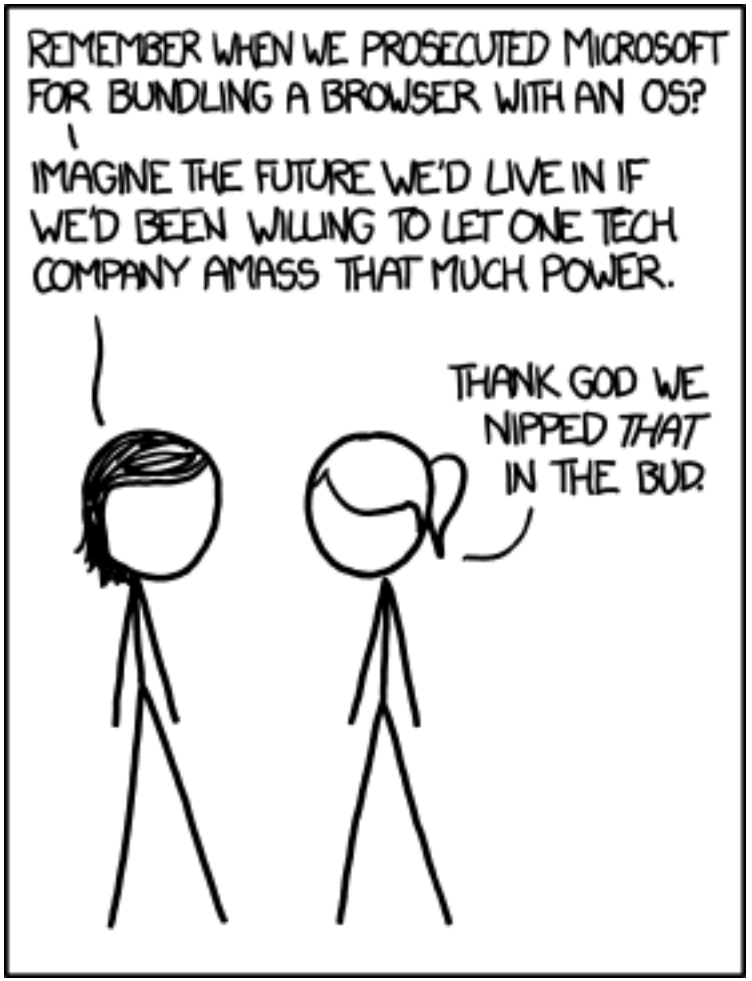
CSE 331
Spring 2025

Intro to the Browser



xkcd #1118

Matt Wang

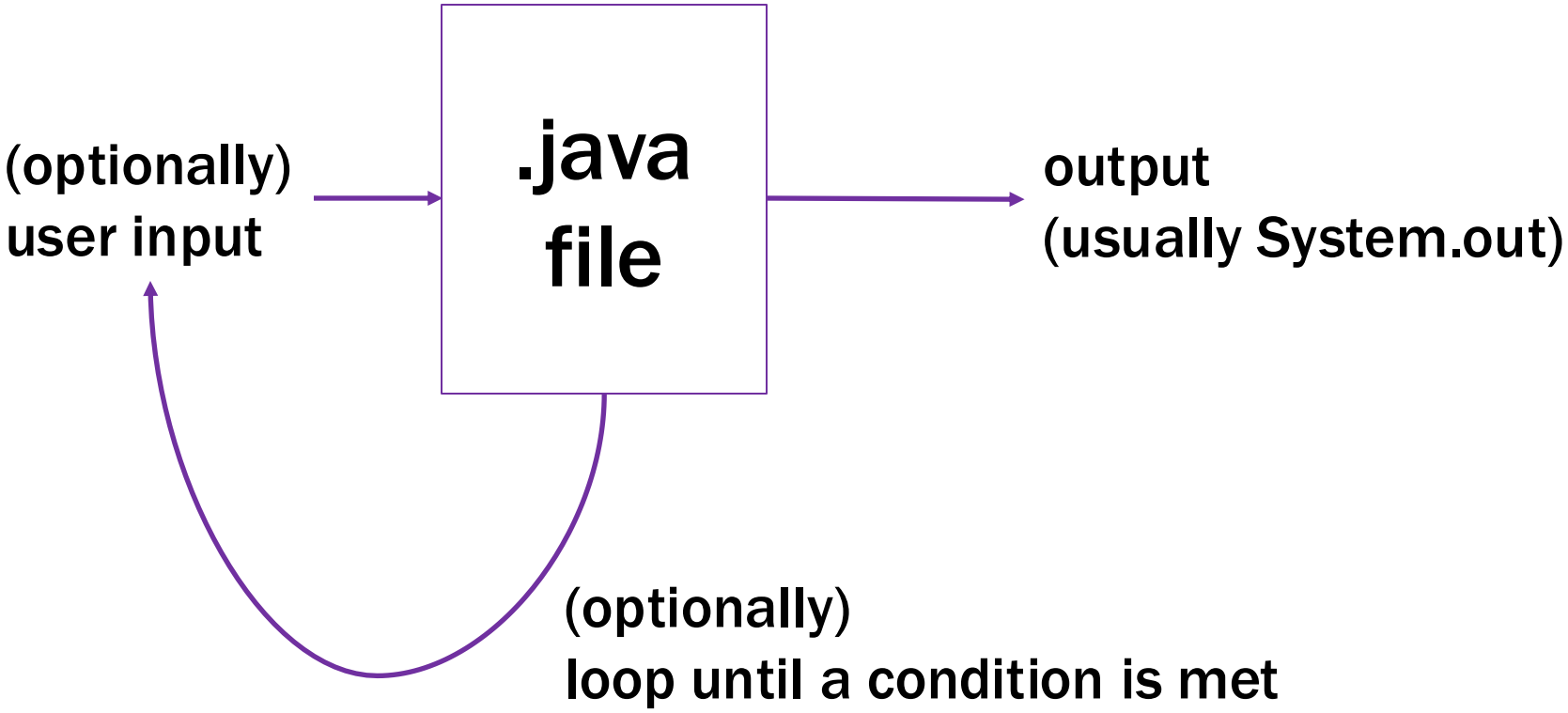
& Ali, Alice, Andrew, Anmol, Antonio, Connor, Edison, Helena, Jonathan, Katherine, Lauren, Lawrence, Mayee, Omar, Riva, Saan, and Yusong

Notes on HW1

- HW1 out!
  - HW is mostly about debugging, *not* just coding
  - so: we are mostly assessing you on debugging, and *not really* on correctness (see: spec!)
  - but, you do need to attempt *all* functions
- advice:
  - read spec carefully – app is complex, but you are implementing a small subset
  - you probably have questions about JS, node, NPM, and express. That's expected – ask them!
  - start early! & take advantage of office hours!

