CSE 331 Summer 2025

Intro to the Browser

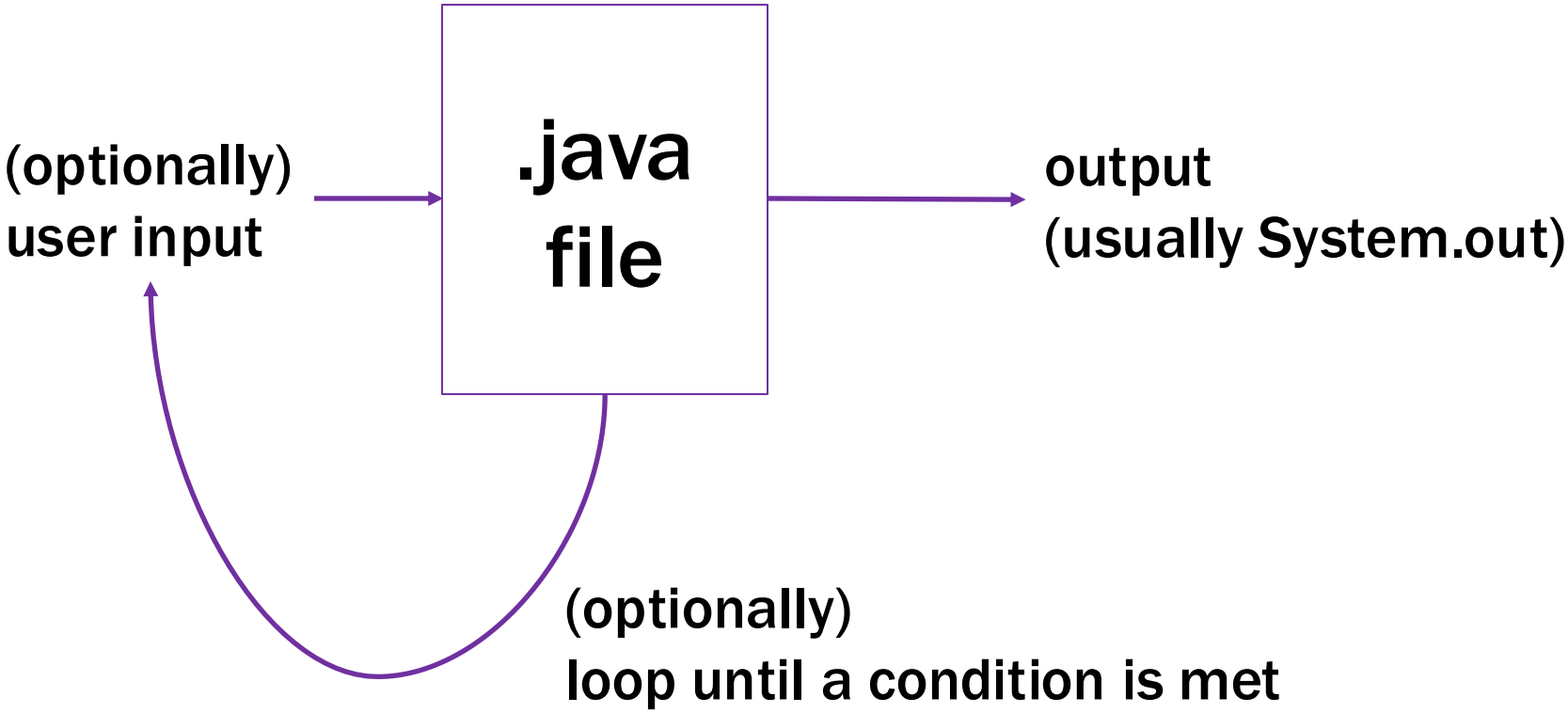
Jaela Field

June 27 Administrivia

- HW1 out!
  - HW is mostly about debugging, *not* just coding
  - This assignment got a refresh this quarter! We welcome feedback.
- advice:
  - Don't forget about Gradescope!
  - read spec carefully
  - It's <u>expected</u> that you'll have questions about JS, node, NPM, and express. Ask them!
  - start early! & take advantage of office hours!

