Keeping Components Pure

https://beta.reactjs.org/learn/keeping-components-pure

Pure functions in JS only perform a calculation and nothing more. You can apply this same thinking to React – each component should be pure and do one thing.

This is why we looked at changing things immutably last week.

A pure function has 2 main characteristics:

- Minds its own business - It does not change any objects or variables that existed before it was called
- Same inputs, same output - Given the same inputs, a pure function should always return the same result (no matter how many times it is called)

Pure functions are like formulas in Maths:

If x = 2 then y = 4. Always.
If x = 3 then y = 6. Always.
If x = 3, y won't sometimes be 9 or –1 or 2.5 depending on the time of day or the state of the stock market.

If we made this in to a JS function, it would look like:

```
function double(number) {
  return 2 * number;
}
```

This is pure! It would be impure if instead of returning 2 * number, it did this:

```
function double(number) {
  number = 2 * number;
}
```

This is changing the original variable! That's allowed in JS functions (if necessary), but NOT allowed in React. Particularly when Strict Mode is running (normally while developing) as it renders each item twice to check for purity! (Checks result of render 1 against render 2).

Local Mutation:

However, **local mutation** is allowed. Pure functions don't mutate variables outside of the function's scope or objects that were created before the call—that makes them impure! However, it's completely fine to change variables and objects that you've just created while rendering. In this example, you create an [] array, assign it to a cups variable, and then push a dozen cups into it:

```
function Cup({ guest }) {
  return <h2>Tea cup for guest #{guest}</h2>;
}

export default function TeaGathering() {
  let cups = [];
  for (let i = 1; i <= 12; i++) {
    cups.push(<Cup key={i} guest={i} />);
  }
  return cups;
}
```

If the cups variable or the [] array were created outside the TeaGathering function, this would be a huge problem! You would be changing a pre-existing object by pushing items into that array.
However, it's fine because you've created them during the same render, inside TeaGathering. No code outside of TeaGathering will ever know that this happened. This is called "local mutation"—it's like your component's little secret.

Side Effects

While functional programming relies heavily on purity, at some point, somewhere, *something* has to change. That's kind of the point of programming! These changes—updating the screen, starting an animation, changing the data—are called **side effects**. They're things that happen *"on the side,"* not during rendering.
In React, side effects usually belong inside event handlers**.** Event handlers are functions that React runs when you perform some action—for example, when you click a button. Even though event handlers are defined *inside* your component, they don't run *during* rendering! **So event handlers don't need to be pure.**
If you've exhausted all other options and can't find the right event handler for your side effect, you can still attach it to your returned JSX with a useEffect call in your component. This tells React to execute it later, after rendering, when side effects are allowed. **However, this approach should be your last resort.**
When possible, try to express your logic with rendering alone. You'll be surprised how far this can take you!

Props and State

In React there are three kinds of inputs that you can read while rendering: props, state, and context. You should always treat these inputs as read-only.
When you want to change something in response to user input, you should set state instead of writing to a variable. You should never change pre-existing variables or objects while your component is rendering.

Why is purity important?

Writing pure functions takes some habit and discipline, but it also unlocks marvellous opportunities:

- Your components could run in a different environment—for example, on the server! Since they return the same result for the same inputs, one component can serve many user requests
- You can improve performance by skipping rendering components whose inputs have not changed. This is safe because pure functions always return the same results, so they (the results) are safe to cache. This is why React is very quick for displaying apps/web pages – it only renders components where something has changed
- If some data changes in the middle of rendering a deep component tree, React can restart rendering without wasting time to finish the outdated render. Purity makes it safe to stop calculating at any time

Every new React feature we're building takes advantage of purity. From data fetching to animations to performance, keeping components pure unlocks the power of the React paradigm.