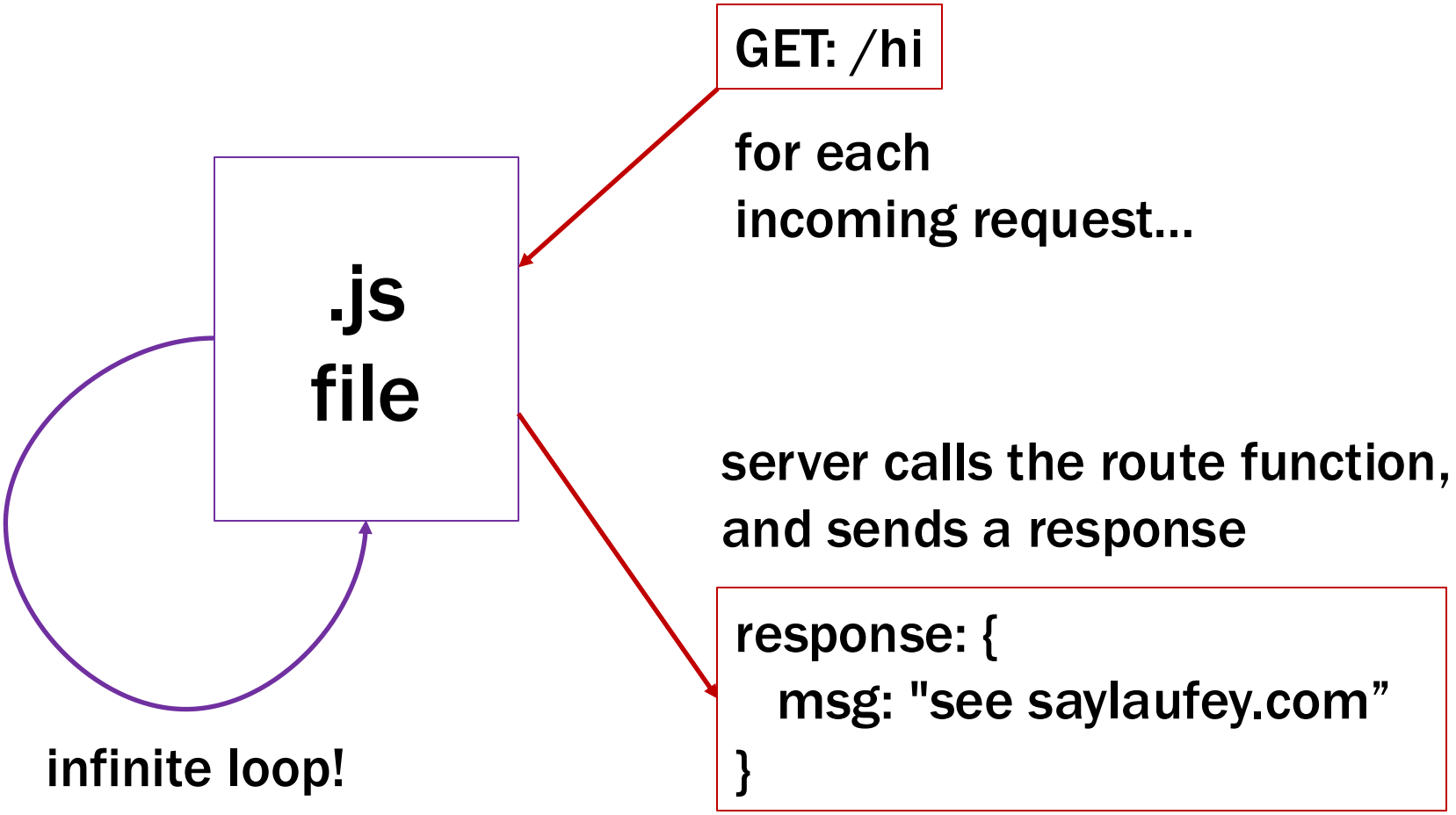
The 123 Programming Model

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



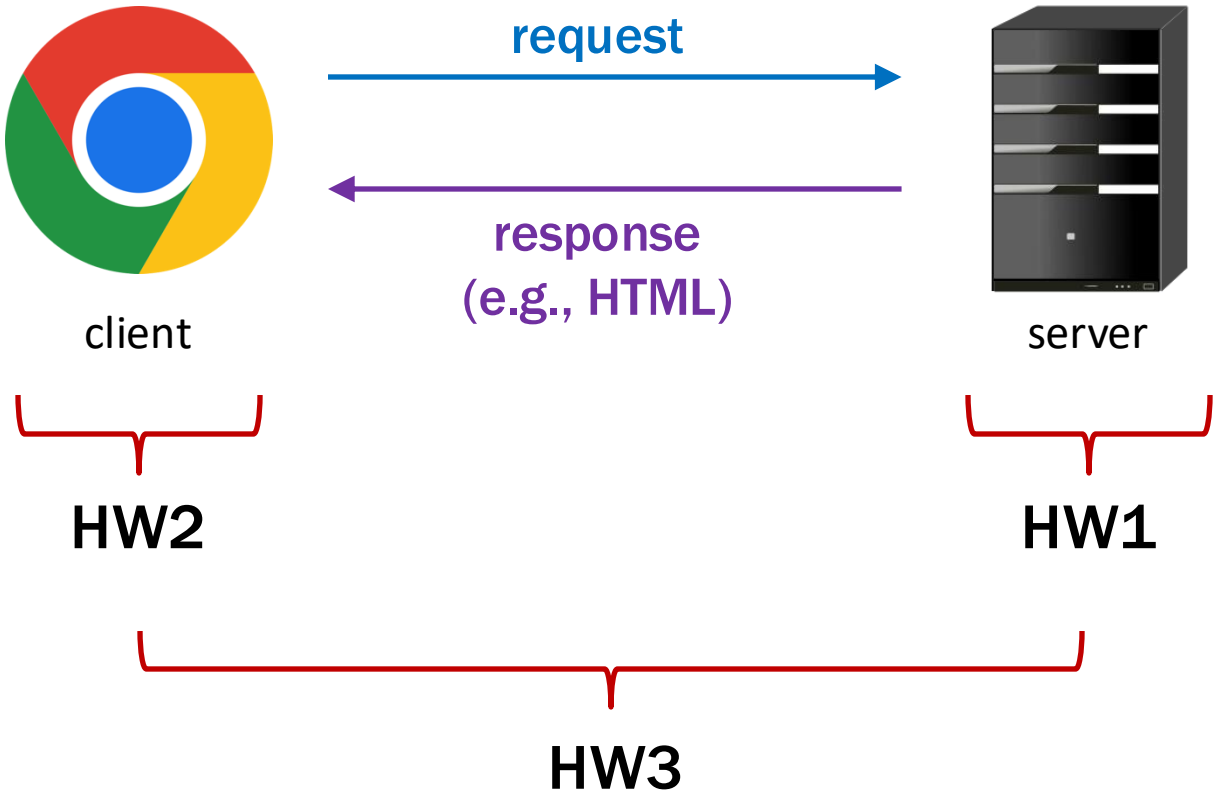
The 331 Server Programming Model

Server Code runs *forever*!



The 331 Programming Model, Zooming Out

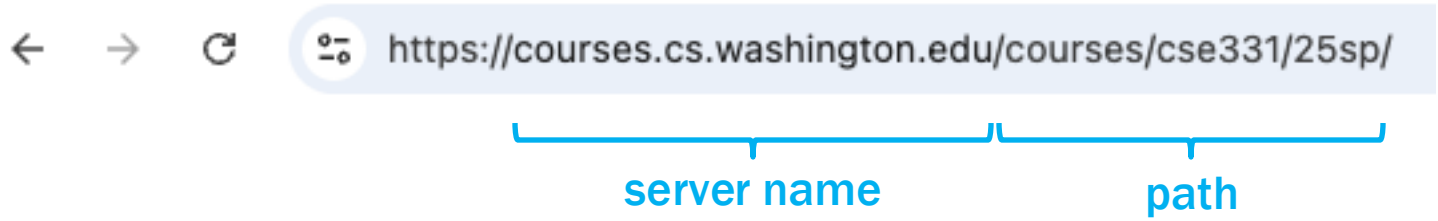
Client-Server programming has *two* programs



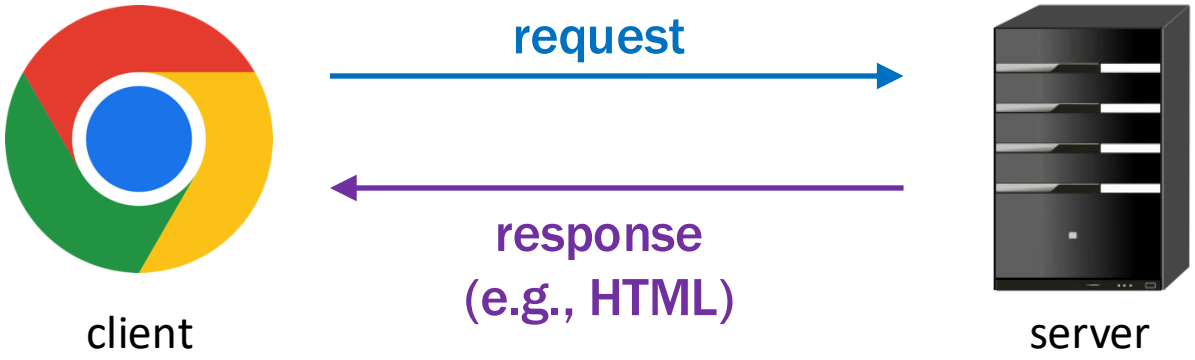
The Browser, HTML, and CSS

Recall: Browser Operation

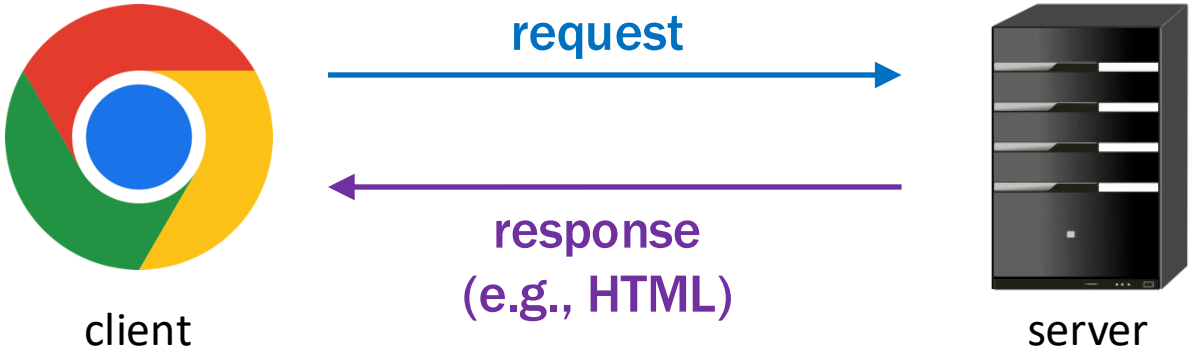
- Browser reads the URL to find what HTML to load



