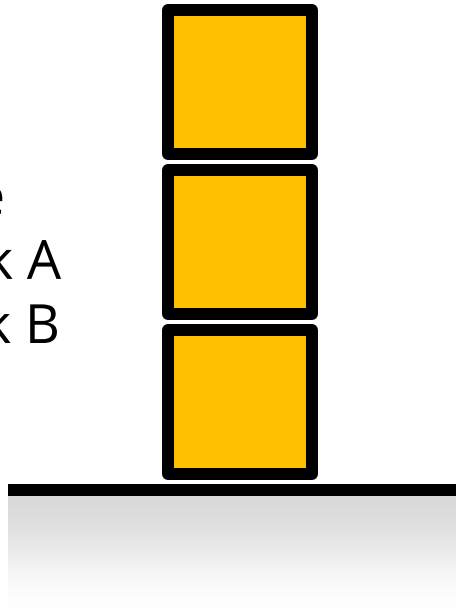
# Declarative

- UI Programming Paradigms
- Declarative Syntax
- Hyperscript and Virtual DOM
- Preact and JSX

# Imperative vs. Declarative

- Imperative Programming
  - describe *how* to achieve a result
- Declarative Programming
  - describe *what* result you want

**Imperative**
1. Place block A on table
2. Place block B on block A
3. Place block C on block B

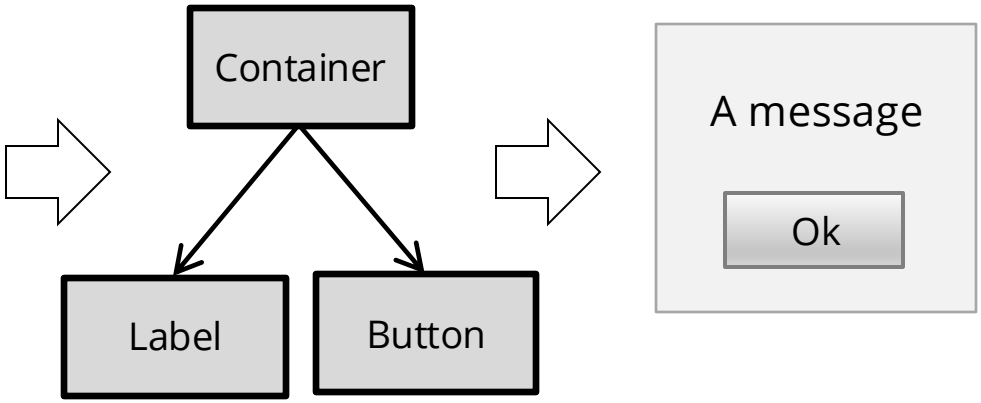


**Declarative**
A tower of 3 blocks on a table

# Imperative UI

- SimpleKit used an *imperative* paradigm to build a UI
  - Our goal is a *tree* of nodes with associated events
  - We write TypeScript to describe *how* to make that tree

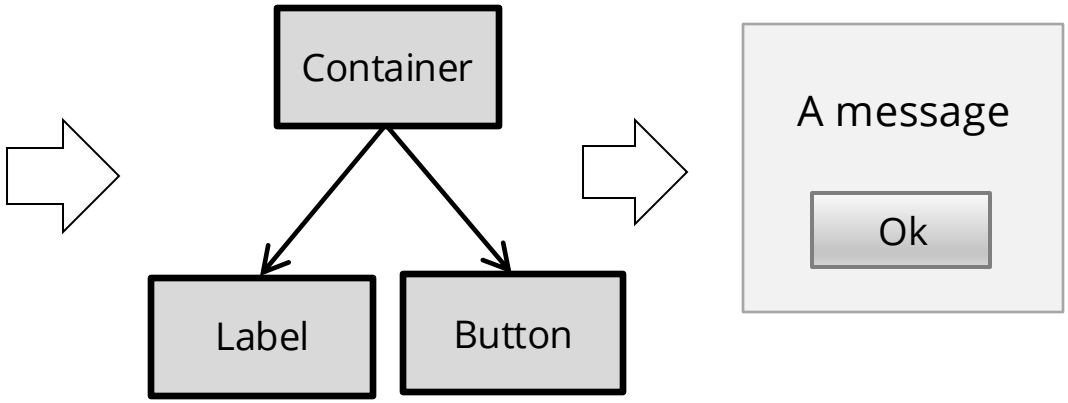
```
const root = new SKContainer()

const l = SKLabel("A message")
root.addChild(l)

const b = SKButton("Ok")
root.addChild(b)
b.addEventListener("action",
 () => doAction() );

setSKRoot(root)
```



