Session 03

JSX and Rendering

CIT 2025, section 3

Morten Rhiger

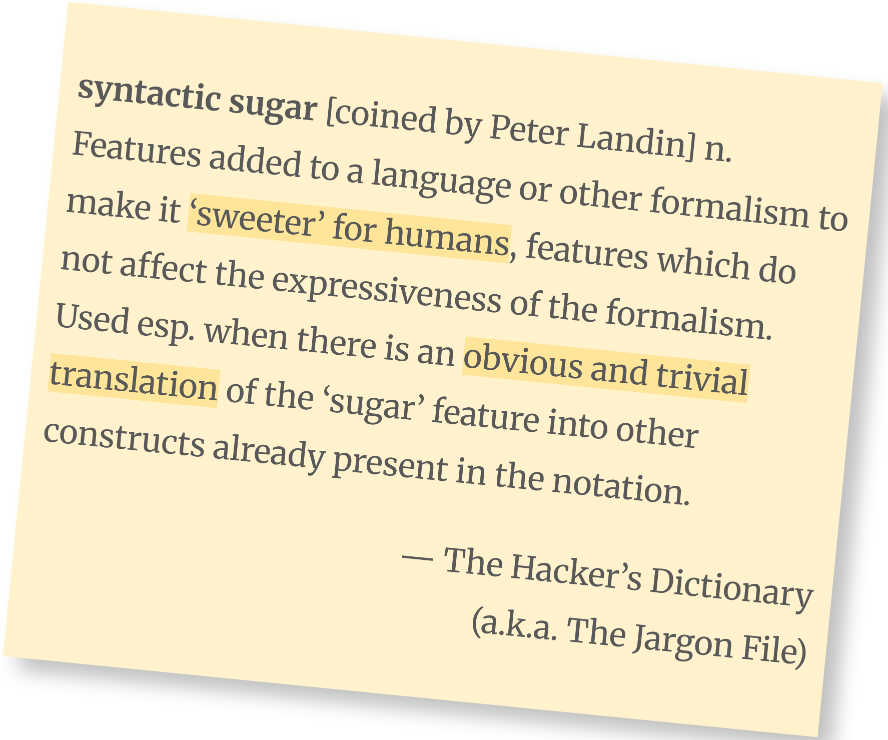
In this session

JSX explained.
How React renders elements.

1. Syntax of JSX
    - Elements in JavaScript
    - JavaScript in elements
    - Attributes
2. HTML & the DOM (a recap)
3. Translating JSX to JavaScript
4. Rendering and the virtual DOM

JSX (JavaScript + XML)

- JSX is “syntactic sugar” for JavaScript that creates and inserts “HTML” in the browser.
- “Syntax” characterize what we can legally write.
- A “grammar” formally describe syntax.



syntactic sugar [coined by Peter Landin] n. Features added to a language or other formalism to make it ‘sweeter’ for humans, features which do not affect the expressiveness of the formalism. Used esp. when there is an obvious and trivial translation of the ‘sugar’ feature into other constructs already present in the notation.

— The Hacker’s Dictionary
(a.k.a. The Jargon File)

a small subset of

Grammar of JavaScript

statement:

```
    expression ;
    if ( expression ) statement
    while ( expression ) statement
    return expression ;
    block
    ⋮
```

expression:

```
    constant
    name
    expression + expression
    ( name, … ) => expression
    ⋮
```

block:

```
    { declaration … statement … }
```

declaration:

```
    let name = expression ;
    const name = expression ;
    function name ( name, … ) block
```

constant: one of

```
    1   2   3   …   "a"   "foo"   …
```

name: one of

```
    x   y   z   f   foo   bar   …
```

a very small subset of

Grammar of JSX

```
statement:
        expression ;
        if ( expression ) statement
        while ( expression ) statement
        return expression ;
        block
        ⋮

expression:
        constant
        name
        expression + expression
        ( name, … ) => expression
        element          ←
        ⋮
```

```
element:
        < name > chunk … </ name >

chunk:
        text
        element
        { expression }
```

a larger subset of

Grammar of JSX

```
statement:
        expression ;
        if ( expression ) statement
        while ( expression ) statement
        return expression ;
        block
        ⋮

expression:
        constant
        name
        expression + expression
        ( name, … ) => expression
        element
        ⋮
```

```
element:
        < name attribute … >  chunk …  </ name >
        < name attribute … />
        <>  chunk …  </>

chunk:
        text
        element
        { expression }

attribute:
        name = " text "
        name = { expression }
        name
        { ...expression }
```