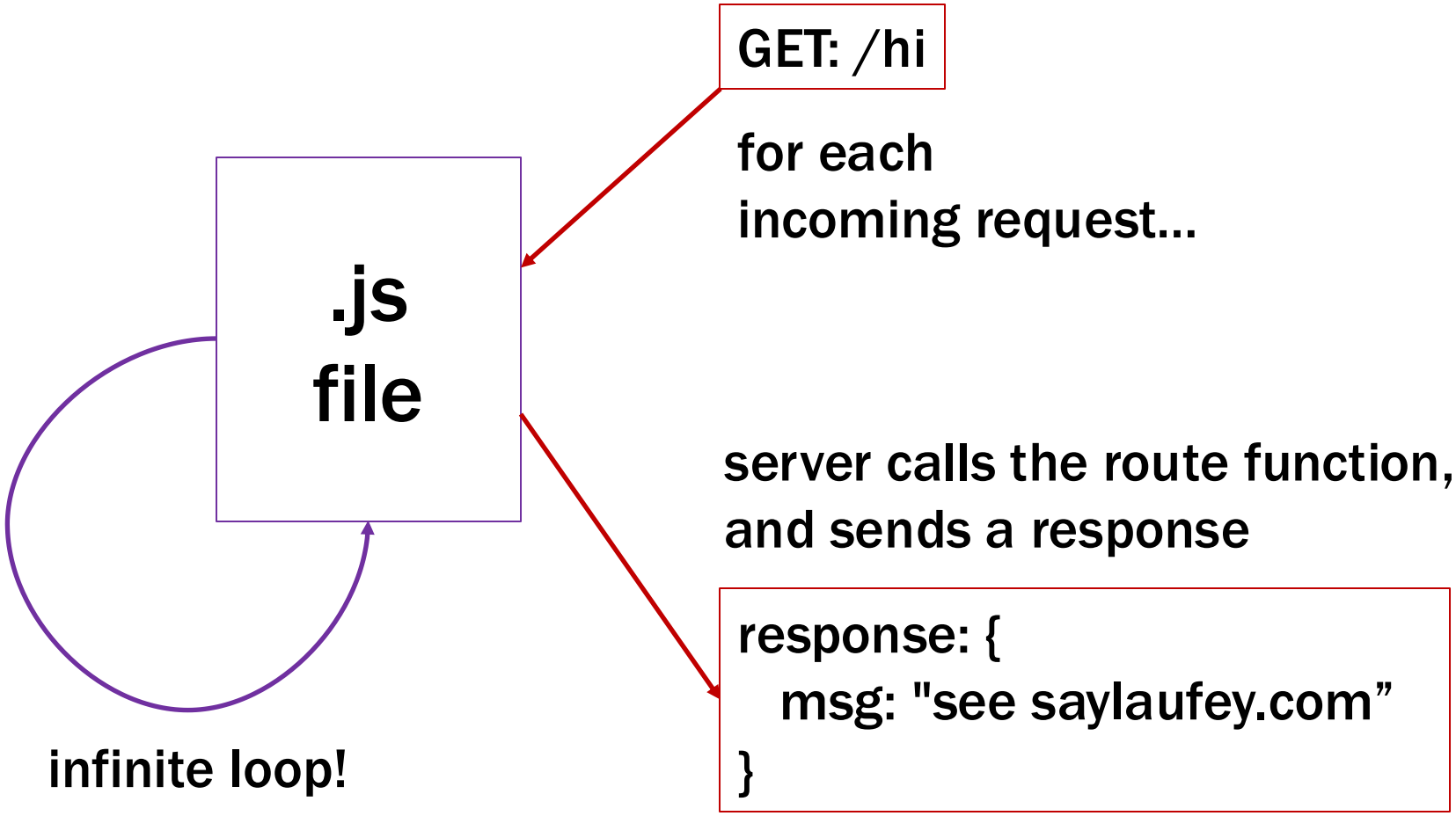
The 123 Programming Model

Run code from front-to-back, once.*



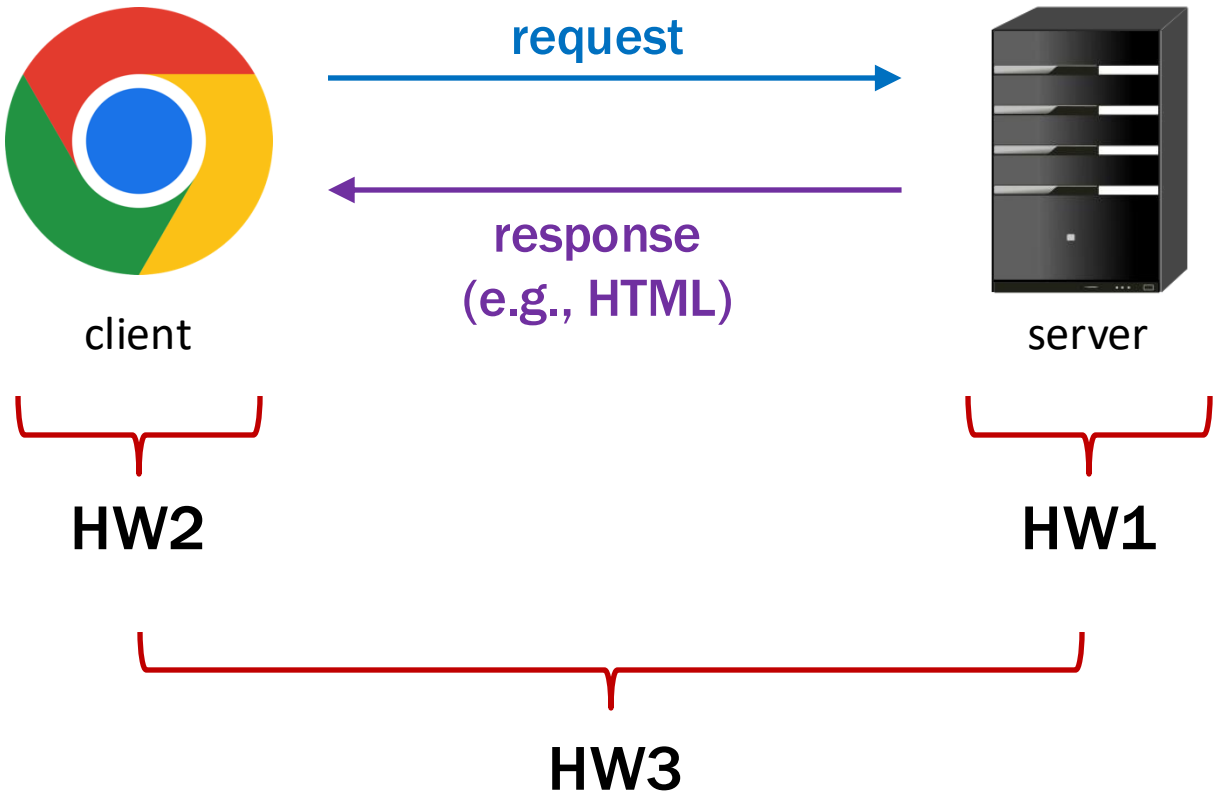
The 331 Server Programming Model

Server Code runs *forever*!



The 331 Programming Model, Zooming Out

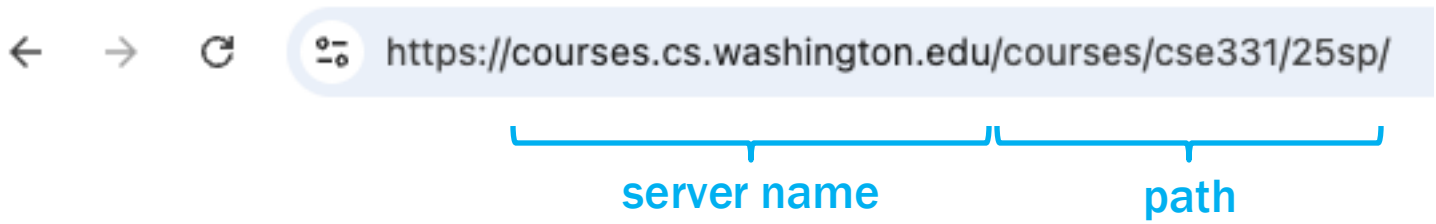
Client-Server programming has *two* programs



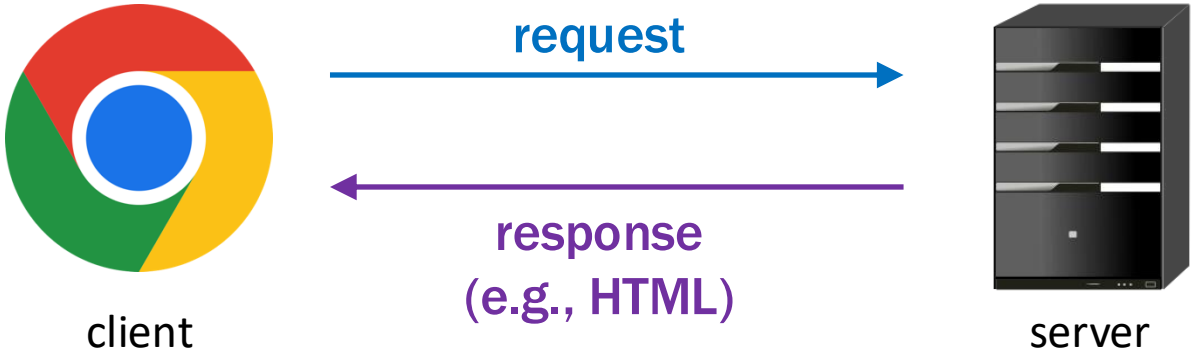
The Browser, HTML, and CSS

Recall: Browser Operation

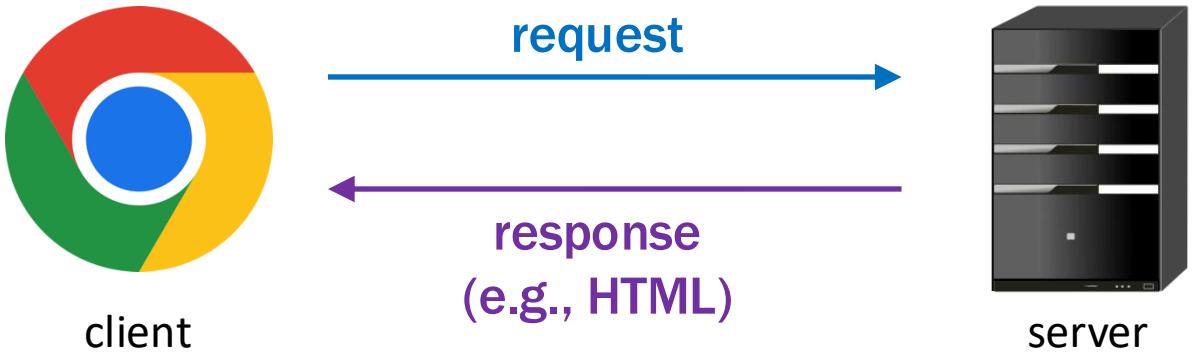
- Browser reads the URL to find what HTML to load



- Contacts the given server and asks for the given path



Browsers: JavaScript and HTML



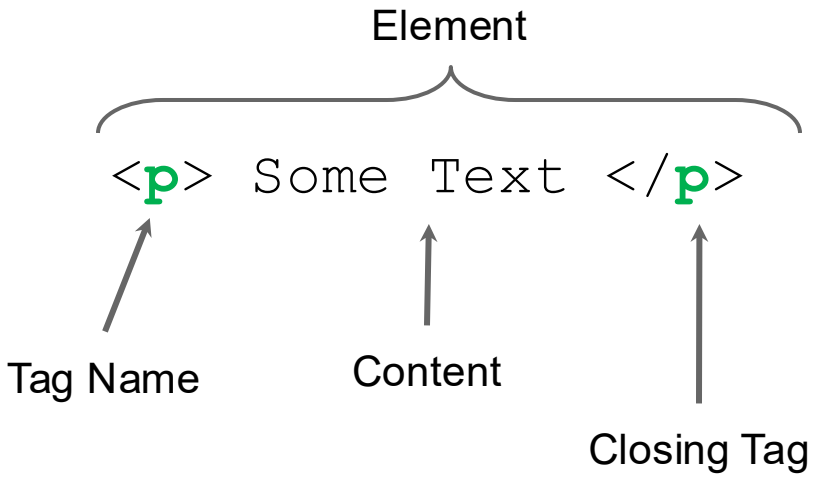
- Browser natively knows how to display HTML
- Page can also include JavaScript to execute
  - but it is not required
  - if present, the JavaScript can *change* the HTML displayed