# Declarative UI

- HTML can be a *declarative* paradigm to build a UI
  - Our goal is still a tree of nodes (called the DOM) with events
  - We write HTML to describe *what* the tree (DOM) is

```
<div class="container">
  <p>
    A message
  </p>
  <button
    onclick="doAction()">
    Ok
  </button>
</div>
```



# Setup for Next Demos

- Simple state

```
// state
let clicked = false;
function setClicked(value: boolean) {
  clicked = value;
  update();
}
```

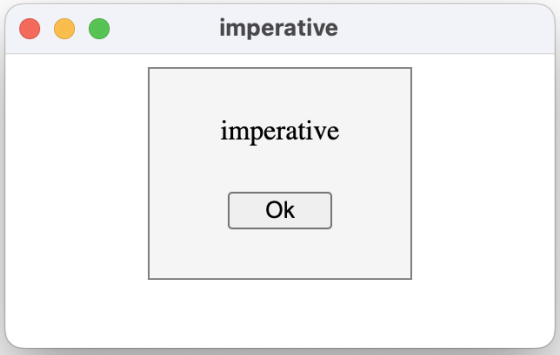
- When state changes, app is re-rendered

```
// when state changes, re-render the App
function update() {
  ...
}

// initial render
update();
```

Keeping it simple without MVC, better reactive methods for state are shown in next lecture

# imperative

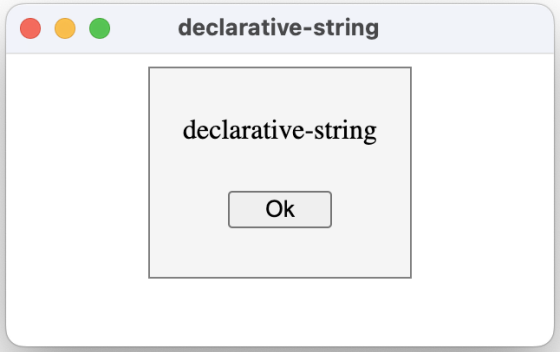
```
// create the UI tree for the app
function App() {
  const container = document.createElement("div");
  container.classList.add("container");

  const label = document.createElement("p");
  label.innerText = clicked ? "CLICKED" : "imperative";
  container.appendChild(label);

  const button = document.createElement("button");
  button.innerText = "Ok";
  container.appendChild(button);

  button.addEventListener("click", () => {
    setClicked(true);
  });

  return container;
}

// when state changes, re-render the app
... root.replaceChildren(App());
```



# declarative-string

```
// create the UI tree for the app
function renderApp(root: Element) {
  root.innerHTML = html`
    <div class="container">
      <p>${clicked ? "CLICKED" : "declarative-string"}</p>
      <button>Ok</button>
    </div>
  `;
  document.querySelector("button")?
    .addEventListener("click", () => {
      setClicked(true);
    });
}

// when state changes, re-render the app
function update() {
  const root =
    document.querySelector("#app")
      as Element;
  renderApp(root);
}
```

