React Introduction

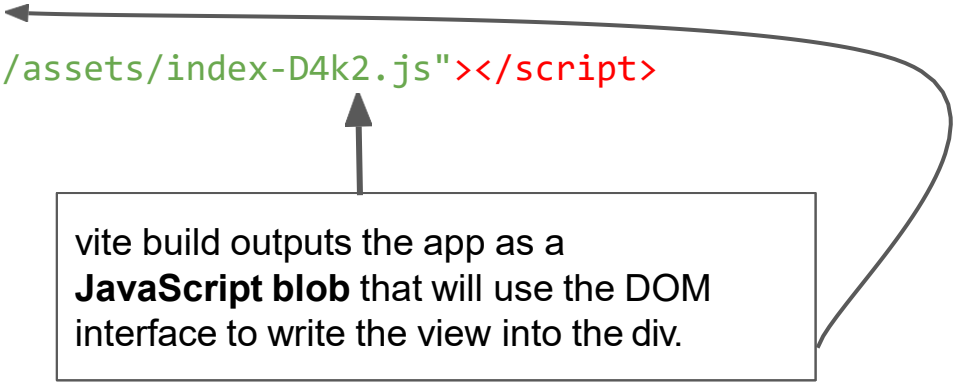
Mendel Rosenblum

React react.dev

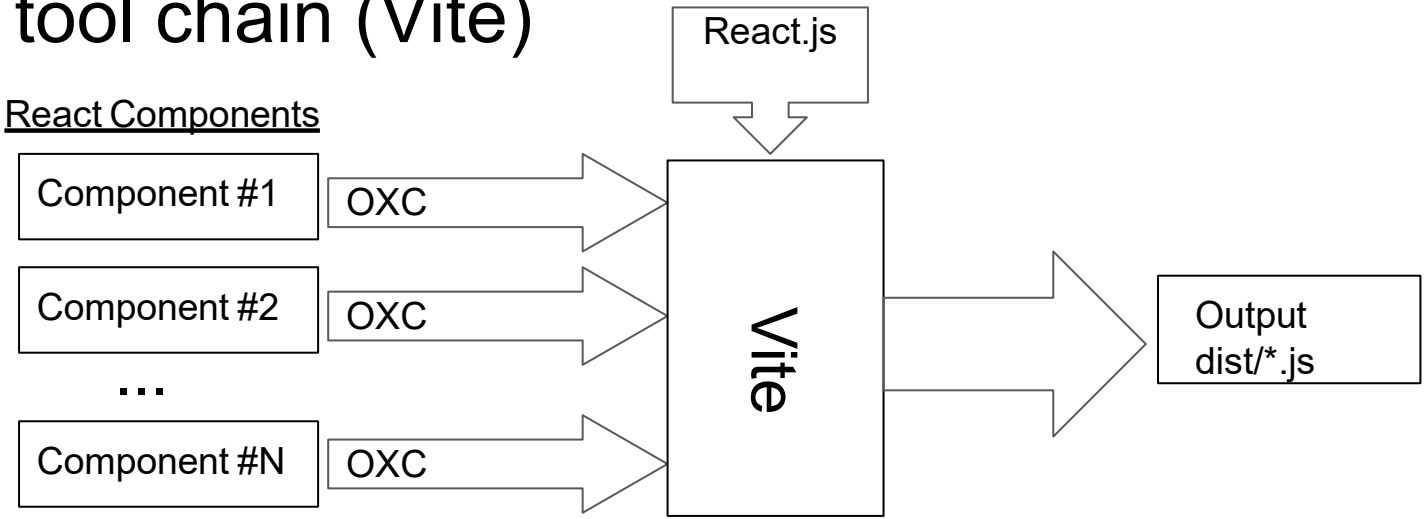
- JavaScript library for building web application user interfaces
  - Runs in the browser - fast updates without page reloads
  - Less opinionated: only specifies rendering view and handling user interactions
- UI built from components (function components + Hooks)
  - View described with JSX: HTML-like syntax inside JavaScript
  - One-way data flow: props go down, events go up
  - Server rendering optional, via frameworks such as Next.js
- Focus on supporting programming in the large and single page applications
  - Modules, reusable components, testing, etc.