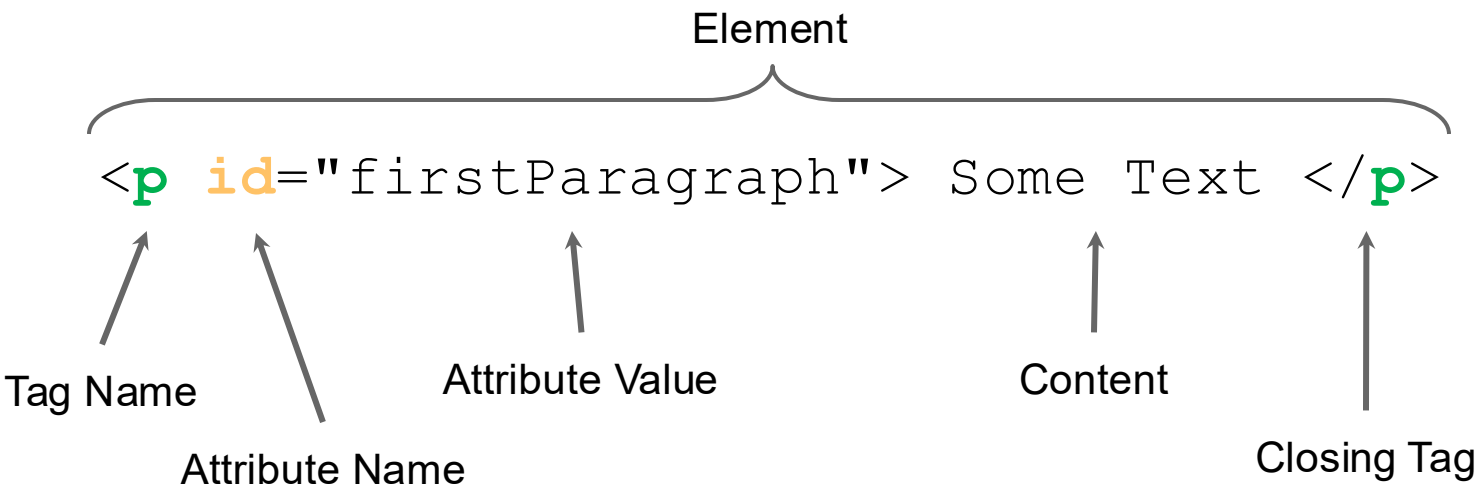
HTML

- HTML = Hyper Text Markup Language
  - text format for describing a document / UI
  - HTML describes the *structure* of the content, and (partially) what you want *drawn* in the browser

- HTML text consists primarily of “tags” and text

HTML Tags

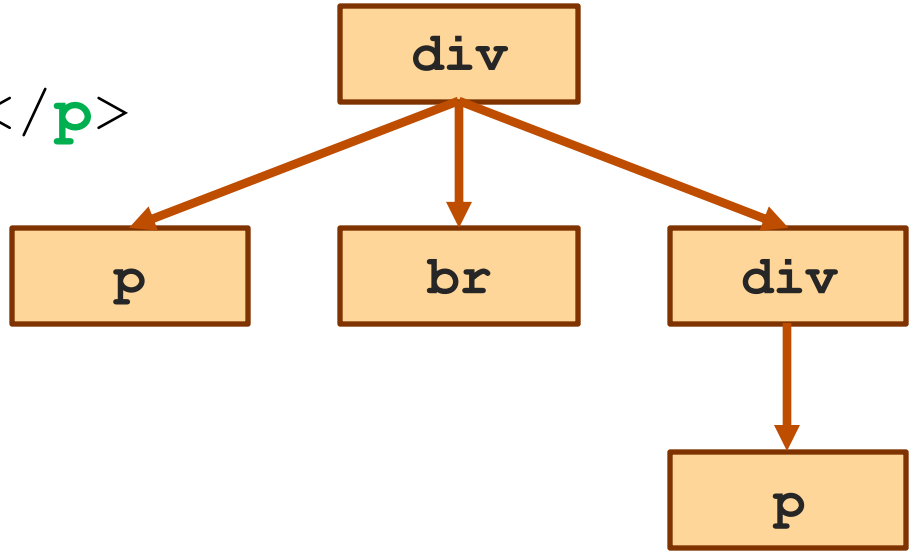

Element
 Some Text
Tag Name
Content
Closing Tag


Element
 Some Text
Tag Name
Attribute Name
Attribute Value
Content
Closing Tag

HTML as a Tree

- **Elements can have children (text or elements)**
  - **text is always a leaf in the tree**

```
<div>
    <p id="firstParagraph"> Some Text </p>
    <br>
    <div>
        <p>Hello</p>
    </div>
</div>
```

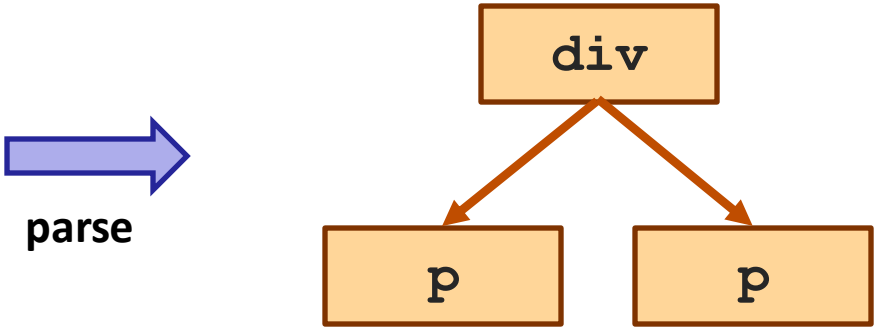


Parsing HTML

- **HTML is a text format that describes a tree**
  - **nodes are elements or text**

```
<div>
  <p>Some text</p>
  <p>More text</p>
</div>
```

HTML text



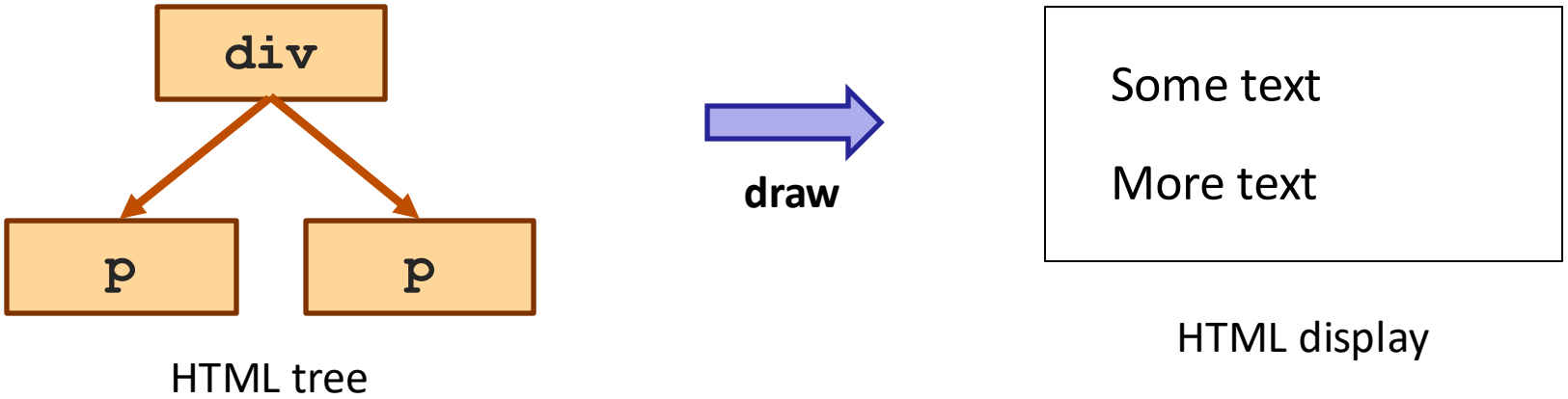
HTML tree

  - **HTML text is <u>parsed</u> into a tree ("DOM")**
  - **JS can access the tree in the variable "`document`"**

    our code lives in the world on the right side

Displaying HTML

- **Browser window displays an HTML document**
  - **tree is turned into drawing in the page**