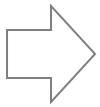
but add listener code is imperative



# Fully Declarative Syntax: HyperScript

- HyperScript is a language to generate descriptions of UI trees

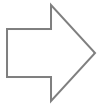
```
div { class "container" }
  p "some text"
  button "Ok"
```



```
<div class="container">
  <p>some text</p>
  <button>Ok</button>
</div>
```

- **`hyperscript`** is a npm package to *generate* HyperScript

```
const msg = "hi hyperscript";
h("div", { class: "container" }, [
  h("p", {}, msg),
  h("button", {}, "Ok"),
]);
```



```
<div class="container">
  <p>hi hyperscript</p>
  <button>Ok</button>
</div>
```

https://dev.to/fromaline/hyperscript-the-hidden-language-of-react-3d1f

# declarative-h

all declarative

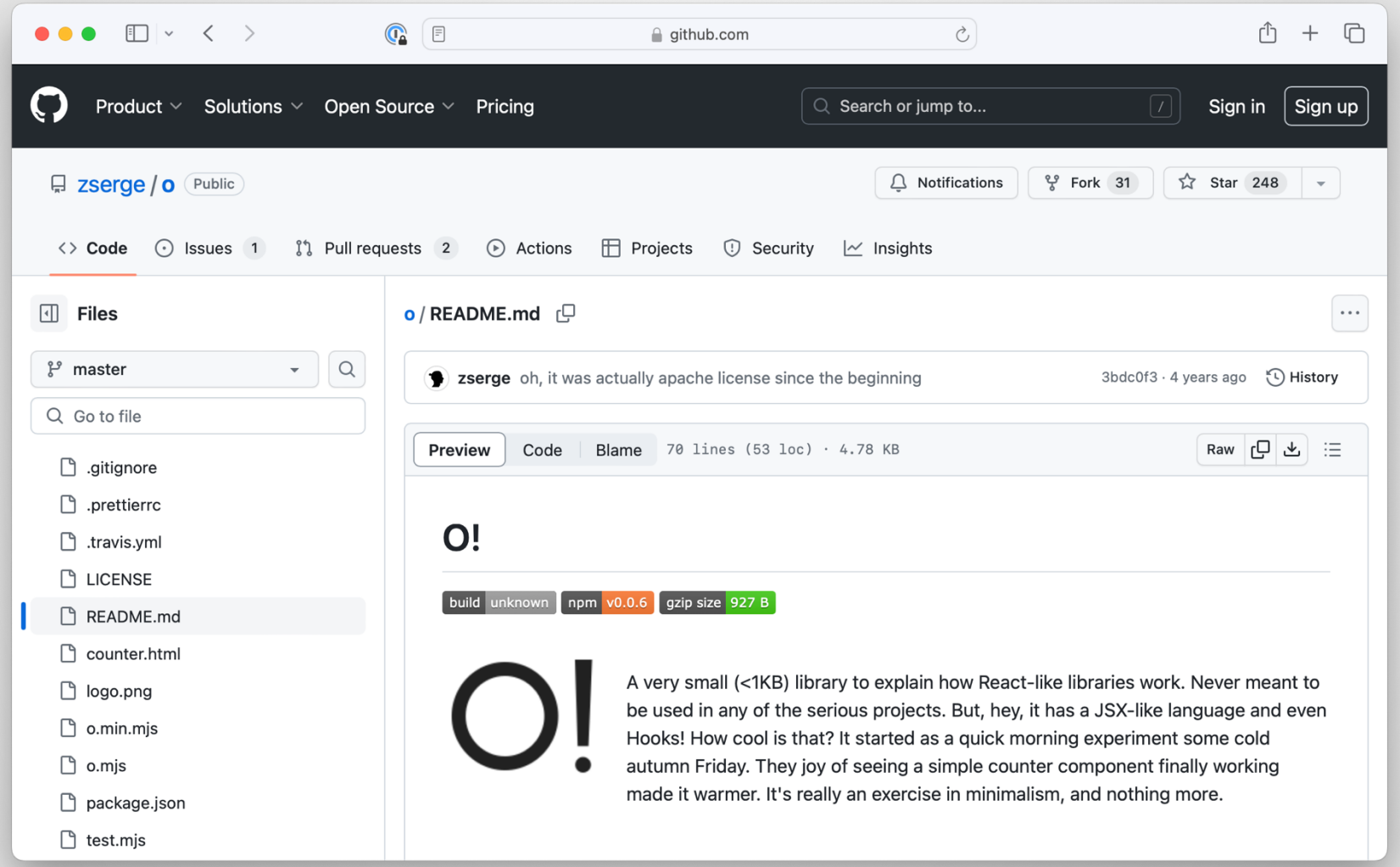
```
// create the UI tree for the app
function App() {
  return h("div", { class: "container" }, [
      h("p", null, clicked ? "CLICKED" : "declarative-h"),
      h("button", { onClick: () => setClicked(true) }, "Ok")
  ]);
}

// when state changes, re-render the app
function update() {
  render(App(),
    document.querySelector("#app") as Element);
}
```

