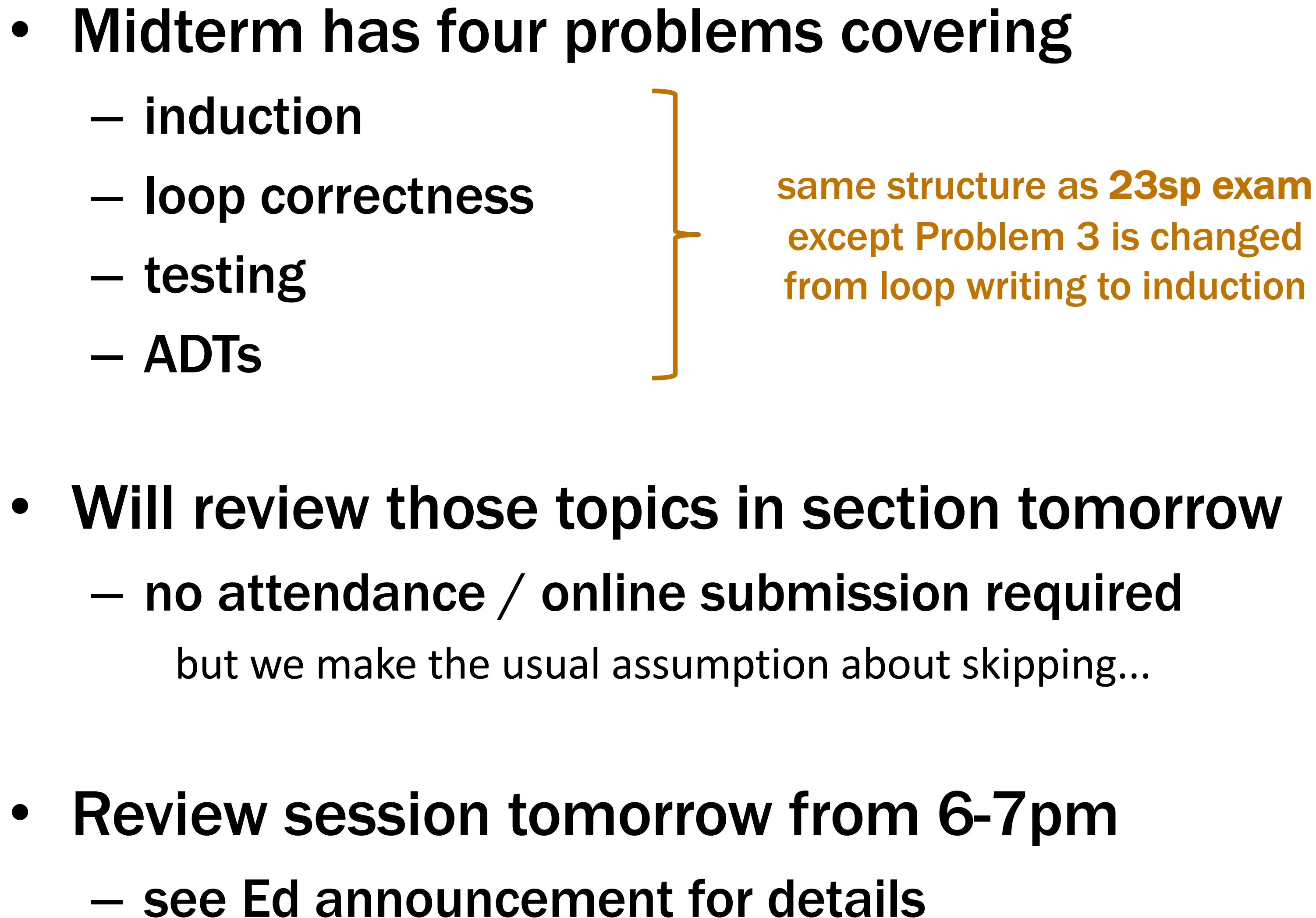
CSE 331

Stateful UI & Debugging

Kevin Zatloukal

Administrivia



Review: Stateful React Components

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

class HiElem extends Component<HiProps, HiState> {
  constructor(props: HiProps) {
    super(props);
    this.state = {greeting: “Hi”};
  }

  render = (): JSX.Element {
    return (<div>
        <p>{this.state.greeting}, {this.props.name}!</p>
        <button onClick={this.doEspClick}>Español</button>
      </div>);
  };

  doEspClick = (evt: MouseEvent<HTMLButtonElement>) => {
    this.setState({greeting: “Hola”});
  };
```

Review: React Components are Like ADTs

- **Components have an invariant like an RI**

  HTML on screen = render(this.state)

  – **don't want to be in a state where that is not true**

    unless you like **painful** debugging!

1. **Do not mutate `this.state`**

   Instead, call `this.setState(..)`

   React will update `this.state` and HTML on screen at the same time

2. **Make sure no data on screen would disappear on re-render**

   Need to use that information in your code

   Need to render exactly what is on the screen

Review: React Components are Like ADTs

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***

Review: React Components are Like ADTs

- Components have an invariant like an RI

  HTML on screen = render(this.state)

  - don't want to be in a state where that is not true

    unless you like **painful** debugging!

1. Do not mutate `this.state` (call `setState`)

   Instead, call `this.setState(..)`

   React will update `this.state` and HTML on screen at the same time

2. Make sure all user input is mirrored in state

   Need to use that information in your code

   Need to render exactly what is on the screen

Review: To-Do List

TodoApp – Add Click