- Contacts the given server and asks for the given path



Browsers: JavaScript and HTML



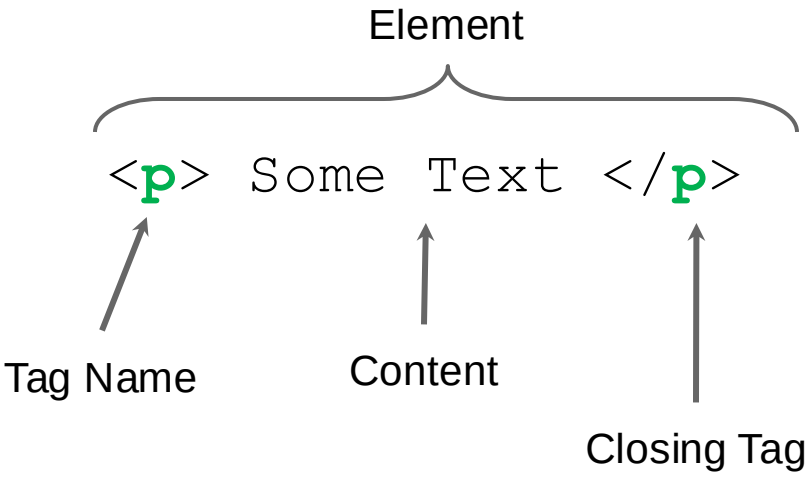
- Browser natively knows how to display HTML
- Page can also include JavaScript to execute
  - but it is not required
  - if present, the JavaScript can *change* the HTML displayed

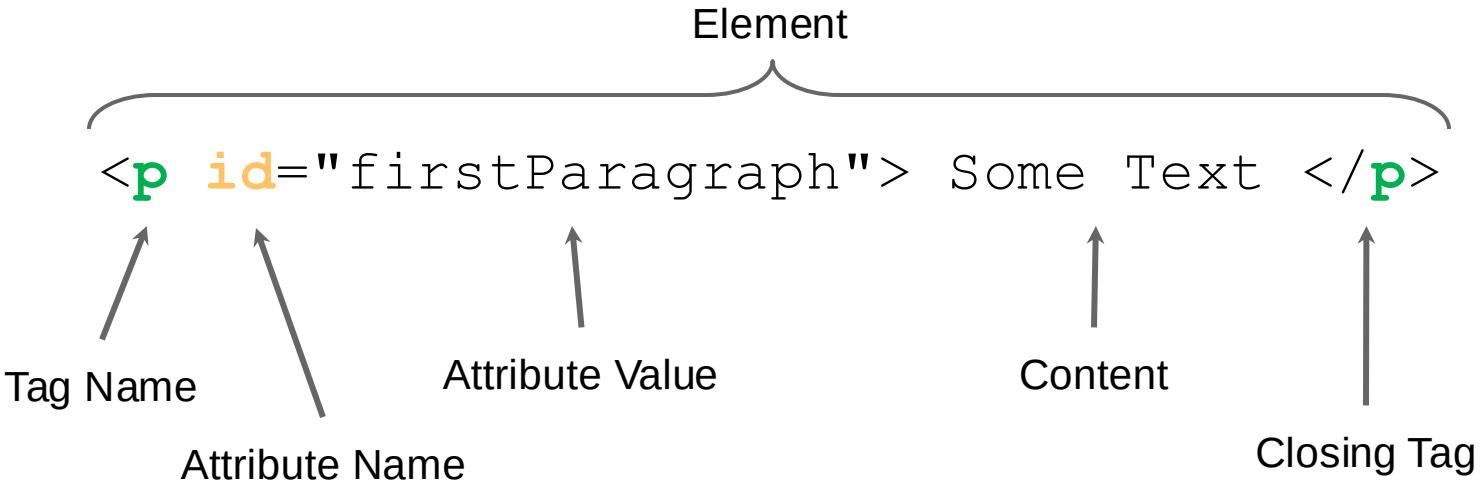
HTML

- 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

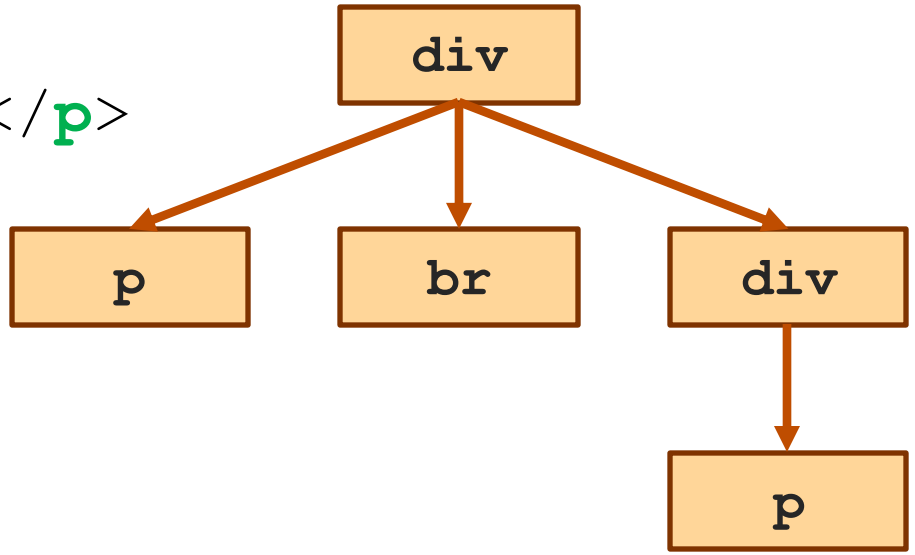

Element
 Some Text
Tag Name
Content
Closing Tag


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>
```

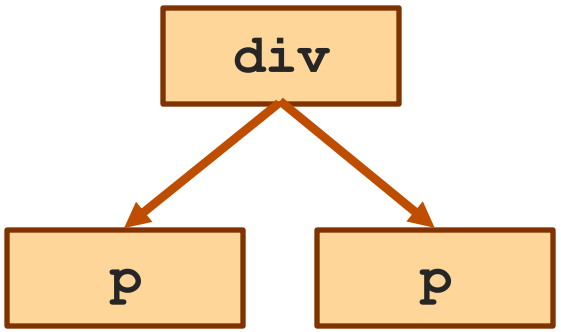


Parsing HTML

- **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



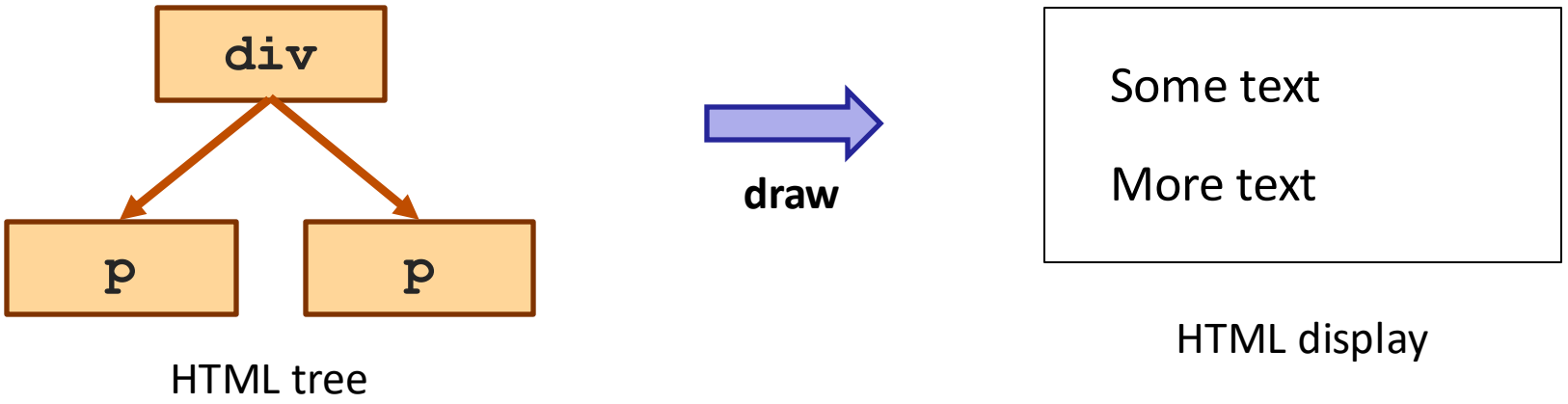
HTML tree

  - **HTML text is parsed 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**



HTML tree

HTML display

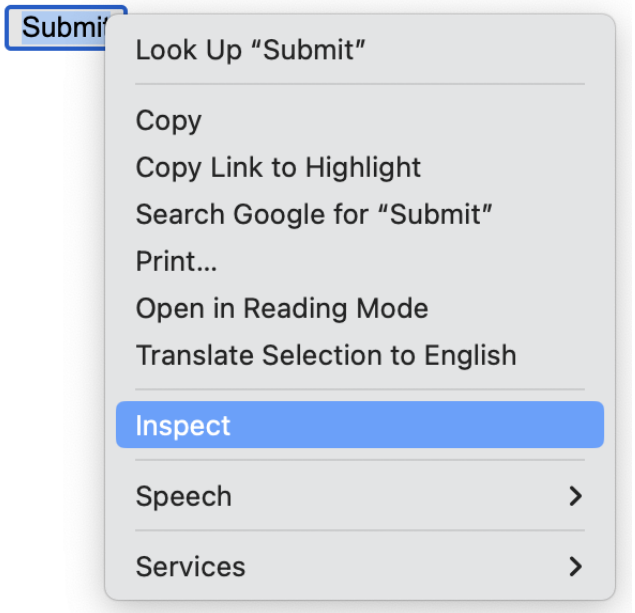
  - **browser displays the HTML in the window**

    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>
```

