WHY JSX?

Both server-side and client-side, you SHOULD:

- functions that take state data and return HTML

Functions that modify the DOM can be broken down:

- Convert state data to HTML
- Update page with HTML

JSX

- is this kind of function
- is written like HTML
- is still actually a function (transpiled)

EXAMINE APP.JSX

Let's open up `src/App.jsx` (was `src/App.js`)

JSX

Basic Rules of JSX Syntax

- All JSX tags can self-close (and must close)
- JSX tags require `className` instead of `class`
- JSX tags can take an object (not string) for `style`
    - If you use `style` - we won't
- Anything inside `{}` replaced with evaluated results
    - Notice how this is NOT `${}`!
- Whitespace trims as much as possible
    - Not just to 1 space!

JSX EXAMPLE

JSX (`coolCat` is a JS variable that holds 'Maru')

```
<div className="demo">
  <span>{ 1 + 1 }</span>
  { coolCat }
</div>
```

Actual output:

```
<div class="demo"><span>2</span>Maru</div>
```

JSX VS COMPONENT

Component is the function/object that returns output

JSX is a syntax for the output

- Components return JSX
- Components are NOT JSX

COMPOSITION

New Components are easily created:

- JSX often put inside `()` for clarity (not required)
- JSX is NOT a string - transpiles into a function
    - `{}` not `${}` in JSX
        - `${}` **can** be in text in JSX
- Components have **one** top container element
- Components are MixedCase, not camelCase
- Components can contain (call) other components

```
const MyComponent = () => (<div className="demo">Hi</div>);
const OtherComp = () =>
  (<div> Check out my greeting: <MyComponent/> </div>);
```

```
<div>Check out my greeting: <div class="demo">Hi</div></div>
```

COMPONENT FILES

JSX files can be `.js` or `.jsx`

- I **require** `.jsx` because it's valuable information

You CAN have multiple components per file

- A component is just a function
    - export it like any other function

BUT the convention is to have one component per file

- I **require** exactly one component per file
    - because we are learning
- Name the file after the component
    - MixedCase, not camelCase

COMPOSITION USING MANY COMPONENT FILES

```
import Cat from './Cat';
import Box from './Box';

const Room = () => {
  return (
    <div className="room">
      <Box><Cat/></Box>
      <Box><Cat/></Box>
      <Box/>
    </div>
  );
};
export default Room;
```

COMPONENTS: CLASSES VS FUNCTION

React Components can be defined as classes:

```
class MyComponent extends React.Component {
  render() {
    return ( <div>
      // ...
```

or as functions:

```
function MyComponent()
  return ( <div>
    // ...
```

Originally some actions required class-based components

In Feb 2019, they released "hooks": classes are no longer required.

WHICH DO WE DO, CLASSES OR HOOKS?

This is a hard decision:

- No time to do both in depth
- Much existing uses class-based components
- Web world changes rapidly for new development
- ...but employers change dependencies slower

A project can use both!

- I teach **function-based** because strong shift to it

PROPS

Like HTML, React Components can be passed attributes, called "props"

The component gets them as arguments:

```
<MyComp name="Bao"/>
```

```
function MyComp(props) {
  return (<div>{props.name}</div>);
}
```

```
<div>Bao</div>
```

You can destructure like any object/function call:

```
function MyComp({ name }) {
  return (<div>{ name }</div>);
}
```

ABOUT PROPS

In HTML

- attributes must be strings
- properties have no value

In JSX, props can be ANY DATA (if in {})

```
<MyComp info={ [ 1, 2, 3 ] }/>
```

In JSX, properties should be set as boolean

```
<MyComp disabled={true}/>
```

JSX is often passed callback functions as props!

```
<MyComp onLogout={logoutCallback}/>
```

CHILDREN (TAG CONTENTS)

To access JSX contents, use special prop "children":

```
<MyComp>This is some content</MyComp>
```

```
const MyComp = ({ children }) => {
  return (
    <div>I heard: <b>{children}</b></div>
  );
};
```

```
<div>I heard: <b>This is some content</b></div>
```

```
import Box from './Box';
import Cat from './Cat';
const Room = () => {
  return (
    <div className="room">
      <Box><Cat name="Maru"/></Box>
      <Box><Cat name="Grumpy"/></Box>
      <Box/>
    </div>
  );
};
export default Room;
```

```
const Box = ({ children }) => {
  const contents = children ? children : <div>Nothing</div>;
  return ( <div> A box contains: {contents} </div>);
};
export default Box;
```

```
const Cat = ({ name }) => (<div>{name}</div>);
export default Cat;
```

WHEN TO USE PROPS

Props are essential to using components

Think of them as arguments to a function call

- Because they are!
- Makes components more flexible and reusable
- Keeps components "dumb" and unaware of app state

We make components reusable and decoupled

- Just like we do functions, like service calls

WHEN NOT TO USE JSX

If your logic doesn't directly generate HTML/JSX

- Consider making it a plain JS function!
- Put in component -OR- import from `.js` file
- Keeps components simple
- Keeps JS testable, reusable, findable

COMPONENT EVENTS

Event Handlers added to *HTML* elements in JSX:

```
const wasClicked = () => {
  console.log('A furry paw lashes out at you');
};

const Box = () => (
  <div onClick={wasClicked}>A Box</div>
);

export default Box;
```