JSX element syntax

`<h1>Hello</h1>`	An HTML element. (Starts with lowercase letter.)
`<Logo/>`	A custom component. (Its name is capitalized.)
`<br />`	HTML element with no child nodes. (The "/" is required.)
`<br></br>`	Ditto.
`<img src="images/bob.png"/>`	HTML element with one attribute, `src`.
`<User name="Bob"/>`	Custom component receives prop name with value `"Bob"`.
`<Page> <Nav/> <Main/> </Page>`	Component Page receives prop `children` (an array containing the two child components).
`<Nav/>`	Prop `children` is `undefined` for this instance of component Nav.
`<> <Nav/> <Main/> </>`	Fragment: Group of several components with no wrapper element.
`<Fragment> <Nav/> <Main/> </Fragment>`	Ditto.

Escaping JSX

`<p> { } </p>`	Empty escape, render nothing.
`<p> { "Bob" } </p>`	Render the content of the string.
`<p> { false } </p>`	A falsy value renders nothing, except if `0`.
`<p> { 23 } </p>`	Render a number, even if `0`.
`<p> { <b>Bob</b> } </p>`	Render an HTML element.
`<p> { <User name="Bob" /> } </p>`	Render what `<User name="Bob" />` renders.
`<p> { ["Bob", <Logo/>] } </p>`	An array renders to the sequence of what the array elements renders to.

JSX attribute syntax

`<img src="images/bob/png" />`	HTML element with one static string attribute.
`<img src={IMGDIR + "bob.png"} />`	Passing a computed string.
`<User name="Bob" age={23} />`	Pass number in `{ … }`, not `" … "`. (Recommended.)
`<User name="Bob" admin={false} />`	Pass booleans in `{ … }`, not `" … "`. (Required!)
`<User name="Bob" admin />`	Prop `admin` is `true`.
`<User name="Bob" />`	Prop `admin` is `undefined` (i.e., falsy).
`<button onClick={myHandler}>Hit me</button>`	Pass a function in `{ … }` (e.g. an event handler).
`<button onClick={() => alert("Ouch")}>Hit me</button>`	Any function expression will do.
`<p style={{backgroundColor: "#f00"}}> … </p>`	Pass an object in `{ … }`.
`<User {...bob} />`	Pass object containing several props.

Conditional rendering (React)

- Render component **only if** condition:
  ```
  <> … {condition && <Component/>} … </>
  ```
- Render value **with default** (when falsy):
  ```
  <> … {value || default} … </>
  ```
- Render this **if** condition, **else** that:
  ```
  <> … {condition ? this : that} … </>
  ```
- Passing **boolean props**:
  ```
  <User isAdmin={true} />         Flag set
  <User isAdmin />                The same

  <User isAdmin={false} />        Flag unset

  <User isAdmin={undefined} />    Flag "unset"
  <User />                        The same
  ```

```
function Webshop({loggedIn}) {

  const [avatar, setAvatar] = useState(null);

  return (
    <div>
      {loggedIn && <LogoutButton/>}

      {avatar || <DefaultAvatar/>}

      Your cart contains
      {cart.length}
      {cart.length === 1 ? "item" : "items"}
    </div>
  );
}

… <Webshop loggedIn /> …
```

HTML attributes in JSX

- HTML attributes when in JSX: (Some must be renamed.)