Recap:

- A component must be pure, meaning:
    - **Mind its own business -** It should not change any objects or variables that existed before rendering.
    - **Same inputs, same output -** Given the same inputs, a component should always return the same JSX.
- Rendering can happen at any time, so components should not depend on each other's' rendering sequence.
- You should not mutate any of the inputs that your components use for rendering. That includes props, state, and context (we haven't done context yet). To update the screen, "set" state instead of mutating preexisting objects.
- Strive to express your component's logic in the JSX you return. When you need to "change things," you'll usually want to do it in an event handler. As a last resort, you can useEffect.
- Writing pure functions takes a bit of practice, but they are what makes React so powerful

Chris' added Gems:

Pure functions are very easily testable – so React apps tend to be easily testable too.

If a component needs to "know" something, it should always come in as a prop. That prop can then just be handed back when returning – it should not be modified.

Code for challenges at bottom of page:

Exercise 1:

Original code is trying to use a document selector and modify that mutably, before the return statement:

Clock.js

```
1  export default function Clock({ time }) {
2    let hours = time.getHours();
3    if (hours >= 0 && hours <= 6) {
4      document.getElementById('time').className = 'night';
5    } else {
6      document.getElementById('time').className = 'day';
7    }
8    return (
9      <h1 id="time">
10       {time.toLocaleTimeString()}
11     </h1>
12   );
13 }
```

We need to define a new variable called className inside the function, set that depending on the outcome of our "if" check (local mutation), then pass that in when rendering the h1 using the className attribute.

Clock.js

```
1  export default function Clock({ time }) {
2    let hours = time.getHours();
3    let className;
4    if (hours >= 0 && hours <= 6) {
5      className = 'night';
6    } else {
7     className = 'day';
8    }
9    return (
10     <h1 className={className} id="time">
11       {time.toLocaleTimeString()}
12     </h1>
13   );
14 }
```

14:26:11

Or as a one-liner using an in-line conditional (ternary) operator(basically an "if" check:

Clock.js

```
1 port default function Clock({ time }) {
2 let hours = time.getHours();
3 return (
4   <h1 className={hours >= 0 && hours <= 6 ? "night" : "day"} id
5     {time.toLocaleTimeString()}
6   </h1>
7 );
```

14:26:48

Exercise 2:

In the profile.js file – it sets a variable of currentPerson on line 4 then changes that mutably inside the Profile function on line 7. So the second person is always overwriting the first one in the currentPerson variable:

Profile.js App.js

```
1  import Panel from './Panel.js';
2  import { getImageUrl } from './utils.js';
3
4  let currentPerson;
5
6  export default function Profile({ person }) {
7    currentPerson = person;
8    return (
9      <Panel>
10       <Header />
11       <Avatar />
12     </Panel>
13   )
14 }
15
16 function Header() {
17   return <h1>{currentPerson.name}</h1>;
18 }
19
20 function Avatar() {
21   return (
22     <img
23       className="avatar"
24       src={getImageUrl(currentPerson)}
25       alt={currentPerson.name}
26       width={50}
27       height={50}
```

Need to change this so the “person” prop on line 6 is being handed to the relevant bits that create the header and picture:

Profile.js App.js

```
1  import Panel from './Panel.js';
2  import { getImageUrl } from './utils.js';
3
4  export default function Profile({ person }) {
5    return (
6      <Panel>
7        <Header person={person} />
8        <Avatar person={person} />
9      </Panel>
10   )
11 }
12
13 function Header({ person }) {
14   return <h1>{person.name}</h1>;
15 }
16
17 function Avatar({ person }) {
18   return (
19     <img
20       className="avatar"
21       src={getImageUrl(person)}
22       alt={person.name}
23       width={50}
24       height={50}
25     />
26   );
27 }
```

Exercise 3:

Stories array is being updated mutably then mapped through, so each render is creating multiple "createStories" objects:

Composition is breaking down a problem

StoryTray.js

```
export default function StoryTray({ stories }) {
  stories.push({
    id: 'create',
    label: 'Create Story'
  });

  return (
    <ul>
      {stories.map(story => (
        <li key={story.id}>
          {story.label}
        </li>
      ))}
    </ul>
  );
}
```

So changed the code to spread the stories array and add createStory once, storing inside a new variable. Then calling the map method on the updated array:

StoryTray.js

```
export default function StoryTray({ stories }) {
  const updatedStories = [...stories, {
    id: 'create',
    label: 'Create Story'
  }];

  return (
    <ul>
      {updatedStories.map(story => (
        <li key={story.id}>
          {story.label}
        </li>
      ))}
    </ul>
  );
}
```