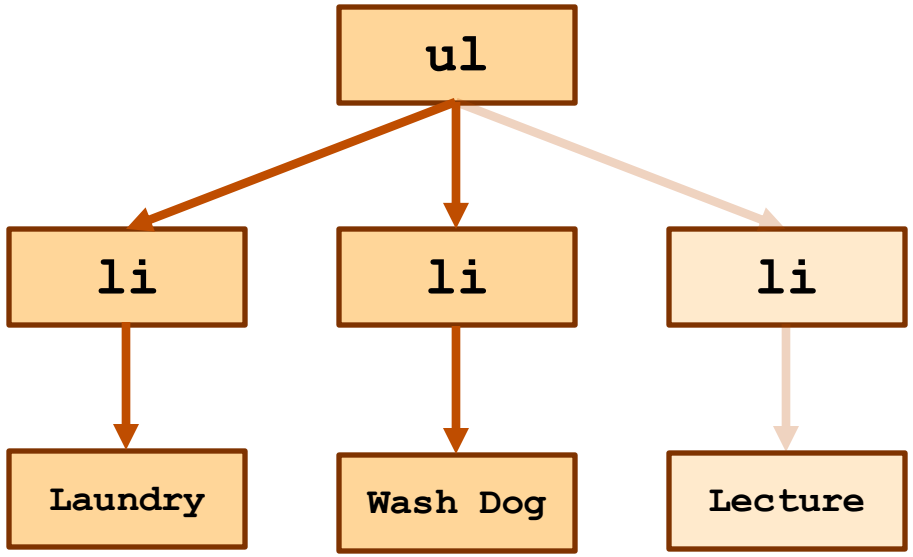
Updating the DOM: Adding Nodes

```
<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



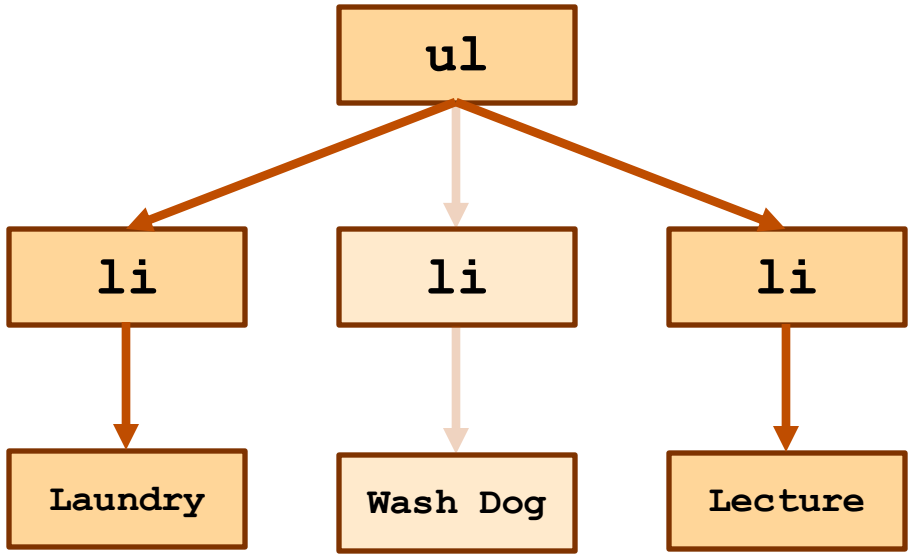
Updating the DOM: Removing Nodes

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

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

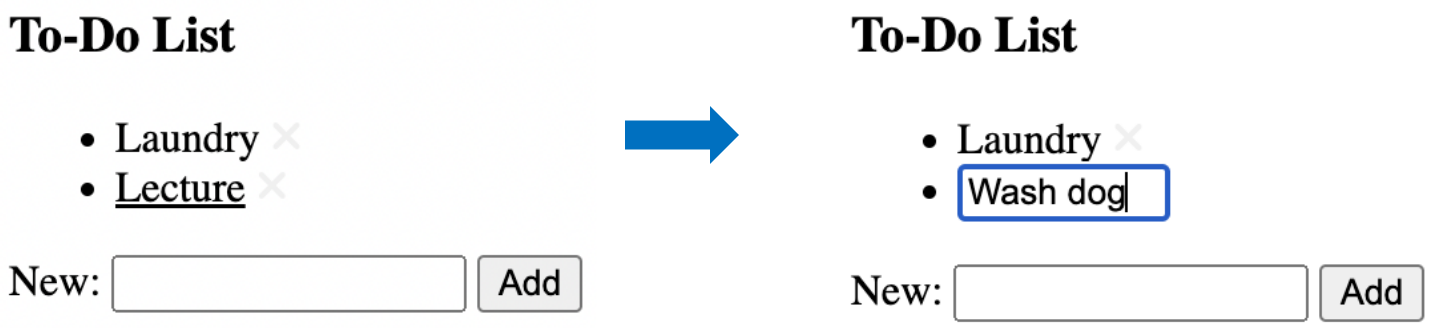
To-Do List

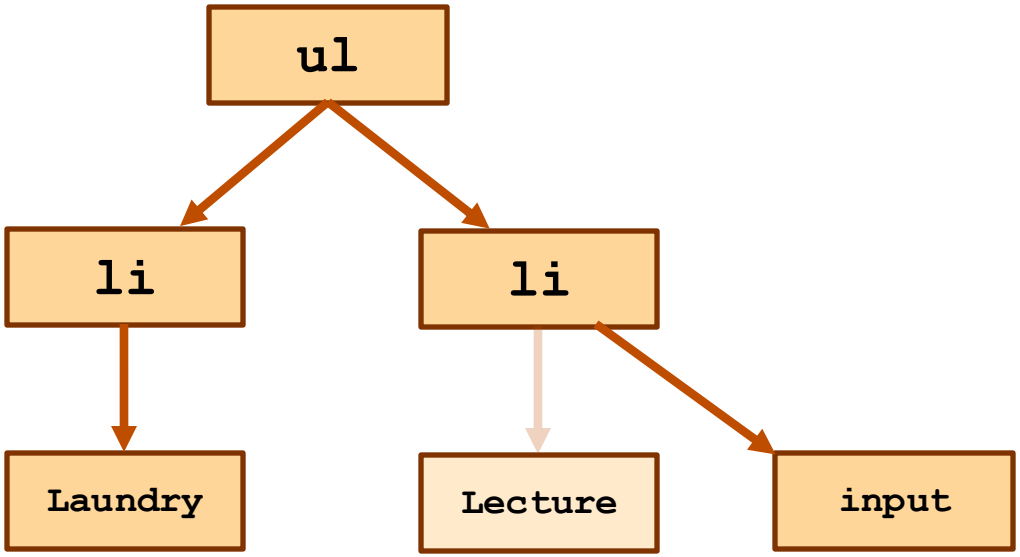
- Laundry ✕
- Wash dog ✕
- Lecture ✕

New: Add



Updating the DOM: Editing Nodes





Problems with Old School UI

- Write code for every way the UI could change
  - 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>;
```