`style`		
`id`		
`class`	⟶	`className`
`for`	⟶	`htmlFor`

- Rename HTML event handlers to **camelCase** when using in JSX:

`onclick`	⟶	`onClick`
`onchange`	⟶	`onChange`
`onmouseover`	⟶	`onMouseOver`
`onmouseout`	⟶	`onMouseOut`
⋮		

- When **using inline CSS** in JSX, style properties must be in **camelCase** and with **no dashes:**

`background-color`	⟶	`backgroundColor`
`font-family`	⟶	`fontFamily`
`text-align`	⟶	`textAlign`
`padding-top`	⟶	`paddingTop`
`margin-right`	⟶	`marginRight`
`border-bottom`	⟶	`borderBottom`
`border-bottom-color`	⟶	`borderBottomColor`
`border`	⟶	`border`
`border-top-left-radius`	⟶	`borderTopLeftRadius`
⋮		

Tips, tricks, and pitfalls

- JavaScript **implicitly terminates lines** with " ; " when possible. Thus, remember parentheses here:

```
return (
  <p>
    Hello <b>world.</b>
  </p>
);
```

- JSX **removes whitespace** between elements or components **on separate lines** (only). If needed, include a space as an expression (on its own line):

```
<p>
  <b>hello</b>
  {" "}
  <i>world.</i>
</p>
```

- Comments in JSX elements?

```
<p>
  // No, not really:
  // This is not a comment!
  /* Nor is this! */

  { /* But this is a (JavaScript) comment */ }
  { // and so is this.
  }

  <User name="Bob"
        // And this is a (JSX) comment and
        /* so is this. */
        age={23}
        />
</p>
```

Tips, tricks, and pitfalls

- Only pass elements or components in prop `children` if **rendering all of them, in order.** This component receives seven child nodes: (Why?)

  ```
  <Hide n={2}>  <b>Last</b>  <b>element</b>  <b>hidden</b>  </Hide>
  ```

  whereas this receives only three:

  ```
  <Hide n={2}>
    <b>Last</b>
    <b>element</b>
    <b>hidden</b>
  </Hide>
  ```

- Better pass elements in a prop containing array:

  ```
  <Hide n={2} elems={[<b>Last</b>, <b>element</b>, <b>hidden</b>]} />
  ```

Exercises

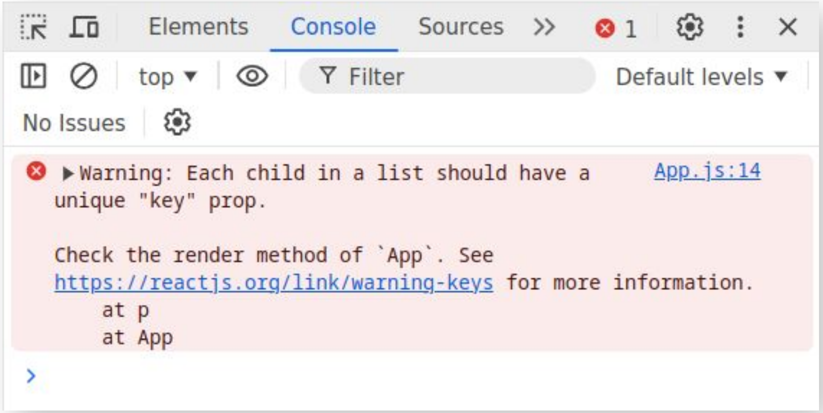
1. Inspired by the previous slide, implement a component `Hide` that takes a prop n and an array of components such that all but the nth of those components are rendered. (Use array method `filter`.)

   Implement one version that takes subcomponents in the `children` prop, and another that takes subcomponents in a custom prop `elems`.

2. Implement (a sketch of) a `Page` component, that takes as prop `children` the content of a page and that wraps that content in a `<main>` element and that inserts a `<header>` above and a `<footer>` below.
   (You decide the content of the header and footer.)

   Instantiate this Page component from component `App`.

Escaping arrays



```
Warning: Each child in a list should have a          App.js:14
unique "key" prop.

Check the render method of `App`. See
https://reactjs.org/link/warning-keys for more information.
    at p
    at App
```

- When inserting an array, each element must have a unique prop called key.