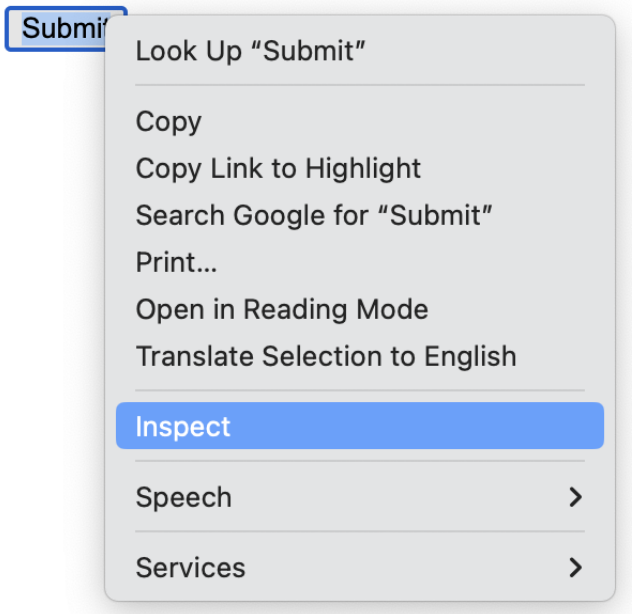
  - **browser displays the HTML in the window**

    browsers *parse* and d*raw* very quickly

  - **JS has *limited* access to display information**

Developer Tools show the HTML

- Click on any HTML element and choose "Inspect"
  - can see exact size in pixels, colors, etc.




```
▼ <body>
  ▶ <p> ··· </p>
  ▼ <div id="submit">
      <button type="button" onclick="getAnswer(event);">Submit</button>
    </div>
```

Styling

- The “style” attribute controls appearance details
  - margins, padding, width, fonts, etc.
  - see an HTML reference for details (when necessary)

- Attribute value can include many properties
  - each is “name: value”
  - separate multiple using “;”

```
<p>Hi,
  <span style="color: red; margin-left: 15px">Bob</span>!
</p>
```

Hi, Bob!

  - we will generally not worry much about looks in this class…

Cascading Style Sheets (CSS)

- Commonly used styles can be named
  - association of names to styles goes in a `.css` file

```
// foo.css
span.fancy { color: red; margin-left: 15px }

// foo.html
… <p>Hi, <span class="fancy">Bob</span></p> …
```

- Useful to avoid repetition of styling
  - makes it easier to change

Old School Web UI

Including JavaScript in HTML

- Server usually sends back HTML to the browser

- Include code to execute inside of script tag:

```
<script>
  console.log("Hi, browser");
</script>
```

- Can also put the script into another file:

```
<script src="mycode.js"></script>
```

Events in the Browser

- Client applications are event-driven
  - register "handlers" for various events

- Can do so like this in HTML (but don't!)

```
<button onClick="handleClick(event)">Click Me</button>

<script>
  const handleClick = (evt) => {
    console.log("ouch");
  };
</script>
```

Changing the HTML

- Change the HTML displayed like this (but don't!)

```
<p>Add 2 to <input type="text" id="num"></input></p>

<p><button onClick="doAdd(event)">Submit</button></p>

<div id="answer"></div>

<script>
  const doAdd = (evt) => {
    const numElem = document.getElementById("num");
    const num = Number(numElem.value);

    const ansElem = document.getElementById("answer");
    ansElem.innerHTML = `The answer is ${num+2}`;
  };
</script>
```

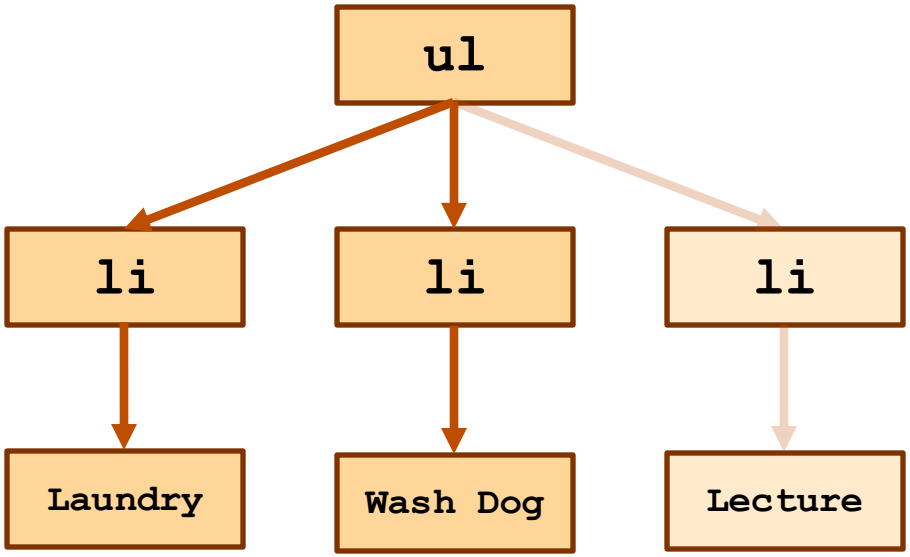
Updating the DOM: Adding Nodes

```
<h3>To-Do List</h3>

<ul id="items">
  <li>Laundry</li>
  <li>Wash Dog</li>
</ul>
```

To-Do List

- Laundry ×
- Wash dog ×

New: Lecture Add



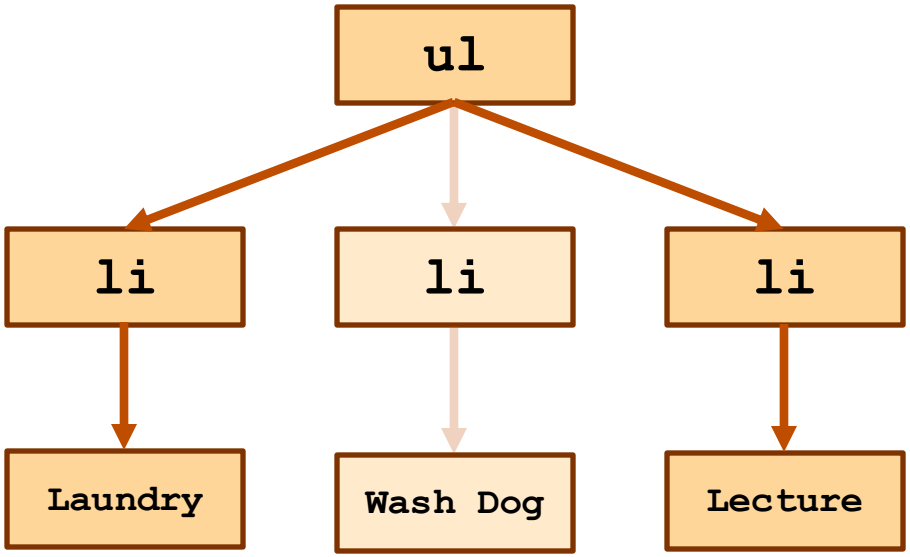
Updating the DOM: Removing Nodes

```
<h3>To-Do List</h3>

<ul id="items">
  <li>Laundry</li>
  <li>Wash Dog</li>
  <li>Lecture</li>
</ul>
```