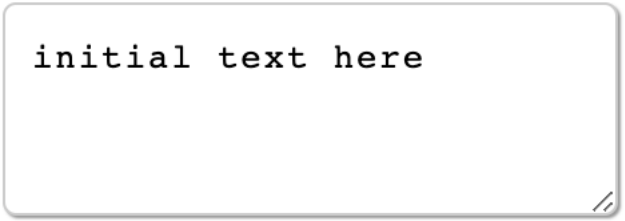
Substitution in JSX

- 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>;
```



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 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

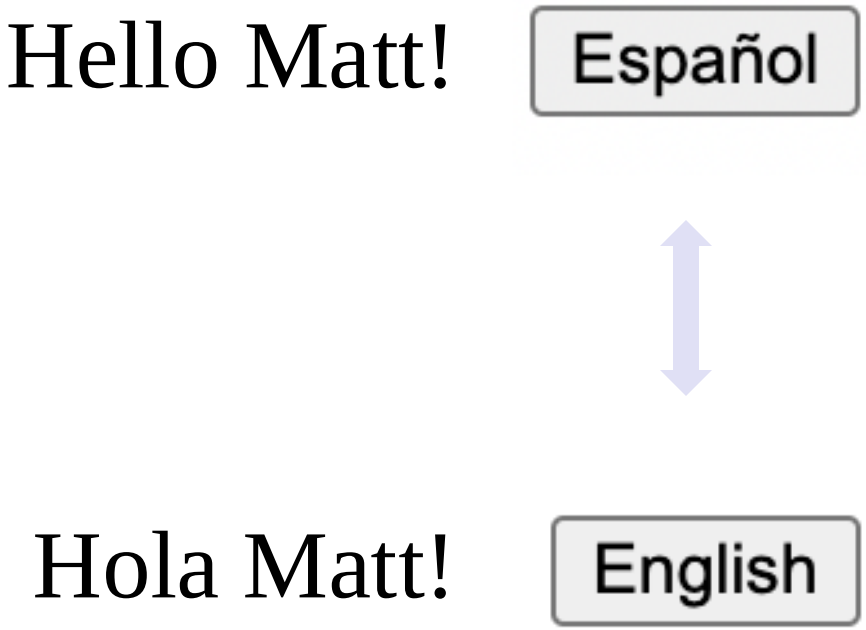
Simplest React Component

- **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, Matt!</p>;
    } else {
      return <p>Hi, Matt!</p>;
    }
  };
}
```

How do we change "`lang`"?

Simplest React Component (rendered)



Changing State in our Component

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

HTML on screen = render(this.state)

	Component	React
t = 10	this.state = s_1	doc = $HTML_1$ = render(s_1)
t = 20	this.setState(s_2)	
t = 30	this.state = s_2	doc $HTML_2$ = 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={"Matt"} size={3}/>);
```

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

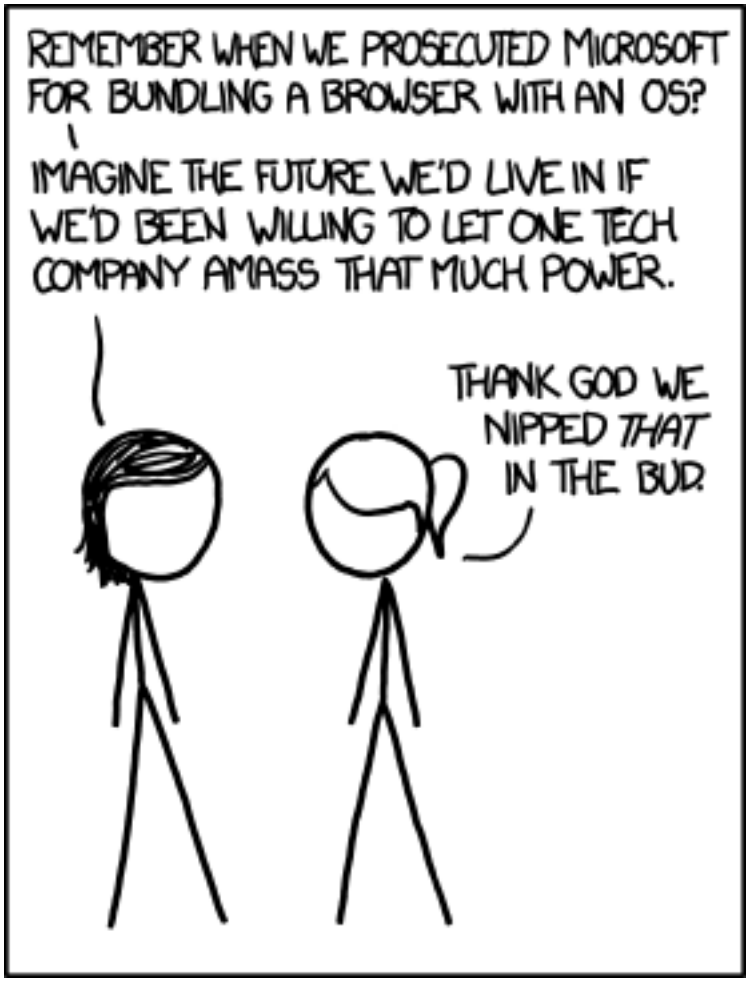
Props and State, Together