# Virtual DOM

- Hyperscript function calls create a *representation* of UI tree
  - It's a JavaScript object
  - Commonly referred to as a virtual DOM (or just "vdom")
- Used for two purposes:
  1. "Render" an actual DOM using imperative methods

     explained next
  2. Lightweight abstraction of DOM to compare changes

     enables efficient DOM diffing for reactivity, ***explained next lecture***




Very minimal example of a reactive UI library

- https://github.com/zserge/o

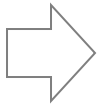
# hyperscript virtual node function

```
export function h(
    type: string,
    props: VNodeProps,
    children: VNodeChildren): VNode {
  return { type, props, children };
}
```

showing simplified version

---

```
h("div", { class: "foo" }, [
  h("button", {}, "Ok"),
]);
```



```
{
  "type": "div",
  "props": {
    "class": "foo"
  },
  "children": [
    {
      "type": "button",
      "props": {},
      "children": ["Ok"]
    }
  ]
}
```

# Render hyperscript virtual node to a DOM element

```
function _render(vnode: VNode): Node {

  // create the corresponding DOM element
  const el = document.createElement(vnode.type);

  // Copy vnode attributes into new DOM element
  const attributes = vnode.props || {};
  for (const key in attributes) {
    const value = attributes[key];
    // Set standard attribute
    el.setAttribute(key, value as string);
  }

  // Recursively render child nodes
  vnode.children.forEach((c) => el.appendChild(_render(c)));

  return el;
}
```

showing simplified version

# Render hyperscript with event attribute

- Example hyperscript definition with event

handler

onClick attribute

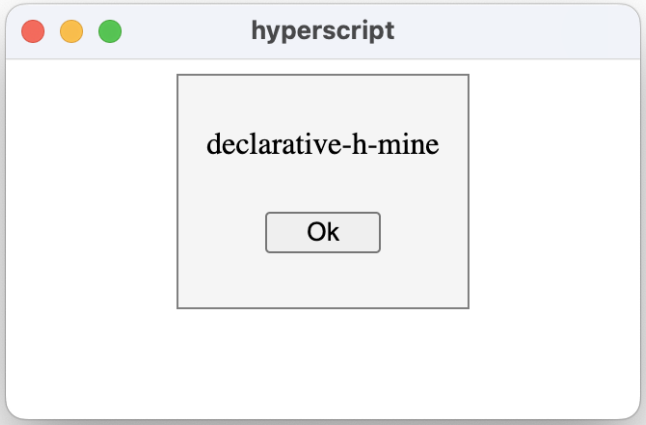
```
h("button",
  { onClick: () => (console.log("🔥 CLICKED!")) },
  "Ok");
```