Or add the li of Create Story Now inside the return statement:

StoryTray.js

```
export default function StoryTray({ stories }) {
  return (
    <ul>
      {stories.map(story => (
        <li key={story.id}>
          {story.label}
        </li>
      ))}
      <li>Create Story Now</li>
    </ul>
  );
}

```

useEffect

This is NOT the main takeaway from today and is only to be used as a last resort when event handlers don't do the job. Always try to put it in an eventHandler first. Chris might write 2 useEffects per project vs. 100s of eventHandlers.

New beta docs don't exist for this as of today, so used the old docs to learn about it: https://reactjs.org/docs/hooks-effect.html

Also well explained here: https://www.w3schools.com/react/react_useeffect.asp

The below code updates the document.title (what's displayed on the tab) and useEffect is being called when the page loads and then each time the component is re-rendered:

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

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

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

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

Chris thinks of useEffect like this: Its job is to keep things in sync. So, any changes that are made to the component, the useEffect will re-run.

To prove that useEffect is running every time the component re-renders, Chris added a way to change the heading, which shouldn't have interfered with useEffect as we would expect useEffect should only be dealing with the count:

```
export default function App() {
  const [count, setCount] = useState(0);
  const [heading, setHeading] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    console.log("This is running", count);
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

  console.log("This is rendering...", count);

  return (
    <div>
      <h1>{heading}</h1>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
      <button onClick={() => setHeading("Hello There")}>Title Change</button>
    </div>
  );
}
```

Important:

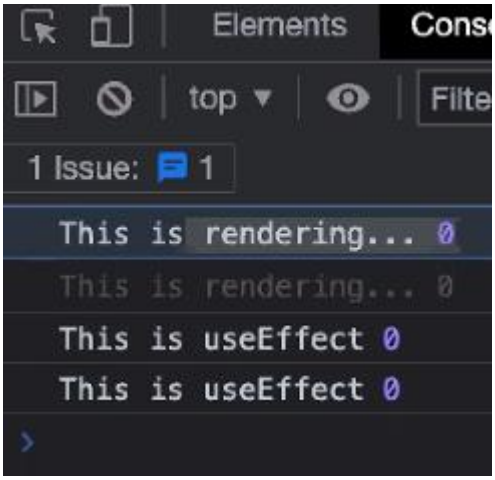
useEffect can often cause infinite renders, so you may need to pass in a dependency array as a second argument (after function declaration) to say which state change(s) (or props) tell useEffect to run. This array should be a list of states to keep in sync with. If ANY (not ALL) of these things change, useEffect will run. This is on line 12:

```
JS App.js M ×

useeffect-demo > src > JS App.js > App
 1  import React, { useState, useEffect } from "react";  6.9k (gzipped: 2.7k)
 2  import "./App.css";
 3
 4  export default function App() {
 5    const [count, setCount] = useState(0);
 6
 7    // Similar to componentDidMount and componentDidUpdate:
 8    useEffect(() => {
 9      console.log("This is running", count);
10      // Update the document title using the browser API
11      document.title = `You clicked ${count} times`;
12    }, [count]);      You, 1 second ago • Uncommitted changes
13
14    console.log("This is rendering...", count);
15
16    return (
17      <div>
18        <p>You clicked {count} times</p>
19        <button onClick={() => setCount(count + 1)}>Click me</button>
20      </div>
21    );
22  }
```

N.B. If you pass in an empty array, you're telling useEffect to stay in sync with nothing – so it will only run once! An example of when this might be useful if you want to only run code once – like a GET request for all recipes from an API (only need to do it once). If you want it to run when there is **any** change – don't pass in an empty dependency array.

useEffect always runs after the render:



Why bother with useEffect?

We could have put the document.title inside the function that's called when the button is clicked, but we used the useEffect hook because it is a **Side Effect** – i.e. it happens after the Render. In our example, we are reaching out to change something outside of the component (document.title), so when we do that, we should do it as a side effect, after the render has taken place.