ReactJS Web Application Page

```
<!doctype html>
<html>
   <head>
      <title>CS142 Example</title>
   </head>
   <body>
       <div id="reactapp"></div>
       <script type="module" src="/assets/index-D4k2.js"></script>
   </body>
</html>
```



React tool chain (Vite)



OXC - Transform language features (e.g. TypeScript, JSX) to basic JavaScript

Vite - Bundle modules and resources (CSS, images) via Rolldown

Output loadable with single `script` tag in any browser

main.jsx - Render element into browser DOM

```
import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';
import ReactAppView from './components/ReactAppView';
const where = document.getElementById('reactapp');
const root = createRoot(where);
root.render(<StrictMode><ReactAppView /></StrictMode>);
```

ES Modules - Bring in React and web app React components.

Creates a React root at the div with id=reactapp and renders the tree of React elements (single component named **`ReactAppView`**) into the browser's DOM.

components/ReactAppView.jsx - Function component

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

function ReactAppView(props) {
    const [yourName, setYourName] = useState('');
    ...
    return ( ... );
}

export default ReactAppView;
```

Plain function: `props` is its argument. The useState Hook replaces this.state and the constructor.

No render() method - the function returns the React element tree of the component's view.

ReactAppView function body

```
<div>
   <label>Name: </label>
   <input type="text" ... />
   <h1>Hello {yourName}!</h1>
</div>
```

```
function ReactAppView() {
     const [yourName, setYourName] = useState('');
     const label = createElement('label', null, 'Name: ');
     const input = createElement('input',
             { type: 'text', value: yourName,
                 onChange: (event) => setYourName(event.target.value) });
     const h1 = createElement('h1', null,
                              'Hello ', yourName, '!');

     return createElement('div', null, label, input, h1);
}
```

Returns element tree with `div` (`label`, `input`, and `h1`) elements



ReactAppView function body w/o variables

```
function ReactAppView() {
     const [yourName, setYourName] = useState('');
     return createElement('div', null,
          createElement('label', null,'Name: '),
          createElement('input',
              { type: 'text', value: yourName, onChange: (e) =>
                setYourName(e.target.value) }),
          createElement('h1', null,
                    'Hello ', yourName, '!')
        );
}
```

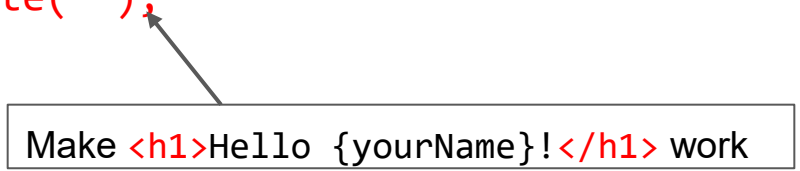
Use JSX to generate calls to createElement

```
function ReactAppView() {
   const [yourName, setYourName] = useState('');
   return (
      <div>
         <label>Name: </label>
         <input
            type="text"
            value={yourName}
            onChange={(e) => setYourName(e.target.value)}
         />
         <h1>Hello {yourName}!</h1>
      </div>
   );
}
```

Component state and input handling

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

function ReactAppView() {
  const [yourName, setYourName] = useState('');
  function handleChange(event) {
    setYourName(event.target.value);
  }
  ....
}
```



- Input calls `setYourName`, which causes React to call ReactAppView() again

One way binding: Type 'D' Character in input box

- JSX statement: `<input type="text" value={yourName} onChange={handleChange} />`

  Triggers `handleChange` call with `event.target.value` == "D"

- handleChange - `setYourName(event.target.value);`

  `yourName` becomes "D" on the next render

- React calls ReactAppView again:
- Feature of React - highly efficient re-rendering





Function components: no `this` binding problem

```
function ReactAppView() {
   ...
   function handleChange(event) {
      setYourName(event.target.value);
   }
   ...
   return <input type="text" value={yourName} onChange={handleChange} />;
}
```

Works! handleChange is a plain function with no `this` to lose.