- Additional code to ”render” event listeners

```
if (typeof value === "string") {
  // Set standard attribute
  el.setAttribute(key, value as string);
} else if (key.startsWith("on")) {
  // Handle event listener attributes
  const type = key.substring(2).toLowerCase();
  el.addEventListener(type, value as EventListener);
}
```

# declarative-h-mine

- `vdom.ts`
  - Implementations of `h` and `render`
- `main.ts`
  - Main body same as declarative-h
- Demos
  - log the VDOM tree as JSON to see structure
  - Compare to JSON log in declarative-h



# Preact and JSX

# Preact

- First public release in 2014
  - by Jason Miller, a software engineer at Google

Goals:

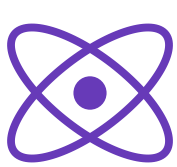
- Render quickly & efficiently
- Small size, lightweight (approximately 3.5 kB)
- Effective memory usage (avoiding GC thrash)
- Understanding codebase should take no more than a few hours
- Aims to be largely compatible with the React API
  - preact/compat to enable React compatibility mode

# Differences with React

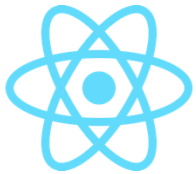
Preact is not intended to be a reimplementation of React

Key differences:

- Preact uses native DOM events
  - React has its own synthetic event system
    (for historical reasons, patch issues in older browsers like IE8)
  - quirks: onInput vs. onChange, onDblClick vs onDoubleClick
- Preact treats Children nodes as native JavaScript arrays
  - React has its own object for managing Children
- Preacts supports "class" to set class attribute
  - React uses "className"



vs.



https://preactjs.com/guide/v10/differences-to-react

# Preact Setup and Environment

- Using Vite

  `npm create vite@latest`

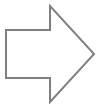
  then choose “Preact” and “TypeScript”

- Node project is very similar to Vanilla TypeScript
  - adds support for .jsx files
    (see JSX and Preact settings in tsconfig.app.json)

# Declarative Syntax: JSX

- Describe DOM trees with a mixture of JavaScript and HTML
- JavaScript files with JSX have ".jsx" extension
  (TypeScript files with JSX have ".tsx" extension)
- Files with JSX are compiled into JavaScript
  (into *hyperscript* function calls)
- Example syntax:

```
const msg = "hi JSX";
<div class="container">
    <p>{msg}</p>
    <button>Ok</button>
</div>
```