  (React needs keys to keep track of how elements move, appear, or disappear.)

```
<>
  <div>
    {
      [ <p>one</p>,
        <p>two</p> ]
    }
  </div>
  <div>
    {
      [ <p key={1}>one</p>,
        <p key={2}>two</p> ]
    }
  </div>
</>
```

Escaping arrays, with map

Typical scenario:

- We have an array of “raw” data.
- We want to display elements or components for each data entry.

Solution:

- **Transform data to UI** using `map`.
- Remember prop `key` on UI elements (whether HTML or custom components.)
- Keys must be unique (among this call to `map`).
- Key must not be computed (i.e., based on array index).
- **Pick a key from the data entry.**

```
const users = [
  { name: "Bob",  age: 23 },
  { name: "Mary", age: 8  },
];

const App = () =>
  <div>
    {
      users.map((u, i) =>
        <User key={u.name}
              name={u.name}
              age={u.age} />)
    }
  </div>;
```

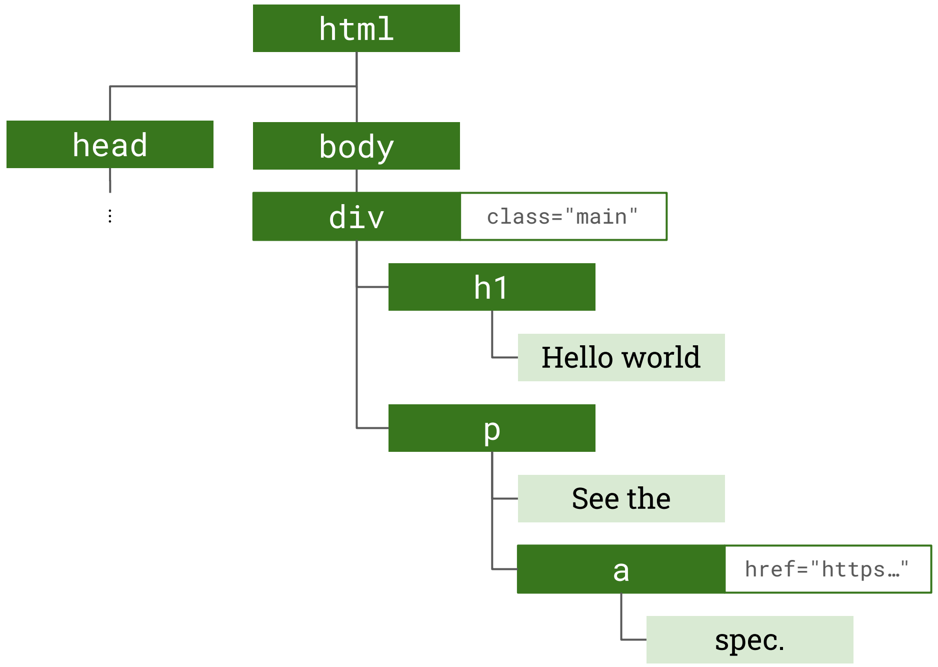
HTML
(HyperText Markup Language)

- HTML is a **notation** describing the content of web pages.
- Stored in files; send over the network.
- The browser (and nothing else) understands HTML:
  - It **reads** HTML, then
  - **renders** the HTML, and
  - **displays** the result in the browser window.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="description" content="A test">
    <link rel="stylesheet" href="styles.css">
    <title>Hello World</title>
  </head>
  <body>
    <!-- The page -->
    <div class="main">
      <h1>Hello World</h1>
      <p>
        See the
        <a href="https://html.spec.whatwg.org/">
          spec.
        </a>
      </p>
    </div>
  </body>
</html>
```

DOM
(Document Object Model)

- The DOM is the (browser's) **representation** of a web page (e.g., an HTML document).
- Hierarchy of **nodes** (i.e., a **tree**):
  - Root node
  - Child nodes
  - Text nodes
  - Attributes
- JavaScript may **manipulate** the DOM:
  - Inspect, create, insert, and remove elements or text.
  - Inspect, add, remove, and modify attributes (e.g., styling of elements).
  - Listen for interactions with elements.
  - Communicate with external server.



JavaScript writing to the DOM

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Javascript in action</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
  <script>
    const root = document.getElementById("root");
    root.innerHTML = "Hello <b>World</b>";
  </script>
</html>
```