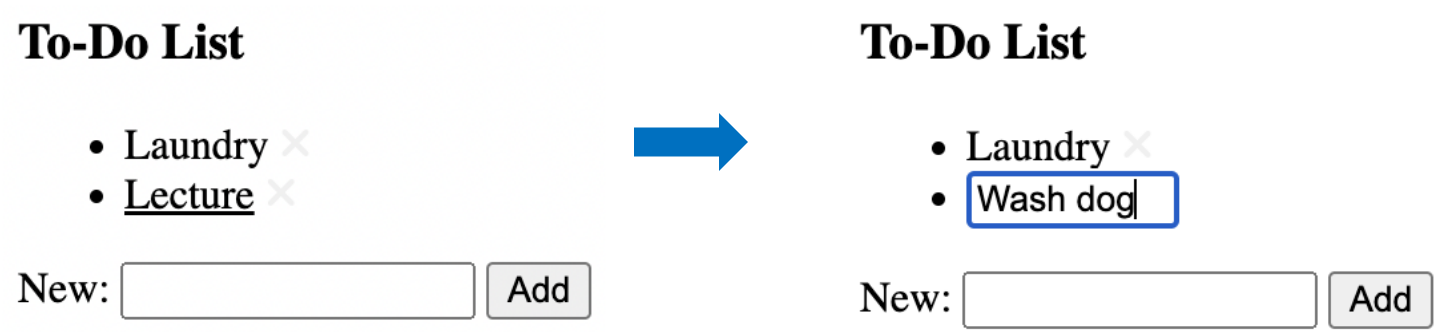
To-Do List

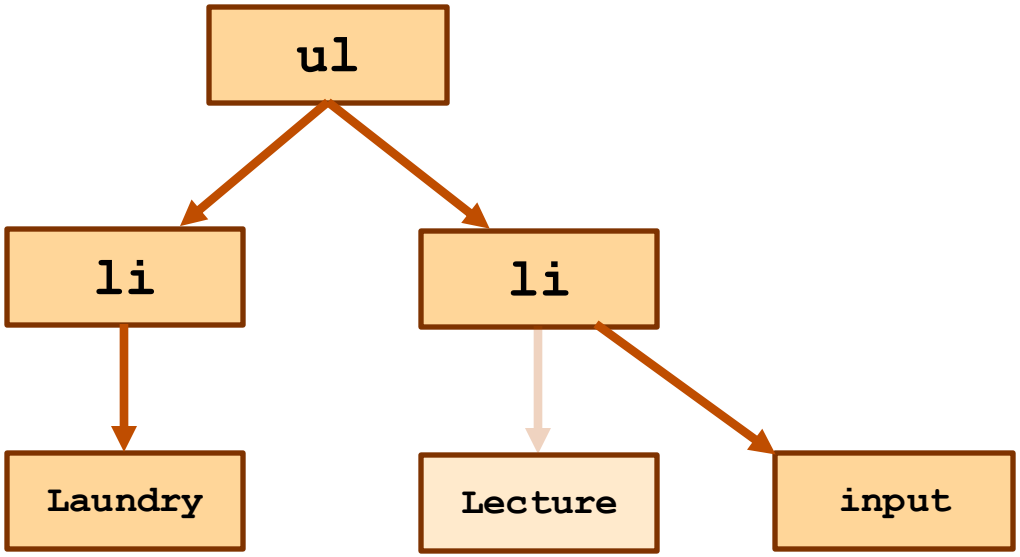
- Laundry ✕
- Wash dog ✕
- Lecture ✕

New: Add



Updating the DOM: Editing Nodes





Problems with Old School UI

- Write code for every way the UI could <u>change</u>
  - many, many cases
  - particularly tricky when working in teams/groups

- Not specific to HTML
  - same issue exists in Windows, on the iPhone, Xbox, etc.
  - if you write code to put things on screen,
    then you write code to change where they are on screen

New School UI

- New approach: what should it look like now?
  - write function that maps current state to desired HTML
  - compare desired HTML to what is on the screen now
  - make any changes needed to turn former into latter
- Huge improvement in productivity
  - introduced in Meta's "React" library
  - library performs the "compare" and "change" parts
- Faster to write HTML UI than anything else
  - many similar libraries exist for the web
  - same approach also used in mobile apps, games, ...

React

- we will use React in this class
  - goal is *not* to make you React experts
  - teach you just enough React to understand "New School UI" *ideas*
  - these ideas will apply everywhere
- similar to JS & Express, only using small subset of the library
- practical note: React is a library installed with npm

React Components

HTML Literals in JSX

- JSX: extension of JS that allows HTML expressions
  - file extension must be `.jsx`

```
const x = <p>Hi there!</p>;
```

Substitution in JSX

- Supports substitution like `` `..` `` string literals,
  - but uses `{..}` not `${..}`

```
const name = "Fred";
return <p>Hi {name}</p>;
```

- Can also substitute the value of an attribute:

```
const rows = 3;
return <textarea rows={rows} cols="25">
         initial text here
       </textarea>;
```

initial text here

JSX Gotchas

- Must have a single root tag (i.e., must be a tree)
  - e.g., cannot do this

```
return <p>one</p><p>two</p>;
```

  - instead, wrap in a `<div>` or just `<> . . </>` ("fragment")

- Replacements for attributes matching keywords
  - use "`className=`" instead of "`class=`"
  - use "`htmlFor=`" instead of "`for=`"

CSS in JSX

- CSS styling can be used in JSX

```
// foo.css
span.fancy { color: red; margin-left: 15px }

// foo.jsx
import './foo.css';  // another weird import
…
return <p>Hi, <span className="fancy">Bob</span>!</p>;
```

- Nice to get this out of the source code

Anatomy of a React Component

- split up large web pages into individual components
- React components are classes
  - class “extends” React’s Component class
  - has a constructor that takes in one argument (more on this in a moment)
  - has a field called state (that holds the app’s … data/state)
- components should have a render method
  - goal: convert app’s state to JSX (which it returns)
  - method should have be “pure” and have no “side effects”; in other words, it should not change state
  - we never call the render method – React does for us

Simplest React Component

- **Component that prints a Hello message:**

```
class HiElem extends Component {
  constructor(props) {
    super(props);

    this.state = {lang: "en"};
  }
  render = () => {
    if (this.state.lang === "es") {
      return <p>Hola, Ali!</p>;
    } else {
      return <p>Hi, Ali!</p>;
    }
  };
}
```

How do we change "`lang`"?

Simplest React Component (rendered)

Hello Ali! Español

Hola Ali! English

Changing State in our Component

```
render = () => {
  if (this.state.lang === "es") {
    return <p>Hola, Ali!
       <button onClick={this.doEngClick}>Eng</button>
      </p>;
  } else {
    return <p>Hi, Ali!
       <button onClick={this.doEspClick}>Esp</button>
      </p>;
  }
};

doEspClick = (evt) => {
  this.setState({lang: "es"};
};
```

React and Component State Changes

```
<button onClick={this.doEspClick}>Esp</button>

doEspClick = (evt) => {
  this.setState({lang: "es"};
};
```

- **Must call `setState` to change the state**
  - **directly modifying `this.state` is a (painful) bug**
- **React will automatically re-render when state changes**
  - **but this does not happen *instantly***

React Responds to `setState` calls

$$\text{HTML on screen} = \text{render}(\text{this.state})$$

	Component	React
$t = 10$	$\text{this.state} = s_1$	$\text{doc} = \text{HTML}_1 = \text{render}(s_1)$
$t = 20$	$\text{this.setState}(s_2)$	
$t = 30$	$\text{this.state} = s_2$	$\text{doc HTML}_2 = \text{render}(s_2)$

React updates this.state **to** s_2 **and** doc **to** HTML_2 ***simultaneously***

React Component with an Event Handler

- Pass method to be called as argument (a "callback"):

```
<button onClick={this.doEspClick}>Esp</button>
```

- Be careful not to do this:

```
<button onClick={this.doEspClick()}>Esp</button>
```

- Including parentheses here is a bug!
  - that would call the method inside render

    passing its return value as the value of the `onClick` attribute
  - we want to pass the method to the button, and have it called when the click occurs

Putting the UI in the Page

- **Initial page has a placeholder in the HTML:**

```
<div id="main"></div>
```

(empty DIV in `index.html`)

- **Put HTML into it from code like this:**

```
const elem = document.getElementById("main");
const root = createRoot(elem);
root.render(<HiElem />);
```

  - `createRoot` **is a function provided by the React library**

    tells React that it should keep the HTML in the page matching what render returns

Putting the UI in the Page: Props

- **Initial page has a placeholder in the HTML:**

```
<div id="main"></div>
```

(empty DIV in `index.html`)

- **Put HTML into it from code like this:**

```
const elem = document.getElementById("main");
const root = createRoot(elem);
root.render(<HiElem name={"Jaela"} size={3}/>);
```

  - **in** `HiElem`, `this.props` **will be** `{name: "Jaela", size: 3}`
  - **each component is a custom tag with its own attributes ("properties")**