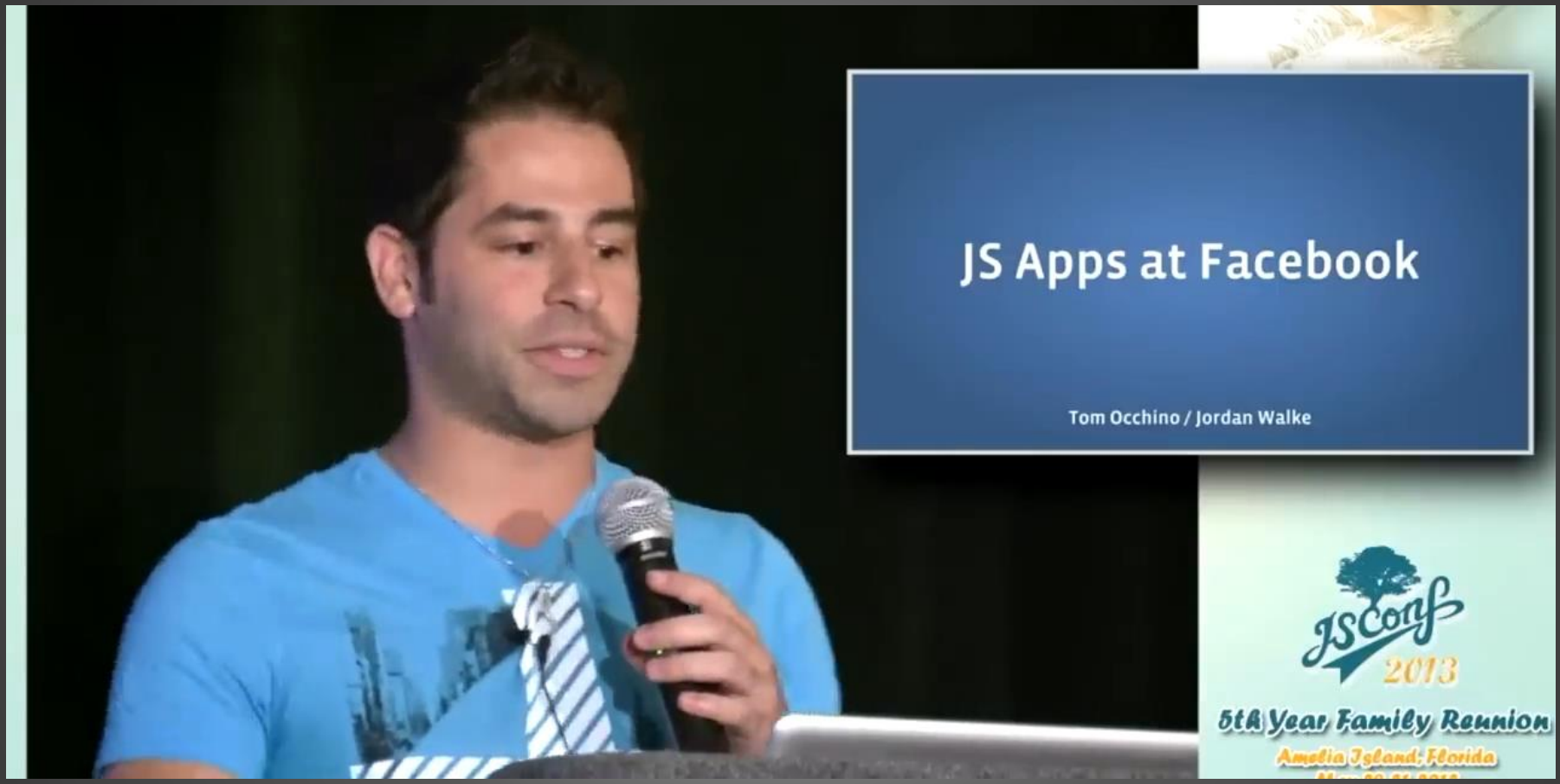


```
<div class="container">
  <p>hi JSX</p>
  <button>Ok</button>
</div>
```

JSX and HTML look almost the same

# declarative-jsx

```
function App() {
  return (
    <div class="container">
      <p>{clicked ? "CLICKED" : "declarative-jsx"}</p>
      <button onClick={() => setClicked(true)}>Ok</button>
    </div>
  );
}
```