```
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 Spring 2025

More React

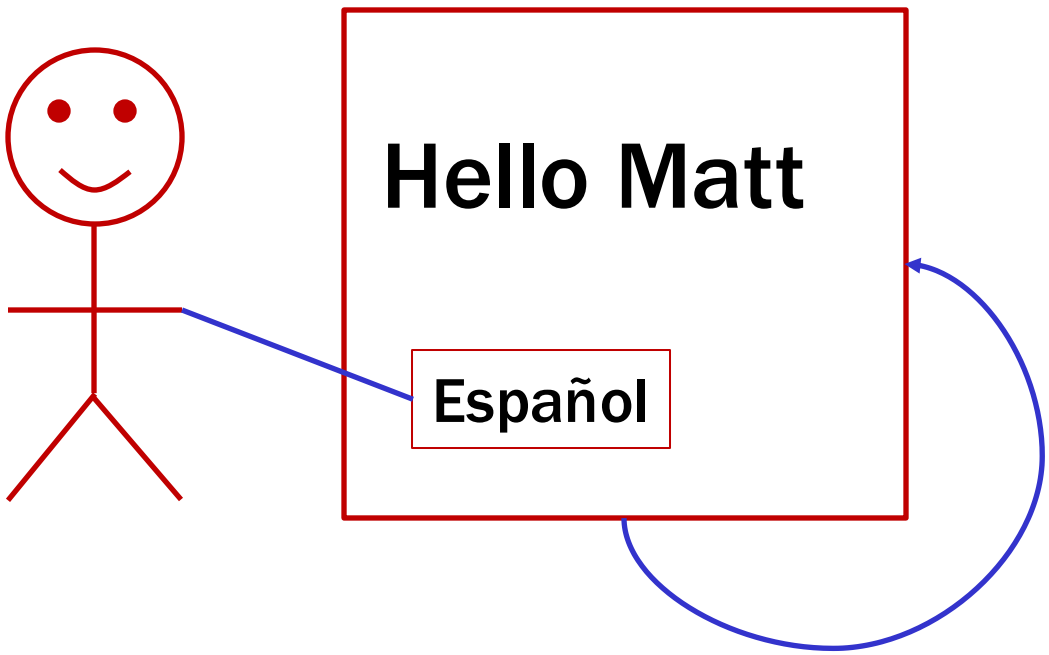


xkcd #1118

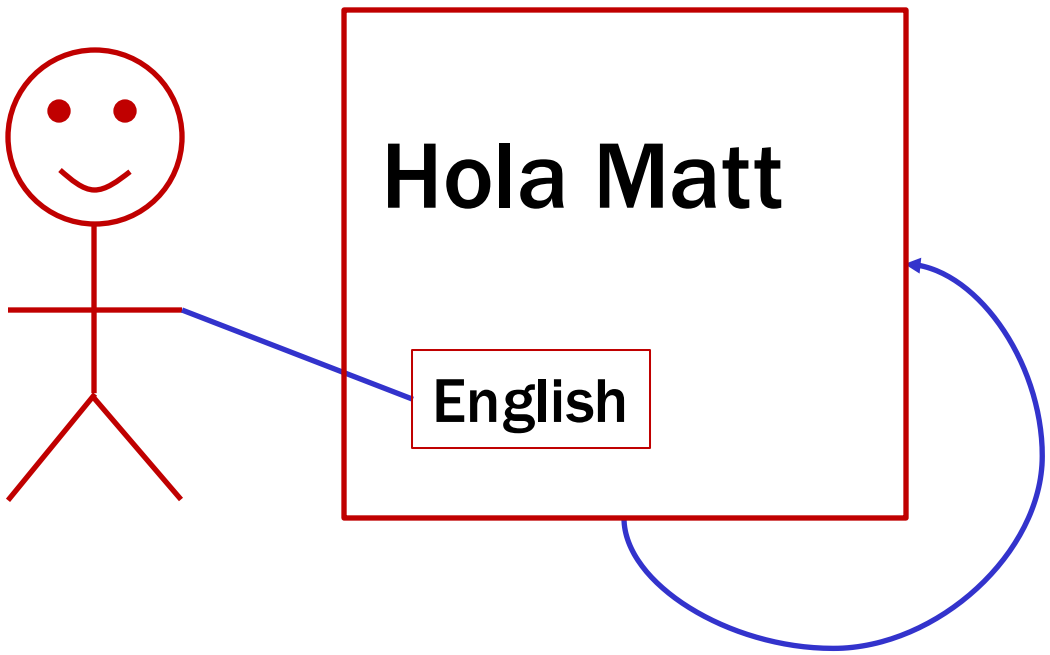
Matt Wang

& Ali, Alice, Andrew, Anmol, Antonio, Connor, Edison, Helena, Jonathan, Katherine, Lauren, Lawrence, Mayee, Omar, Riva, Saan, and Yusong

The “Old School UI” model (1/2)



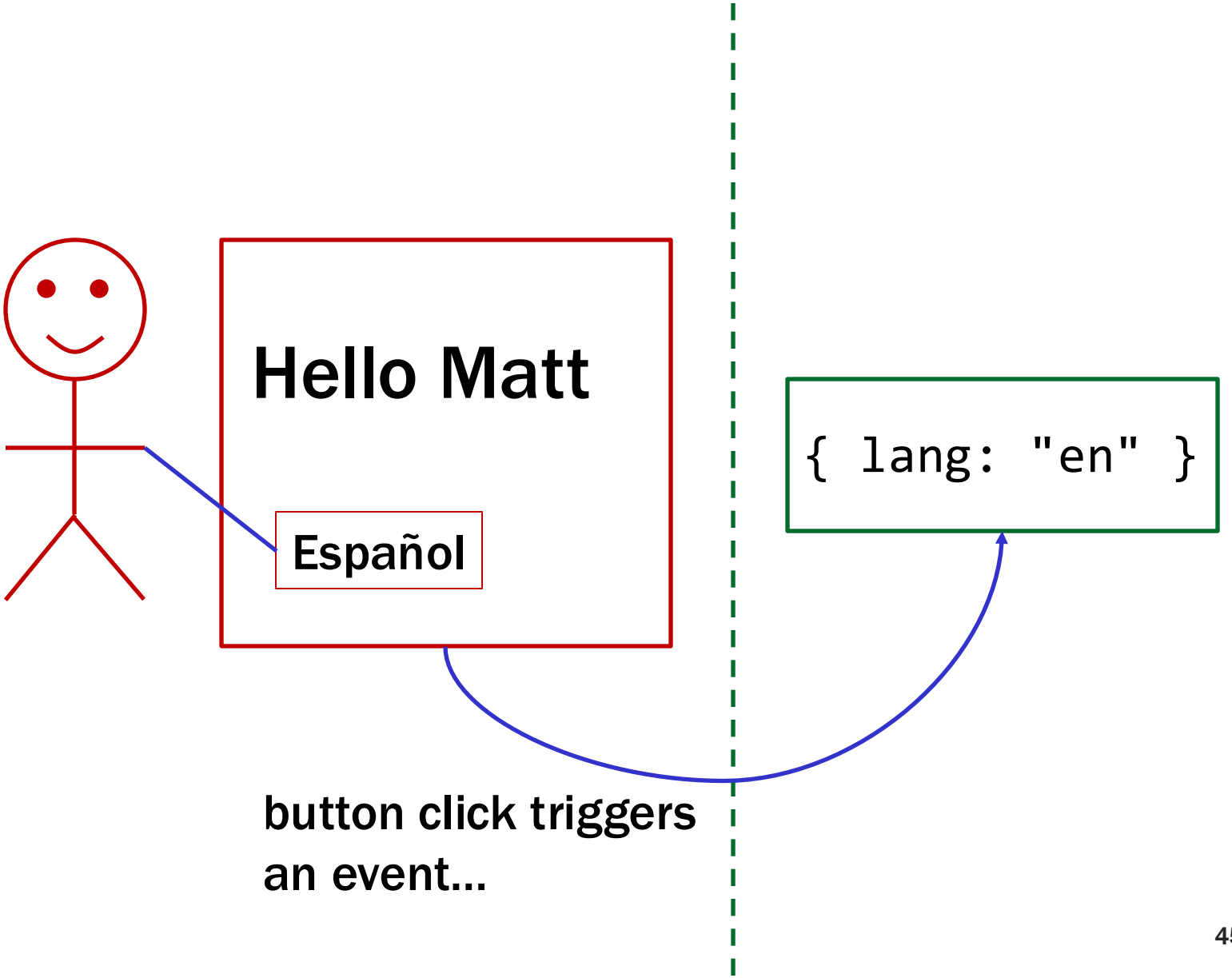
The “Old School UI” model (2/2)



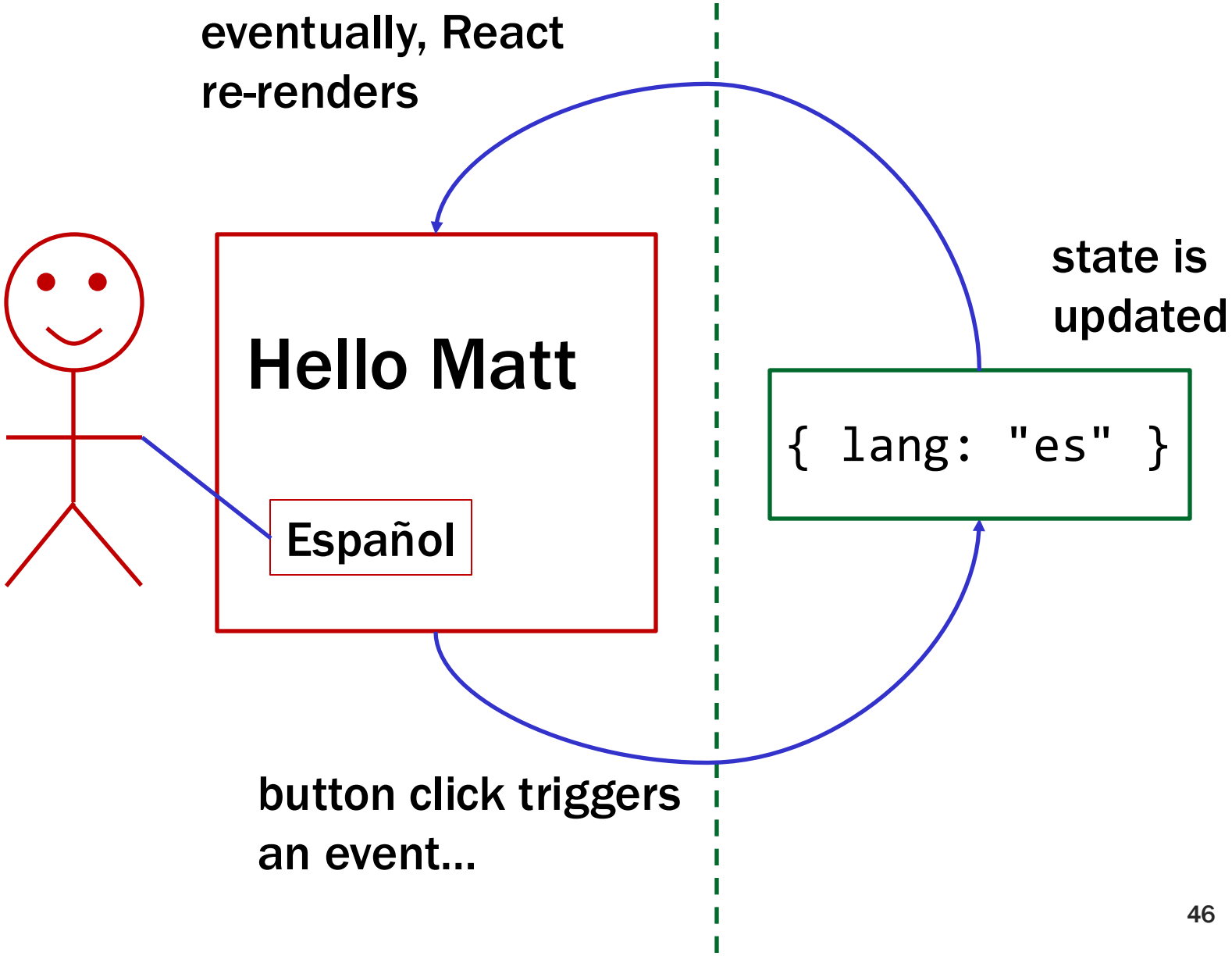
State is stored *in* the DOM / HTML.

