

```
// ✅
const handleInc = () => {
  setCount((prev_count) => prev_count + 1);
  setCount((count) => count + 1); // 這兩行意思是一樣的
};
```

`useEffect()` hook

- React 會在每一次 render 完去跑這個 function
- 一次搞定 `componentDidMount`, `componentDidUpdate`, & `componentWillUnmount`

`useEffect()` hook

- no dependency value

```
useEffect(() => {
  // Runs on every render
});
```

- empty array

```
useEffect(() => {
  // Runs only on the first render
}, []);
```

- props or state

```
useEffect(() => {
  // Runs on the first render
  // and any time any dependency value changes
}, [prop, state]);
```

原本的寫法: `this.state` + lifecycle

```
class Example extends React.Component {
  constructor(props) { /* ... */ }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }

  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={(state) => this.setState({ count: state.count + 1 })}>
          Click me
        </button>
      </div>
    )
  }
```

`useState` + `useEffect`

```
import { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

> 是不是簡單多了（汗）

在 `componentWillUnmount()` 階段清除資源

- E.g. 登入後 subscribe to some service，登出後把 subscription 相關的資源清除掉
  - 使用 lifecycle function

```
componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(this.props.friend.id, this.handleStatusChange);
}
```

  - 使用 `useEffect()`

```
useEffect(() => {
  const handleStatusChange = (status) => setIsOnline(status.isOnline);
  ChatAPI.subscribeToFriendStatus(props.friend.id,handleStatusChange);

  // 指定如何在這個 effect 之後執行清除
  return function cleanup() {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  }
});
```

將不同 states 分成不同的 hooks

- 讓同個 state 的邏輯放在一起，而不是被 lifecycle functions 拆開

```
function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    const handleStatusChange = (status) => setIsOnline(status.isOnline);
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
  });
}
```

幾個坑

1. 為了避免錯誤和影響程式易讀性，`useEffect()` 中，有用到會改變的 React 變數都要放進 dependency array
2. 盡量避免把會在 `useEffect()` 用到的函式定義在 function component 內

> class component 在更新元件時，只會去呼叫對應的生命週期函式。但是 function component 在更新元件時是重新呼叫整個 function component 函式定義域。

```
function Counter() {
  const [cnt, setCnt] = useState(0);
  console.log("hello world!"); // 每次按下按鈕都會有 log
  return (
    <button onClick={() => setCnt(cnt+1)}>+</button>
  );
}
```

情境一: 該函式和任何 state /props 無關

```
function callHelloWorld(){
    console.log("hello world!");
}
function Counter() {
  const [cnt, setCnt] = useState(0);

  useEffect(() => {
    callHelloWorld();
  }, []);

  return (
    <button onClick={() => setCnt(cnt+1)}>+</button>
  );
}
```

情境二: 該函式只和某個特定的state、props改變有關

- 在 `useEffect()` 裡處理，不用另開 function

```
function Counter(props) {
  const [cnt, setCnt] = useState(100);

  useEffect(() => {
    if (props.flag === "reset") {
      console.log("hello world");
    }
  }, [props]);

  return (
    <button onClick={() => setCnt(cnt+1)}>+</button>
  );
}
```

情境三: 該函式和多個state、props有關

- 使用 React 提供的原生 hook `useCallback()`

Hook 的規則

- **只在最上層呼叫 Hook**
  - 不要在迴圈、條件式或是巢狀的 function 內呼叫 Hook，確保當每次 component 被 render 時 Hooks 都依照正確的順序被呼叫
- **只在 React Function 中呼叫 Hook**
  - 不要在一般的 JavaScript function 中呼叫 Hook

Custom Hook

- 創造自己的 React hook
- 可以在 custom hook 中創造 state、生命週期等等
- custom hook 中的 state 和生命週期是獨立運作的
- 非常好用！有興趣可以參考官網

作業（二選一）

- 把上週的 flashcard app 改成 state 的方式，並加入「查看我的最愛」功能
- 實作一個計算機，至少要有加減乘除、AC 的功能

References

- Ric's Web Programming Class Slides
- https://reactjs.org/docs/getting-started.html
- https://ithelp.ithome.com.tw/users/20107790/ironman/3338
- https://ithelp.ithome.com.tw/users/20116826/ironman/2278
- https://www.w3schools.com/react/react_hooks.asp