Excerpt from React.js: The Documentary

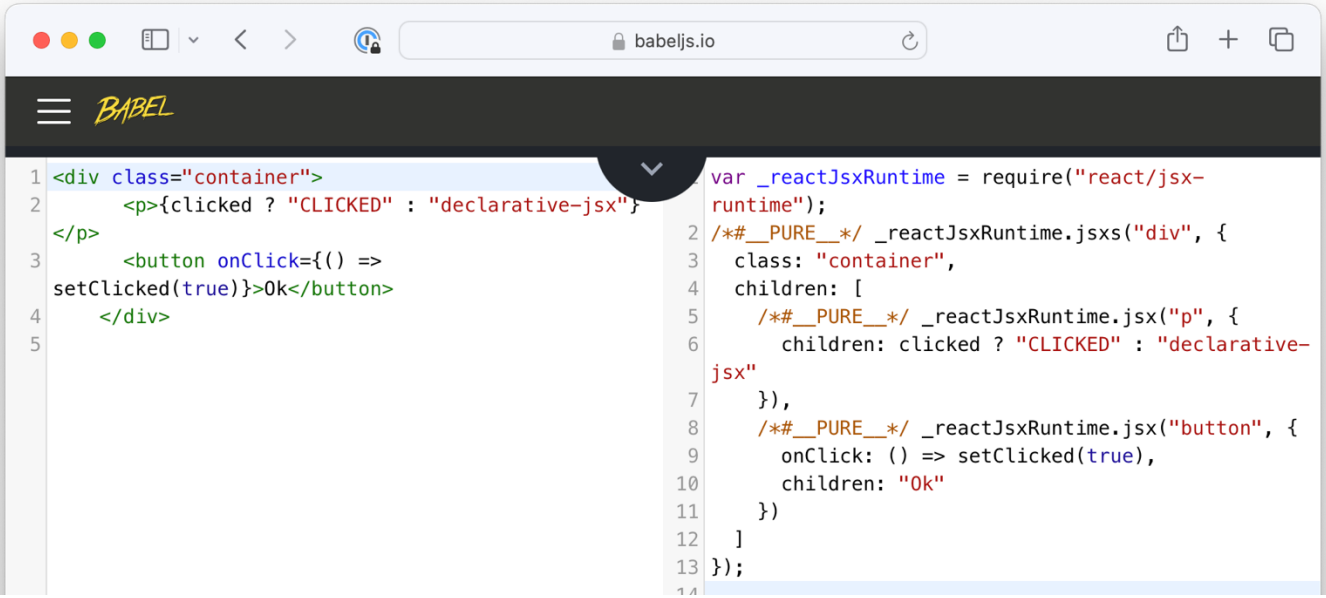
- https://youtu.be/8pDqJVdNa44?si=uptfGr9KTMuKMkcc&t=2171

# How JSX Works

In practice, may not literally be hyperscript, but something very similar

- JSX is just “syntactic sugar” for hyperscript

1. JSX is compiled into hyperscript
   - Try pasting JSX into this bable repl https://babeljs.io/repl#
2. hyperscript is used to render
   - h function to create a Virtual DOM object
   - render function to create DOM from Virtual DOM




https://jasonformat.com/wtf-is-jsx/

# Preact Components

- Components are the building blocks of a Preact application
- Components have custom properties (i.e. "props")
- Example using a custom component:

```
const vdom = (
  <div>
    <h1>My component is below here:</h1>
    <MyComponent msg="hi component" />
  </div>
);
```

a custom component with a custom msg "prop"

# Functional Components

Functions are the most common way to create components:

```
function MyComponent(props: { msg: string }) {
  return (
    <div class="container">
      <p>{props.msg}</p>
      <button>Ok</button>
    </div>
  );
}
```

a custom "prop"

# Class Components

Avoid using classes to define components, you should define components with functions.

- Components can also be defined as classes
  - method is no longer common, functional components are better

```
class MyComponent extends Component<{ msg: string }> {

  constructor(props: { msg: string }) {
    super(props);
  }

  render() {
    return (
      <div class="container">
        <p>{this.props.msg}</p>
        <button>Ok</button>
      </div>
    );
  }
}
```

a custom "prop"

https://preactjs.com/guide/v10/components#class-components

# Component Children

- Components can be nested like HTML elements
- Components can have HTML or Component nodes as children
- This enables control over how Virtual DOM elements nested within a component should be rendered
- The Array of children is a *special implicit prop*

```
function Container(props: { children: any }) {
  return <div class="container">{props.children}</div>;
}

const vdom = (
  <Container>
    <p>Text</p>
    <button>Ok</button>
  </Container>
);
```

render children as they are

# Props Type Definitions

- TypeScript requires type definition for Component props
- Best practice is to define a `MyComponentProps` type
  - can have optional props

```
type NumberBoxProps = {
  num: number;
  colour?: string;
};

function NumberBox({ num, colour = "grey" }: NumberBoxProps) {
  return
    <div style={`background-color: ${colour};`}>
      {num}
    </div>;
}
```