```
// Called when the user clicks on the button to add the new item.
doAddClick = (_: MouseEvent<HTMLButtonElement>): void => {
  const name = … // how do we get the name??
  const items = this.state.items.concat(
      [ {name: name, completed: false} ]);
  this.setState({items: items});
};
```

- **we need the content of the new name input box**
  - don’t try to reach into the `document` to get it (that’s asking for trouble)

TodoApp – State

```
// State of the app is the list of items and the text that the
// the user is typing into the new item field.
type TodoState = {
  items: Item[];     // existing items
  newName: string;   // mirrors text in the field to add a new name
                     //    (need this for two reasons…)
};

…

// Called each time the text in the new item name field is changed.
doNewNameChange = (evt: ChangeEvent<HTMLInputElement>): void => {
  this.setState({newName: evt.target.value});
}
```

TodoApp – Add Click

```
// Called when the user clicks on the button to add the new item.
doAddClick = (_: MouseEvent<HTMLButtonElement>): void => {
  const name = this.state.newName;
  const items = this.state.items.concat(
      [ {name: name, completed: false} ]);
  this.setState({items: items});
};
```

Example: Buggy To-Do List

TodoApp – Render

```
// Return a UI with all the items and elements that allow them to
// add a new item with a name of their choice.
render = (): JSX.Element => {
  return (
    <div>
      <h2>To-Do List</h2>
      {this.renderItems()}
      <p className="instructions">Check an item to mark it…</p>
      <p className="more-instructions">New item:
        <input type="text" className="new-item"
               value={this.state.newName}
               onChange={this.doNewNameChange} />
        <button type="button" className="btn btn-link"
                onClick={this.doAddClick}>Add</button>
         (<a href="#" onClick={this.doHideClick}>Hide</a>)
      </p>
    </div>);
}
```

Buggy To-Do List

- Buggy app is not setting `value={..}` in text box
- Mirroring user input in state means
  1. Storing it in a field of `this.state`
  2. Writing the current value in the rendered HTML
- Re-render can occur when you don't expect it
  - especially when other people are writing code too
  - debugging is painful when it doesn't work

More React Gotchas

- Make sure you declare your methods like this

```
doBtnClick = (evt: MouseEvent<HTMLButtonElement>) => {…};
```

- Make sure you pass them like this

```
<button onClick={this.doBtnClick}>Click Me</button>
```

  – no "`()`" after the method name!

- Otherwise, the event handlers won't work

More React Gotchas

- **Note that `setState` is not instant**

```
// Suppose this.state.x is 2
this.setState({x: 3});
console.log(this.state.x);  // still 2!
```

  - **it adds an <u>event</u> that later updates the state**

    (React tries to batch together multiple updates)

More React Gotchas

- **Never modify anything in render**
  - **should be a pure function**
- **Never modify `this.state` outside of the constructor**
  - **use `this.setState` instead**
- **Remember that debugging will be painful**
  - **stateful components are inherently complex (Level 3)**
  - **separate anything complex into helper functions**
    - reason through them carefully and test them thoroughly
    - can have helper function that calculates new states, HTML to display, ec.
  - **write code to also double check (defensive programming)**

More Events

Events

- Components update their state when events occur
  - event calls a “handler”, which is a method of the class
  - event handler updates state via `setState`
- Some common examples
  - button click, hyperlink click
  - typing in text field
  - check box clicked
  - drop-down changed
  - timers
- See MDN for all possible elements and events…

Button Click Events

```
<button onClick={this.doBtnClick}>Click Me</button>
```

- **Click results in a call to our method**

```
doBtnClick = (evt: MouseEvent<HTMLButtonElement>) => {
  console.log("I've been clicked");
};
```

- **Event handlers are passed an event object**
  - **mouse clicks send** `MouseEvent` **objects**

    generic type with a parameter identifying the target of the click

Review: Event Handler Conventions

- We will use this convention for event handlers

```
doMyCompMyEvent
```

component name — event name

  - e.g., `doAddClick`, `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

- Linter will enforce this

components should be just rendering & event handlers

Link Click Events

```
<a href="#" onClick={this.doLinkClick}>Click Me</a>
```

- **Click results in a call to our method**