Despite appearance this is **not** inline JS in HTML

- JSX converts this to an addEventListener()
- This is all defined in JS or JSX, not HTML
- Notice handler is just JS, not JSX

COMPONENT STATE

Each component can have its own state

- class-based did so as a state object
- function-based use "hooks" - special closures

Either way, be careful in managing your state

- If the state doesn't belong to the component, it should be passed in as a prop
- Complex state is a source of bugs

COMPONENT STATE DEMO

```
import Box from './Box';

const App = () => ( <Box/> );
export default App;
```

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

const Box = () => {
  const [isOpen, setIsOpen] =  useState(false);
  return (
    <button
      onClick={ () => setIsOpen( !isOpen ) }
    >
      The Box is {isOpen ? 'Open' : 'Closed'}
    </button>
  );
};
export default Box;
```

COMPONENT STATE DETAILS

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

`useState` is a *named import* from the react library

```
const [isOpen, setIsOpen] =  useState(false);
```

- useState is a function
- passed the default value (if any)
    - We are deconstructing to two variables
    - default only applies if a value was never set
- returns an array of two values
    - latest value for this state
    - a function to change the value

`useState()` runs on EACH render

USESTATE EXAMPLES

```
const Counter = function() {
  const [count, setCount] = useState(1);
  return (
    <div onClick={ () => setCount(count+1) }>
      {count}
    </div>
  );
};
```

MORE USESTATE EXAMPLES

```
const Compare = () => {
  const [count1, setCount1] = useState(1);
  const [count2, setCount2] = useState(1);
  const bumpOne = () => setCount1( count1+1 );
  const bumpTwo = () => setCount2( count2+1 );

  let comparison = 'is equal to';
  if(count1 > count2) {
    comparison = 'is greater than';
  } else if (count1 < count2) {
    comparison = 'is less than';
  }

  return (
    <div>
      <button onClick={ bumpOne }>{count1}</button>
      {comparison}
      <button onClick={ bumpTwo }>{count2}</button>
    </div>
  );
};
```

MORE USESTATE EXAMPLES

```
const List = () => {
  const [list, setList] = useState([]);
  const addToList = () => {
    const random = Math.floor(Math.random()*10);
    setList( [...list, random] );
  };
  return (
    <div>
      <p>The list is: { list.join(',') }</p>
      <button onClick={ addToList }>Add Another!</button>
    </div>
  );
};
```

PURE COMPONENTS

"Pure Functions" are functions that are not modified by, and do not modify, an outside state

- Return a value based only on the data passed in

"Pure Components" are the same:

- Return a value based only on the data passed in

```
const MyComp = ({ label, action }) => {
  return (<button onClick={action}>{label}</button>);
};
```

WHY WAS THAT GOOD?

Inline JS is bad, why is this good?

```
const MyComp = ({ label, action }) => {
  return (<button onClick={action}>{label}</button>);
};
```

- This is transpiled JSX
    - the output is NOT html with inline js

What's the value?

- Same as functions - this encapsulates responsibilities
- Change in one place

COMMON EARLY JSX MISTAKES

- Not using MixedCase for components
- Being too specific
    - components should be reusable
    - components should not "know" the outside
- Putting too much in one component
    - Like functions, break it down
    - one function, one purpose
    - one component can call others
- Expecting props to auto mean the same as HTML
- Putting too much logic in JSX
    - You should put in raw JS and import

APPLICATION STATE

In "vanilla" JS, app state is JS variables in memory

- Same in React
- Top-level component passes down to children
- Child components can pass to deeper children

If too much state is passing too deep, you want application state management

- some basic using React Context
- or use an outside lib (Redux, etc)
- complex state outside React-as-view

APPLICATION STATE DEMO

```
import { useState } from 'react';
import Counter from './Counter';
import TopN from '/.TopN';
import listOfStuff from './somelist'; // Not component

const App = () => {
  const [count, setCount] = useState(1);
  const countUp = () => setCount(count+1);
  return ( <div>
    <Counter count={count} onCount={countUp}/>
    <TopN showTop={count} list={listOfStuff}/>
  </div>);
};
export default App;
```

- Counter and TopN know **little** about each other
- Or even the context they are called in
- This is good practice - function or component

RENDERING AND THE VIRTUAL DOM

A defined component is an uncalled function.

Converting a component to HTML is "rendering"

A component can be rendered multiple times

React has a **virtual dom** - it keeps a lightweight copy of the DOM and renders changes to that.

- If it sees the new result is actually different, THEN it updates the real DOM
- Makes for faster changes
- Means you don't have to track if a render is required

VIRTUAL DOM

Because the VDOM tracks what **it thinks** the page is like...

- It is a BAD idea to change the DOM outside of React
- You can, but it's a source of bugs
- React may overwrite changes it doesn't know about

You can change outside of the area React manages

- React does not cover the whole page, just everything inside some root element

REACT AND FORMS

How to handle forms without querying the DOM?

One way: gather input as it happens

```
const HasText = () => {
  const [text, setText ] = useState('');
  const updateText = (e) => setText(e.target.value);

  return (
    <div>
      I see {text}
      <input onChange={ updateText } value={text} />
      <button onClick={ () => setText('') }>Clear</button>
    </div>
  );
};
```