optional prop

default prop value

https://preactjs.com/guide/v10/typescript/#function-components

https://medium.com/codex/how-to-type-react-props-in-typescript-f40c41ff3c6f

# Props Destructuring

- Avoid `props.myprop` syntax by destructuring props argument
  - makes it easier to assign default props values

```
type NumberBoxProps = {
  num: number;
  colour: string;
};

function NumberBox({ num, colour }: NumberBoxProps) {
  return
    <div style={`background-color: ${colour};`}>
      {num}
    </div>;
}
```

Without props argument destructuring:

```
function NumberBox(props: NumberBoxProps) {
  return
    <div style={`background-color: ${props.colour};`}>
      {props.num}
    </div>;
}
```

https://preactjs.com/guide/v10/typescript/#function-components

https://medium.com/codex/how-to-type-react-props-in-typescript-f40c41ff3c6f

# Defining Events in Components

- Preact uses standard DOM events with declarative syntax
- If event handler is small, *include function definition inline*:

```
const jsx = <button onClick={() => console.log("click")}>
  Click
</button>
```

- If event handler is more complex, then call *handler function*:

```
function handleClick() {
  console.log("click");
}

const jsx = <button onClick={handleClick}>Click</button>
```

- Event handlers can be passed as props to components

# JSX Must Evaluate to an Expression

- An expression is a valid unit of code that resolves to a value
- JSX is an expression, and everything in JSX must be an expression
  - to insert a JavaScript expression into JSX, use { }
- To insert the value of a variable

```
const msg = "Hello World";
const jsx = <p>{msg}</p>;
```

- To iterate through an array, typically use map

```
const items = ["a", "b", "c"];
const jsx = <ul>{ items.map((item) =>
    <li>{item}</li>
  )}</ul>;
```

- For conditional logic, typically use ternary operator

```
const jsx = <p>{isDone ? "Done" : "Not Done"}</p>
```

# Dynamic Attribute Values

Option 1: Using *template literal*

```
function NumberBox({ num, colour }: NumberBoxProps) {
  return <div style={`background-color: ${colour};`}>
           {num}
         </div>;
}
```

template literal with JavaScript expression

Option 2: Using *hyperscript object*

```
function NumberBox({ num, colour }: NumberBoxProps) {
  return <div style={{backgroundColor: colour}}>
           {num}
         </div>;
}
```

note camelcase used, not - separator

# Components Must Return One Root Node

- If you don't naturally have a single root node, best approach is to wrap component nodes in <Fragment> node

```
function App() {
  return (
    <LeftView />
    <RightView />
  );
}
```

**Error:**
More than one root node not allowed

```
function App() {
  return (
    <div>
      <LeftView />
      <RightView />
    </div>
  );
}
```

**Solution 1:**
Insert  to make one root node

***Issue:***
extra  in DOM

```
function App() {
  return (
    <Fragment>
      <LeftView />
      <RightView />
    </Fragment>
  );
}
```

**Solution 2:**
Use special <Fragment> node to make one root node

***Note:***
<Fragment> is not rendered in DOM

```
function App() {
  return (
    <>
      <LeftView />
      <RightView />
    </>
  );
}
```

**Solution 2 Variation:**
Use <> and </> as shorthand for <Fragment>

https://preactjs.com/guide/v10/whats-new/#fragments

# count

- Counter example from MVC and HTML-CSS as Preact components
- Very simple global state forces re-render each time it changes
  - passes count as prop
  - proper state management and styling covered in next lectures
- Demos
  - getting ref to app div
  - App fragment usage
  - LeftView onClick event and count prop
  - NumberBox (in RightView)
  - NumberBox style expression
  - RightView iteration methods
  - RightView optional colour prop
  - Some  "prop drilling" to set colour