Legacy class components: a method passed to onChange lost `this`, so it needed `bind(this)`, a class-field arrow function, or an arrow function in JSX.



A digression: `camelCase` vs `dash-case`

Word separator in multiword variable name

- Use dashes: `active-buffer-entry`
- Capitalize first letter of each word: `activeBufferEntry`

Issue: HTML is case-insensitive but JavaScript is not.
React's JSX has HTML-like stuff embedded in JavaScript.

React: Use camelCase for attributes

Exception: `aria-label` and `data-id` keep their dashes

Programming with JSX

- Need to remember: JSX maps to calls to `createElement`
  - HTML-like syntax converted to function calls
- `createElement(type, props, ...children);`
  - `type`: HTML tag (e.g. `h1, p`) or a `function component`
  - `props`: attributes (e.g. `type="text", onChange={}`) Uses camelCase!
  - `children`: Zero or more children which can be either:
    - String or numbers
    - A React element
    - An Array of the above

JSX compiles to jsx() calls since React 17

- Need to remember: JSX compiles to calls to `jsx()`
  - `<p id="a">Hi</p>` → `jsx('p', { id: 'a', children: 'Hi' })`
- `jsx(type, props, key);`
  - `type`: HTML tag (e.g. `h1, p`) or a `function component`
  - `props`: attributes (e.g. `type="text", onChange={}`) Uses camelCase!
  - `key`: taken out of props and passed as a separate argument
  - `props.children`: zero or more children, which can be either:
    - String or numbers
    - A React element
    - An Array of the above



JSX templates must return a valid `children` param

- Templates can have JavaScript scope variables and expressions
  - `<div>{foo}</div>`
    - Valid if `foo` is in scope (i.e. if foo would have been a valid function call parameter)
  - `<div>{foo + 'S' + computeEndingString()}</div>`
    - Valid if `foo & computeEndingString` in scope
- Template must evaluate to a value
  - `<div>{if (useMongolian) { … } }</div>` - **Doesn't work: `if` isn't an expression!**
  - Same problem with "for loops" and other JavaScript statements that don't return values
- Leads to contorted looking JSX: Example: Anonymous immediate functions
  - `<div>{ (function() { if …; for ..; return val;})() }</div>`
  - `Better: const val = compute(); return <div>{val}</div>;`

Conditional render in JSX

- Use JavaScript Ternary operator (?:)

```
<div>{useMongolian ? <b>Сайн уу</b> : "Hello"}</div>
```

- Use JavaScript variables

```
let greeting;
const en = "Hello";  const mn = <b>Сайн уу</b>;
const { useMongolian } = props;
if (useMongolian) {greeting = mn} else {greeting = en};

<div>{greeting}</div>
```



Iteration in JSX

- Use JavaScript array variables

```
let listItems = [];
for (let i = 0; i < data.length; i++) {
   listItems.push(<li key={data[i]}>Data Value {data[i]}</li>);
}
return <ul>{listItems}</ul>;
```

- Functional programming

```
<ul>{data.map((d) => <li key={d}>Data Value {d}</li>)}</ul>
```