Works for small examples, breaks with complex apps

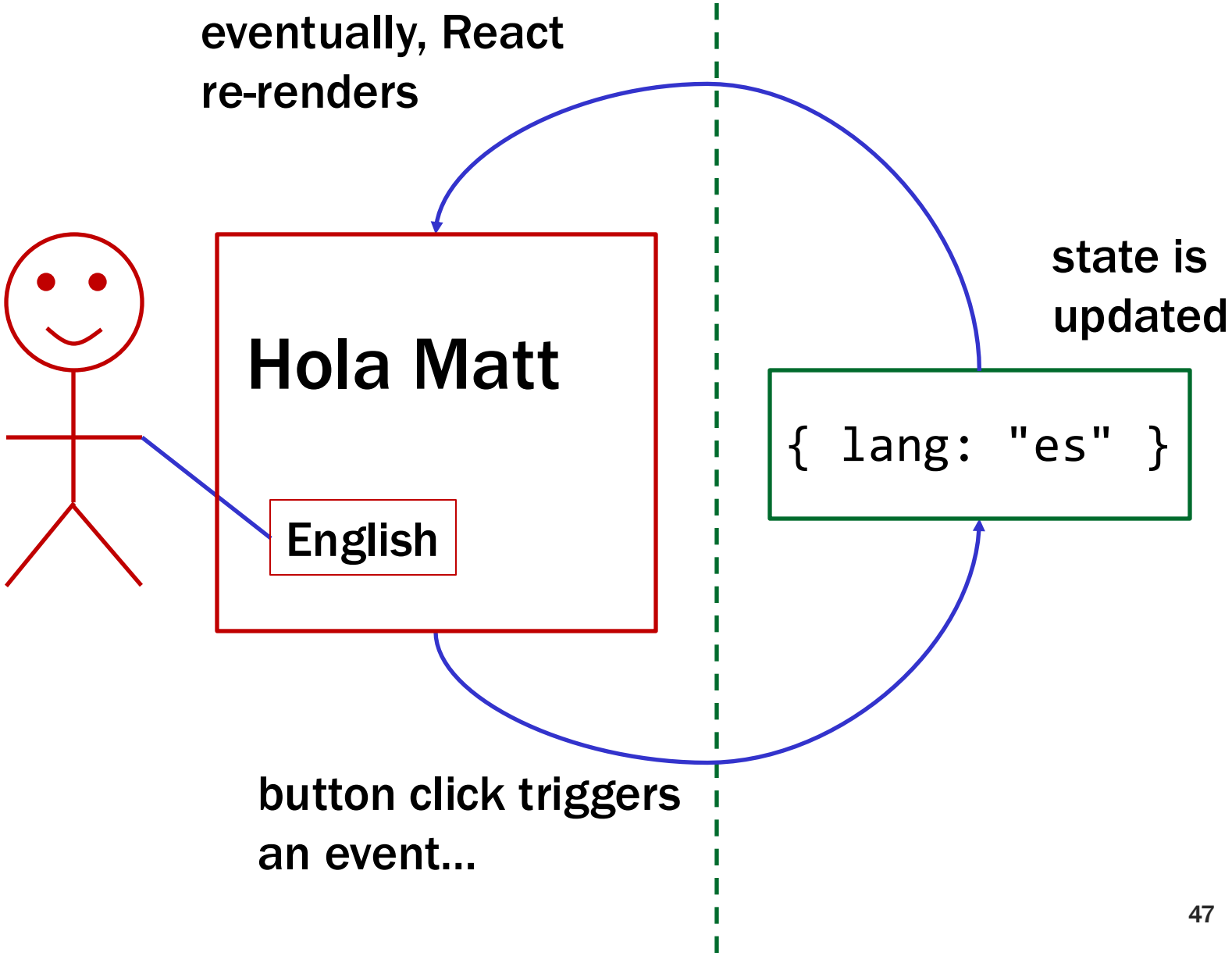
The React Model (1/3)



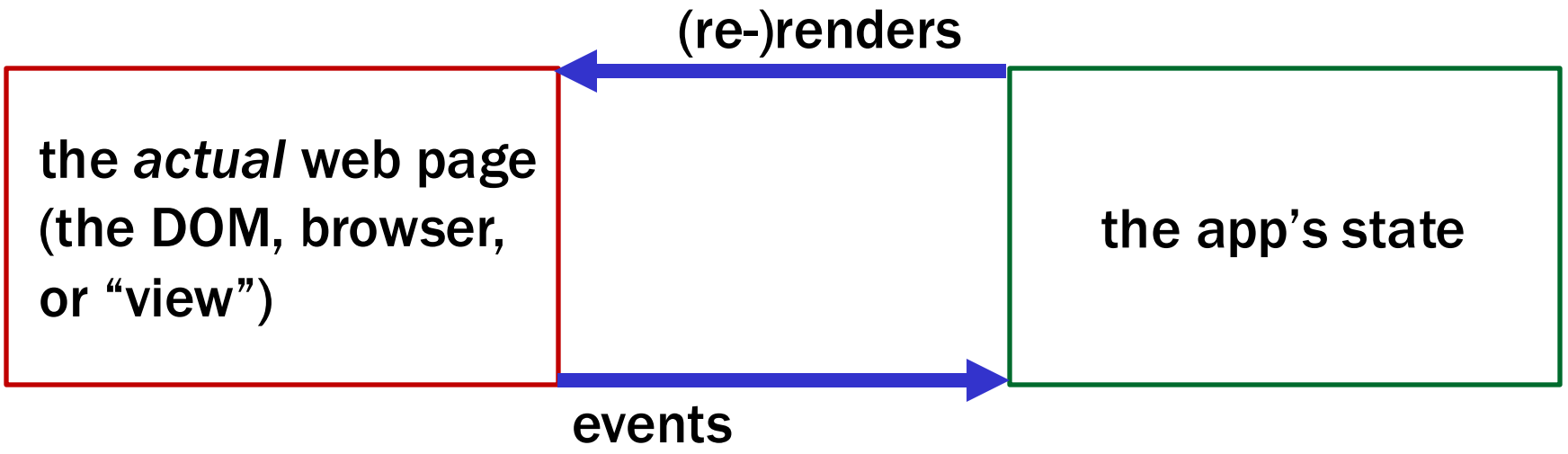
The React Model (2/3)



The React Model (3/3)



Reactive UIs, more generally



- you *only* have to write event handlers & render function
- but, you <u>have</u> to play by the rules, or new bugs!!
  - view *must* be a function of <u>*just*</u> the app state
  - render *must* be "pure" (no side effects!)

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={"Matt"}/>);
```

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

- **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 change it:**

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

    this.state = {name: "Matt"};
  }
  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

Hello Matt!

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>
```

  - **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!)