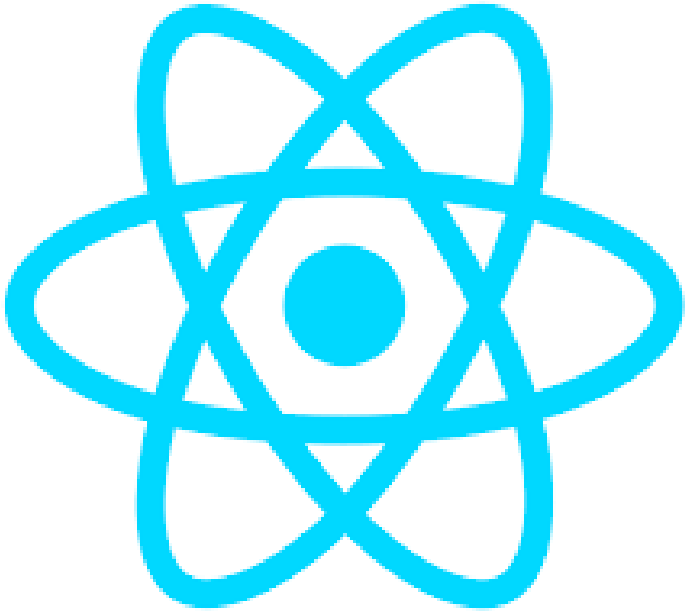
Props and State, Together

```
render = () => {
  if (this.state.lang === "es") {
    return <p>Hola, {this.props.name}!
        <button onClick={this.doEngClick}>Eng</button>
      </p>;
  …
  }
};
```

- `render` **can use both** `this.props` **and** `this.state`
  - **difference 1: caller give us props, but we set our state**
  - **difference 2: we can *change* our state**

CSE 331
Summer 2025

React



ft.   

Jaela Field

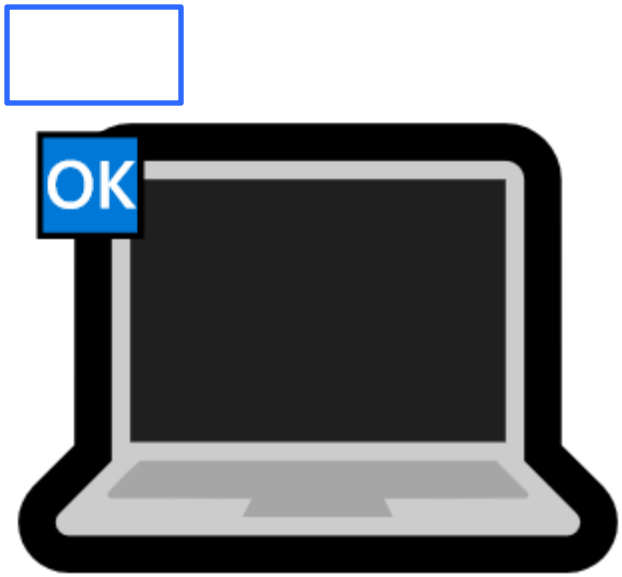
The next few slides were “drawn” live in lecture. Unfortunately, the audio didn’t work for this portion, so these slides include text that repeats what was said!

This is a recap of what we went over last time, comparing the “Old school UI” and the “New school UI,” React approach.

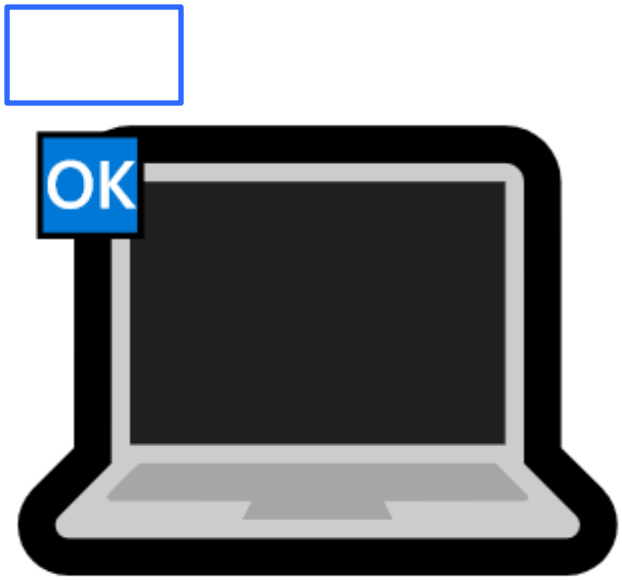
This is our user, they are looking at our app in a browser

This is our user's finger/mouse for clicking!

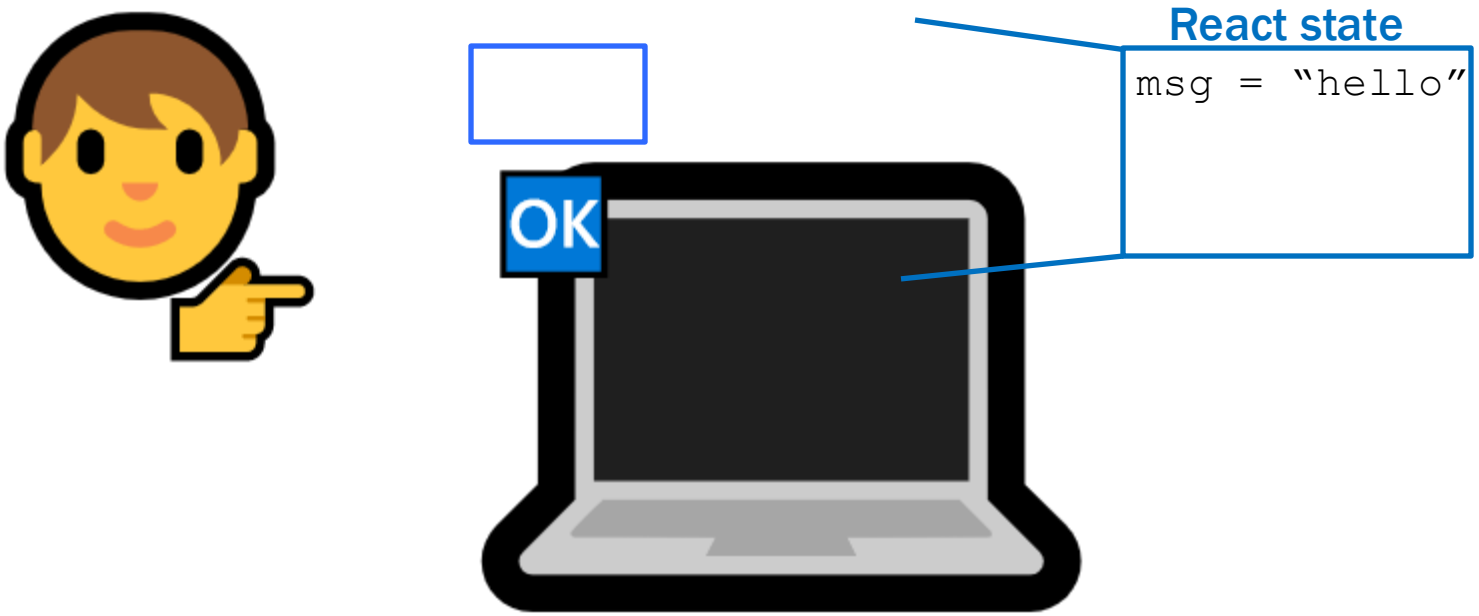
This is our example app, it has a text box (with "hello" typed in) and an "OK" button. (It's not the best app ever)