`key=` attribute improves efficiency of rendering on data change

Styling with React/JSX

```
import './ReactAppView.css';

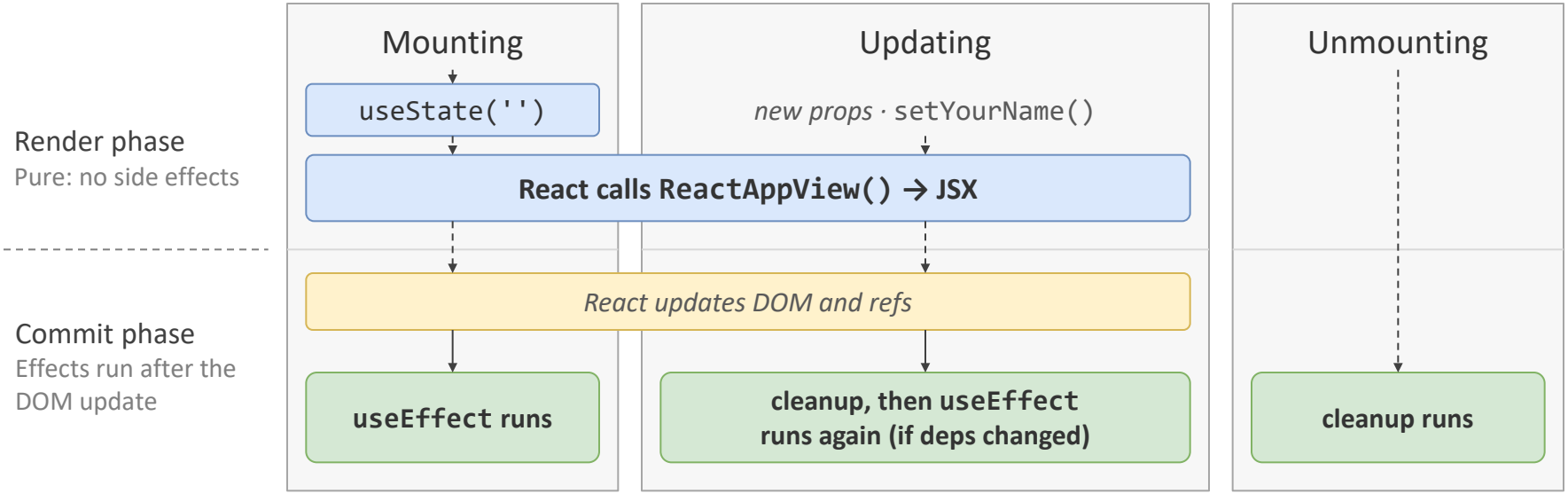
function ReactAppView() {
  ...
    // no render(): the function returns JSX
    return (
        <span className="cs142-code-name">
          ...
        </span>
    );
```

Vite can import CSS style sheets:

```
.cs142-code-name {
  font-family: Courier New, monospace;
}
```

Must use `className=` for HTML `class=` attribute (JS keyword conflict)

Component lifecycle with Hooks



Cleanup = the function returned from `useEffect`. It replaces `componentDidMount`, `componentDidUpdate` and `componentWillUnmount`.



Example of lifecycle with useEffect

```
function Example() {
  const [counter, setCounter] = useState(0);
  useEffect(() => {   // Start 2 sec counter
    const incFunc = () => setCounter((prev) => prev + 1);
    const timerID = setInterval(incFunc, 2 * 1000);
    return () => clearInterval(timerID);   // Shutdown timer
  }, []);   // [] = run once, after mount
  ...
}
```



Function Components and props

- A React component is a function that takes props and returns JSX

```
function MyComponent(props) {
  return <div>Миний нэр {props.name}</div>;
}
```

Or using destructuring...

```
function MyComponent({name}) {
  return <div>Миний нэр {name}</div>;
}
```

Use:

```
<MyComponent name='Функц компонент! JSX ашиглав.'/>
```

React Hooks (react.dev/reference/react/hooks)

- useState, useEffect, … add state and effects to function components

Example React Hooks

```
import { useState, useEffect } from 'react';
function Example() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `Та ${count} удаа дарлаа.`;
  }, [count]);
  return (
    <div>
      <p>Та {count} удаа дарсан.</p>
      <button onClick={() => setCount(count + 1)}>
         Дарах
      </button>
    </div>
  );
}
```



Communicating between React components

- Passing information from parent to child: Use props (attributes)

```
<ChildComponent param={infoForChildComponent} />
```

- Passing information from child to parent: Callbacks

```
function parentCallback(infoFromChild) { /* … */ }

<ChildComponent callback={parentCallback} />
```

- `React Context: createContext() + useContext()`
  - Global variables for subtree of components