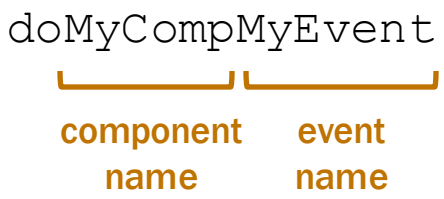
Second React Component: Mirrored State

```
        <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

CSE 331 Spring 2025

Typescript & Modular Code (with React)

Matt Wang

& Ali, Alice, Andrew, Anmol, Antonio, Connor, Edison, Helena, Jonathan, Katherine, Lauren, Lawrence, Mayee, Omar, Riva, Saan, and Yusong

JS Wacky Weekly Wednesday

Why does (in JS):

```
[98, 100, 99].sort()
```

evaluate to `[100, 98, 99]`?

How would JS evaluate:

```
[
 console.log, -5, -42,
 console.log("hi")
].sort()
```

Types?

- Many JS shenanigans have to do with types :(
- In HW1, we asked: would a **type checker** help?
  - students reported: 41% fall, 45% winter, spring TBD :)
  - industry studies found even higher numbers (over 60%)
- Mega-scale applications use type-checked languages
  - problems get even worse with multiple programmers
  - basically, unheard of to not use one
- Now: huge shift to "bolt-on" types to dynamically-typed languages (JS, Python, Ruby, …)

TypeScript

TypeScript Adds Declared Types to JavaScript

- TypeScript includes declared types for variables
  - file names end with `.ts` or `.tsx` (not `.js` or `.jsx`)
  - one extra config file `tsconfig.json`
- Compiler checks that the types are valid
  - produces JS just by *removing* the types
- Critical to **understand** how the type system works
  - know which bugs it catches and which it misses
  - you can then focus your attention on the second group

TypeScript Adds Declared Types

- Type is declared after the variable name:

```
const u: bigint = 3n;
const v: bigint = 4n;

const add = (x: bigint, y: bigint): bigint => {
  return x + y;
};

console.log(add(u, v));  // prints 7n
```

  - return type is declared after the argument list `(...)` and before `=>`

- "Where types go" is the main syntax difference vs Java

Basic Data Types of TypeScript

- **JavaScript includes the following types**

`number`	
`bigint`	
`string`	
`boolean`	
`null`	
`undefined`	
`Object`	(record types)
`Array`	(e.g., `string[]` as in Java)

- **TypeScript has these and also…**

`unknown`	(could be anything)
`any`	(turns off type checking — do not use!)

Function Type Notation

- Functions themselves have types. Given:

```
const add = (x: bigint, y: bigint): bigint => {
  return x + y;
};
```

- the type of add itself is

```
(x: bigint, y: bigint) => bigint
```

- different notion than “what does add return”
- will see this frequently with event handlers

Think Pair Share: I've Become So Numb(er)

```ts
// TS
const addTS = (x: number, y: number): number => {
  return x + y;
};

addTS(1n, 2n);
```

```java
// Java
double addJava(double a, double b) {
return a + b;
}

addJava(1, 2);
```

sli.do #cse331

Which of these would compile?

Literal Types

- Any literal value is also a type:

```
let x: "foo" = "foo";
let y: 16n = 16n;
```

- Variable can only hold that specific value!
  - can assign it again, but only with the same value
  - seems silly, but turns out to be useful...

Creating New Types: Unions

- **Union Types** `string | bigint`
  - can be either one of these

- Not possible in Java!
  - TS can describe types of code that Java cannot

- Unknown type is (essentially) a union

```
type unknown = number | bigint | string | boolean | ...
```

Enumerations

- **unions of literals are “enums”**

```
const dist = (dir: "left"|"right", amt: bigint): bigint => {
  if (dir === "right") {
    return amt;
  } else {
    return -amt;
  }
};
```

- **TypeScript ensures that callers will only pass one of those two strings (“left” or “right”)**
  - **impossible to do this in Java**

    (must fake it with the enumeration **design pattern**)

Java Enums

- **Another design pattern built into Java:**

```
enum Dir {
  LEFT, RIGHT
}
```

- `Dir.LEFT` **and** `Dir.RIGHT` **are the only 2 instances**

- **Cannot pass a** `Dir` **where** `String` **is expected**
  - **must add methods to convert between them**

Creating New Types: Records

- Can create compound types in multiple ways
  - put multiple types together into one larger type

- **Record Types** `{x: bigint, s: string}`
  - anything with *at least* fields "x" and "s"

```
const p: {x: bigint, s: string} = {x: 1n, s: "hi"};
console.log(p.x);  // prints 1n
```

Creating New Types: Tuples

- Can create compound types in multiple ways
  - put multiple types together into one larger type

- Tuple Types `[bigint, string]`
  - create them like this

```
const p: [bigint, string] = [1n, "hi"];  // an array
```

  - give names to the parts ("destructuring") to use them

```
const [x, y] = p;
console.log(x);  // prints 1n
```

Records vs Tuples

- Records and tuples provide the same functionality
  - both allow you to put parts together into one object
  - conceptually interchangeable

- They differ in *who* names the parts and *when*
  - **record**: creator picks the names

    everyone must use the same name
  - **tuple**: user of the tuple picks the names

    each user can pick their own names

- 331 convention: destructure tuples (only)
  - no reason to destructure records, so we disallow it

Optional Fields in TypeScript

- **Records can have optional fields**

```
type T = {x: bigint, y?: bigint};

const t: T = {x: 1n};
```

  - **type of "** `t.y` **" is "** `bigint | undefined` **"**

- **Functions can have optional arguments**

```
const f = (a: bigint, b?: bigint): bigint => {
  console.log(b);
};
```

  - **type of "** `b` **" is "** `bigint | undefined` **"**

Type Aliases

- TypeScript lets you give shorthand names for types

```
type Point = {x: bigint, y: bigint};

const p: Point = {x: 1n, y: 2n};
console.log(p.x);  // prints 1n
```

- Usually nicer but not necessary
  - e.g., this does the same thing

```
const p: {x: bigint, y: bigint} = {x: 1n, y: 2n};
console.log(x);  // prints 1n
```

Structural Typing

- **Deep difference between TypeScript and Java types**

- **TypeScript uses "structural typing"**
  - **sometimes called "duck typing"**

    "if it walks like a duck and quacks like a duck, it's a duck"

```
type T1 = {a: bigint, b: string};
type T2 = {a: bigint, b: string};

const x: T1 = {a: 1n, b: "two"};
```

  - **can pass " `x` " to a function expecting a " `T2` "!**
  - **can pass " `x` " to *any* function expecting a record with a bigint a and a string b**

Nominal Typing

- Java uses "**nominal typing**"

```
class T1 { int a; int b; }
class T2 { int a; int b; }

T1 x = new T1();
```

  - **cannot** pass "`x`" to a function expecting a "`T2`"

- Libraries do not interoperate unless it was pre-planned
  - create "adapters" to work around this

    example of a design pattern used to work around language limitations

Think Pair Share: Be There, or Be Square

```
type Square = { width: number };
type Rectangle = { width: number, height: number };
type SquareOrRect = { width: number, height?: number };

const squareArea = (s: Square) : number => {
  return s.width * s.width;
}
```

Can we pass to squareArea:

1. any Square?
2. squareOrRects with no height
3. any SquareOrRect?
4. any Rectangle?

sli.do #cse331

Type Inference

- If you leave off the type, TS will try to guess it
  - often, but not always, it guesses correctly

- This will work fine

```
const p = {x: 1n, y: 2n};
console.log(p.x);  // prints 1n
```

  - compiler should correctly guess `{x: bigint, y: bigint}`
  - can see in VS Code by hovering over "`p`"

Type Inference in 331

- If you leave off the type, TS will try to guess it
  - often, but not always, it guesses correctly

- 331 convention: type declarations are required on…
  - function arguments and return values
  - variables declared outside of any function ("top-level")

    these could be exported, so types should be explicit

- We do not require declarations on local variables
  - but it is fine to include them
  - if TS guesses wrong, you will need to include it

React Components and TypeScript

```
type HiProps = {name: string};
type HiState = {greeting: string};

class HiElem extends Component<HiProps, HiState> {

  constructor(props: HiProps) {
    super(props);

    this.state = {greeting: "Hi"};
  }
```

- Component is a generic type
  - first component is type of `this.props` (readonly)
  - second component is type of `this.state`

Linters

Linters

- Linters are like type checkers
  - try to find potential bugs in the program
  - as well as poor style / design issues

- In 331, we have our own linter ("comfy-tslint")
  - e.g., types are declared except local vars in functions
  - coming in HW3 :)

- They can be overzealous
  - can flag issues that aren't really problems
  - (happens with type checkers also, but less frequently)

Unused Variables & Linters

- **Linter will complain about unused variables**

```
const f = (a: bigint, b: bigint): bigint => {
  return b;
};
```

  - **linter will complain that `a` is unused**

    this looks suspicious, doesn't it?

- **This ignores variables whose names start with "_"**
  - **the underscore indicates you know it is unused**
  - **change the variable name to get rid of the error**