React writing the DOM

```
import { createRoot } from 'react-dom/client';

const root_elem = document.getElementById("root");
const root = createRoot(root_elem);

root.render(<>Hello <b>World</b></>);
```

From JSX to JavaScript

JSX

```
⋮
<h1 style={{color: "red"}}>

  Hello

  <Big bold>mighty</Big>

  { getName(who) }

</h1>
```

JavaScript

```
import {jsx, jsxs} from "react/jsx-runtime";

⋮
jsxs("h1",
  { style: { color: "red" },
    children: [
        "Hello",
        jsx(Big,
          { bold: true,
            children: "mighty"
          }
        ),
        getName(who)
    ]
  }
)
```

Declarative UIs in React

Remember this from session 01?

We describe **what** to render:

> **We implement code that describes the look of the user interface once, given a set of inputs (props).**
>
> **Data only flow "down" through elements.**

We don't think about which UI elements should change **when** and **how**.

We don't implement these changes **explicitly** (i.e., "when the user clicks that button, make sure to update the page here and there").

We may perceive this as if

> **React generates and renders the entire UI whenever something needs to change.**

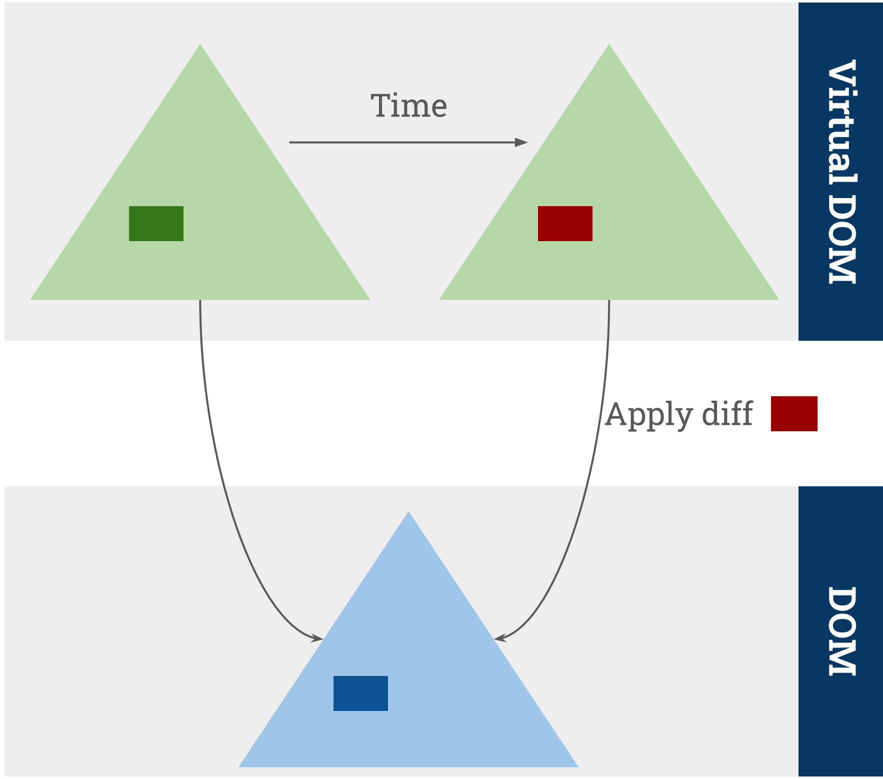
What actually happens is more clever. See next slide.

The virtual DOM in React

In React, UI elements are rendered to a (hidden) **virtual DOM.**

When React registers that parts of the UI changes, it

1. generates a new virtual DOM,
2. compares the old and new virtual DOM,
3. finds the **differences** (using a “diffing” algorithm), and
4. changes the real DOM at the places corresponding to these differences.


Time
Virtual DOM
Apply diff
DOM