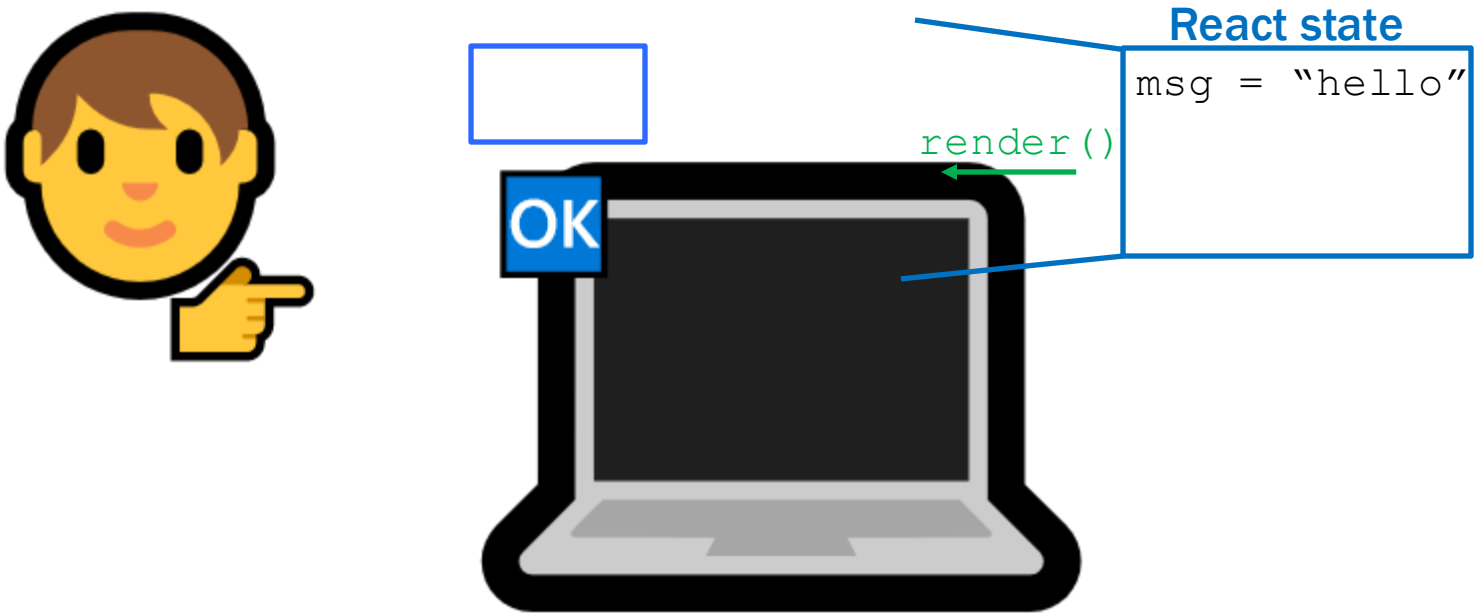
In the "old school" JS version of this app, this image essentially captures the entirety of the app.



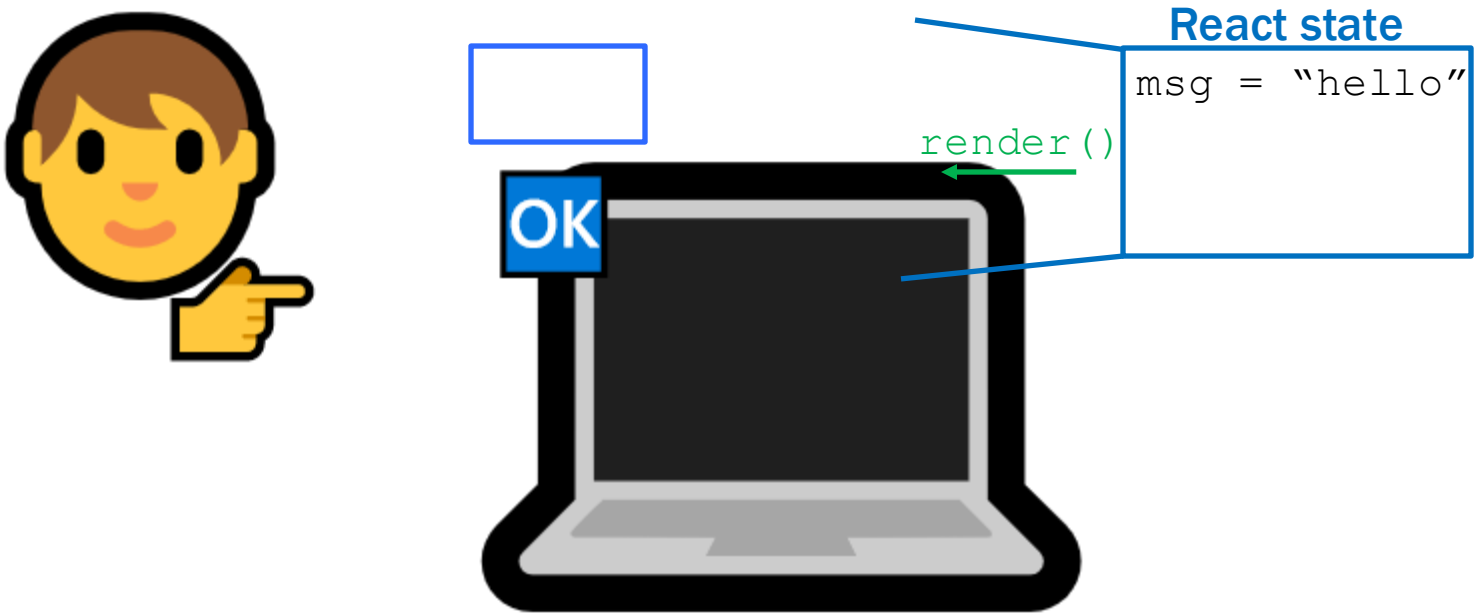
- **In old school JS, <u>All</u> of the app's state is displayed directly in the browser, "stored" within the HTML elements the user sees.**
  - If we want to access the value displayed in the text area, we track down that html element and ask "what value do you hold!!"
  - If we want to change the value displayed, we reach into the HTML on the screen and replace the contents with something new



- **In new school JS (React), there is an additional layer that sits "behind" the browser, holding the actual program state.**
  - This adds an extra layer of complexity, but the payoff is worth it, especially as apps grow

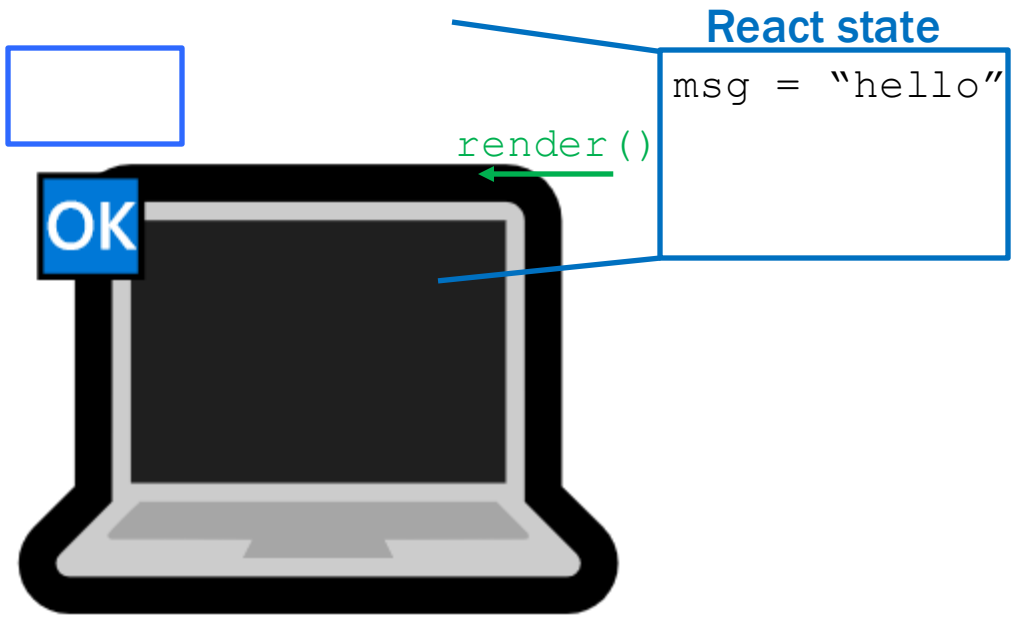


- **Key idea: `render` function**
  - Computes the HTML the user sees, given the stored state
- **`render` takes the state `msg` and creates an input element containing it**
  - `<input value={this.state.msg}></input>`

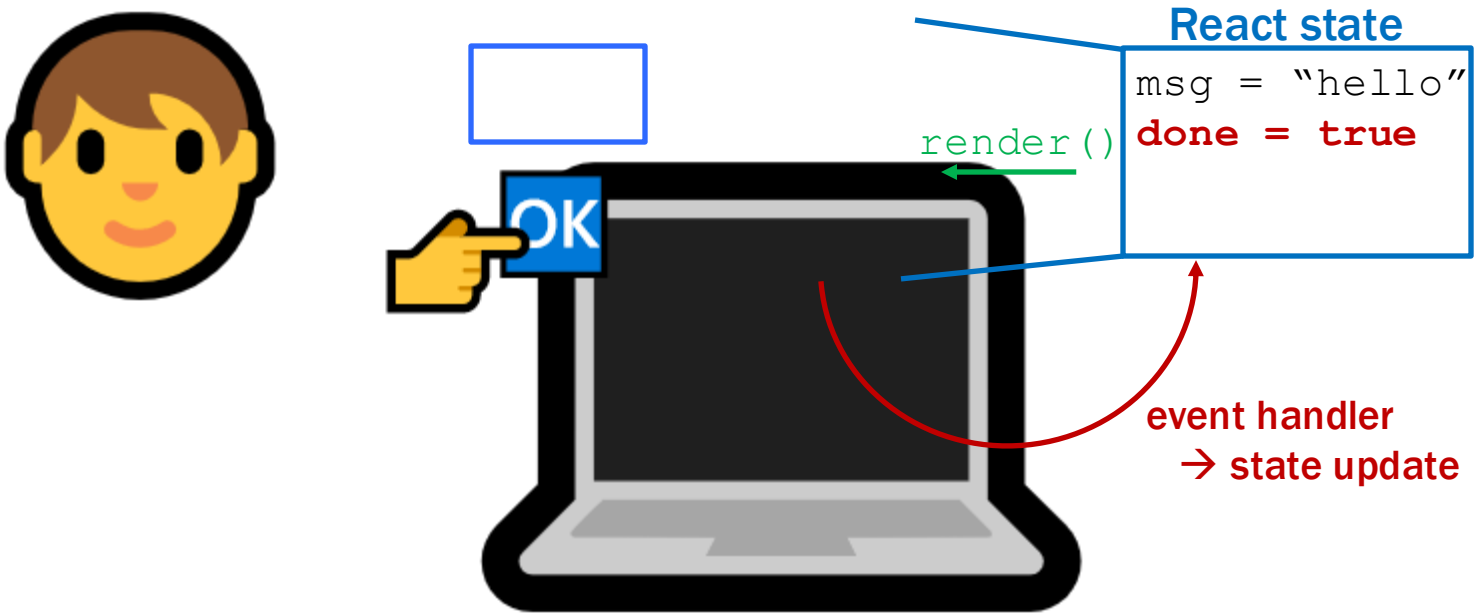


- In React, the source of truth for what the browser should displayed is contained in the state
  - VS. in "old school" JS the source of truth is exactly the html on the screen

This is our user's finger clicking the button ☺



- What happens when a user clicks OK ?
  - In "old school" JS, an event handler will run each on click (resulting in editing HTML on screen, adding HTML, etc.)



- What happens when a user clicks OK ?
  - In React, we translate the click action into a state update
  - We <u>do not</u> change the html! It is `render()`'s job to reflect change in browser

Reminder: React in Practice

- **Writing User Interface with React:**
  - **write a class that extends** `Component`
  - **implement the** `render` **method**

- **Each component becomes a new HTML tag:**

```
root.render(<HiElem name={"Jaela"}/>);
```

  - **in** `HiElem`**,** `this.props` **will be** `{name: "Jaela"}`

- **Can use** `props` **and** `state` **(and only those!) in** `render`**:**

```
render = () => {
  if (this.state.lang === "en") {
    return <p>Hi, {this.props.name}!
        <button onClick={this.doEspClick}>Esp</button>
      </p>;
  …
```

Second React Component: More User Input

- **Put name in state and let the user <u>change</u> it:**

```
class HiElem extends Component {
  constructor(props) {
    super(props);

    this.state = {name: ”Jaela"};
  }
  render = () => {
    return <p>Hi, {this.state.name}</p>;
  };
}
```

How do we change the name?

Ask the user for their name.

Second React Component: The View

What is your name? Jaela Done

Hello Jaela!

Second React Component: adding <input>

```
constructor(props) {
  super(props);
  this.state = {showGreeting: false};
}
render = () => {
  if (this.state.showGreeting) {
    return <p>Hi, {this.state.name}!</p>;
  } else {
    return <p>What is your name?
        <input type="text"></input>
        <button …>Done</button>
      </p>
  }
};
```

Second React Component: Updating State?

```
    <input type="text"></input>
    <button onClick={this.doDoneClick}>Done</button>

doDoneClick = (evt) => {
  this.setState({showGreeting: true});
  // what about "name"?
};
```

How do we get the name text?

Do not reach into document!
(Always a bug. Often a *heisenbug*.)

Text Value of Input Elememts

- **These two are different:**

```
<input type="text"></input>
```

```
<input type="text" value="abc"></input>
```

abc

  - **missing** `value` **means** `value=""`

- **The** `render` **method says what HTML should be <u>now</u>**
  - **bug if calling render would inadvertently change things**

    particularly if it would delete user data!
  - **if we want the second picture, we need to set** `value` **in** `render`

Second React Component: Input Events

```
<input type="text" value={this.state.name}
     onChange={this.doNameChange}></input>
<button onClick={this.doDoneClick}>Done</button>

doNameChange = (evt) => {
  this.setState({name: evt.target.value});
};
```

- `evt.target` **is the** `input` **element**
- `evt.target.value` **is the current text in the** `input` **element**

Second React Component: Input Event Handler

```
<input type="text" value={this.state.name}
     onChange={this.doNameChange}></input>
<button onClick={this.doDoneClick}>Done</button>

doNameChange = (evt) => {
  this.setState({name: evt.target.value});
};

doDoneClick = (evt) => {
  this.setState({showGreeting: true});
};
```

- **Never reach into the document to get state!**
  - **React can re-render at any time**
  - **will be a heisenbug when you forget (usually, it still works!)**

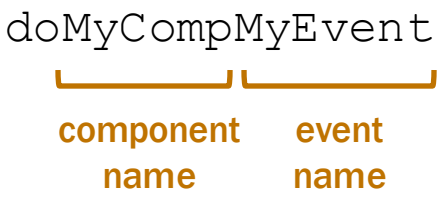
Second React Component: Mirrored State

```
<input type="text" value={this.state.name}
     onChange={this.doNameChange}></input>
<button onClick={this.doDoneClick}>Done</button>

doNameChange = (evt) => {
  this.setState({name: evt.target.value});
};

doDoneClick = (evt) => {
  this.setState({showGreeting: true});
};
```

- **Any state you need should be mirrored in your state**
  - **set** `value` **and handle** `onChange`

Event Handler Conventions

- We will use this convention for event handlers



  - e.g., `doDoneClick`, `doNewNameChange`

- Reduces the need to explain these methods
  - method name is enough to understand what it is for
  - method name is the only thing you know they read

- Components should be just rendering & event handlers

Example: To-Do List

React Payoff

- No need to write code to
  - add a new item to the HTML
  - remove an item from the HTML
  - update an item in the HTML

    all of this is code is tricky (especially if state is not mirrored properly)

- Instead, we only write:
  1. state: what does our app care about?
  2. render method: tell React what it should look like *right now*
  3. event handlers: tell React how to update state when buttons are clicked

- React figures out what to add, remove, and update

React Requirements for Lists

- **To do this, React needs more from**
  - **needs to distinguish change from add/remove**

```
<li>wash dog</li>
<li>laundry</li>
```

```
<li>wash dog</li>
<li>write lecture</li>
<li>laundry</li>
```

  - **did I insert a new item *or* change one and add another?**

    impossible to really know without more information

- **React requires each list item to have a `key=".."` property that uniquely identifies it**

React Requirements for Lists: Keys

- To do this, React needs more from
  - needs to distinguish change from add/remove

```
<li key="1">wash dog</li>
<li key="2">laundry</li>
```

```
<li key="1">wash dog</li>
<li key="3">write lecture</li>
<li key="2">laundry</li>
```

  - can now see that "2" was not changed
  - only difference is that "3" was inserted

- React will give you a warning (console) if you forget
  - will try its best to figure out what happened
  - always fix these to be safe