```
doLinkClick = (evt: MouseEvent<HTMLAnchorElement>) => {
  evt.preventDefault();  // don't change the URL
  console.log("I've been clicked");
};
```

- **Default action of a link is to go to that URL**
  - **harmless in this case (just adds "#" to the end of the URL)**
  - **can stop that with** `evt.preventDefault()`

Text Field Events

```
<input type="text" value={this.state.curText}
       onChange={this.doTextChange}></input>
```

current text

- Any typing in the text box causes a call to

```
doTextChange = (evt: ChangeEvent<HTMLInputElement>) => {
  console.log("Text is now: ${evt.target.value}");
  this.setState({curText: evt.target.value});
};
```

  - `evt.target` **is the thing that was clicked on**

    has type `HTMInputElement` in this case
  - "`.value`" **is the text currently shown in the text field**

    its value has just changed to something new

Check Box Events

```
<input type="checkbox" id="myCheckBox"
       onChange={this.doCheckChange}/>
<label htmlFor="myCheckBox">laundry</label>
```

☐ laundry

- **Clicking inside the box**

```
doCheckChange = (evt: ChangeEvent<HTMLInputElement>) => {
  console.log("Checked? ${evt.target.checked}");
};
```

  - `evt.target.checked` **is true / false**

- **Label contains the text to show next to the check box**
  - `htmlFor` **is useful for screen readers**

Drop-Downs

```
<select value="NA">
  <option value="NA">Pick a Quarter</option>
  <option value="20au">Fall 2020</option>
  <option value="21sp">Spring 2021</option>
</select>
```

Pick a Quarter ⌵

- **HTML `select` element creates a drop-down**
  - **one option for each choice**
  - **text in between `<option>` and `</option>` is shown**
  - **the `select`'s "`value`" at indicates which one is selected**

Drop-Downs

```
<select value="NA" onChange={this.doOptChange}>
  <option value="NA">Pick a Quarter</option>
  <option value="20au">Fall 2020</option>
  <option value="21sp">Spring 2021</option>
</select>
```

What's missing?

- **Picking an option causes an** `onChange`

```
doOptChange = (evt: ChangeEvent<HTMLSelectElement>) => {
  console.log("Picked option: ${evt.target.value}");
};
```

  – `evt.target.value` **is the "**`value`**" from the** `option` **chosen**

    `evt.target.value` here is either "`NA`", "`20au`", or "`21sp`"

Drop-Downs

```
<select value={this.state.opt}
        onChange={this.doOptChange}>
  <option value="NA">Pick a Quarter</option>
  <option value="20au">Fall 2020</option>
  <option value="21sp">Spring 2021</option>
</select>
```

- **State of component should be mirrored in state!**
  - **keep the selected value in a field, e.g., `this.state.opt`**

```
doptChange = (evt: ChangeEvent<HTMLSelectElement>) => {
  console.log("Picked option: ${evt.target.value}");
  this.setState({opt: evt.target.value});
};
```

Timers

```
setTimeout(this.doMyTimeout, 500);
```

- Calls the handler after 500 milliseconds

```
doMyTimeout = () => {
  console.log(“Timer went off!”);
};
```

  - no arguments provided

Arguments to Event Handlers

- **Often want to pass arguments to event handlers**
  - **can do so like this:**

```
setTimeout(() => this.doMyTimeout(“egg”), 500);

doMyTimeout = (name: string) => {
  console.log(“${name} timer went off!”);
};
```

  - **creates a new function on the spot**
  - **when called, that function calls** `doMyTimeout` **with the arg**

Arguments to Event Handlers

- The same thing applies to all other event handlers, e.g.

```
<input type="checkbox" id="myCheckBox"
  onChange={(evt) => this.doCheckChange(evt, "laundry")}/>
<label htmlFor="myCheckBox">laundry</label>

…

doCheckChange = (evt: ChangeEvent<HTMLInputElement>,
                 name: string) => {
  console.log("Done with ${name}? ${evt.target.checked}");
};
```

  - event handler takes the event and an argument

    `setTimeout`, in contrast, does not pass an event object

Review: To-Do List

TodoApp – State

```
// Represents one item in the todo list.
type Item = {
  name: string;
  completed: boolean;
};

// State of the app is the list of items and the text that the
// the user is typing into the new item field.
type TodoState = {
  items: Item[];     // existing items
  newName: string;   // mirrors text in the field to add a new name
                     //    (need this for two reasons…)
};
```

TodoApp – Render Items (abbreviated)

```
renderItems = (): JSX.Element[] => {
  const items: JSX.Element[] = [];
  for (let i = 0; i < this.state.items.length; i++) {
    if (!this.state.items[i].completed) {
      items.push(
        <div className="form-check" key={i}>
          <input className="form-check-input" type="checkbox"
                 id={"check" + i} checked={false}
                 onChange={(evt) => this.doItemClick(evt, i)} />
          <label className="form-check-label" htmlFor={"check"+i}>
            {this.state.items[i].name}
          </label>
        </div>);
    } else { … /* read-only once completed */ }
  }
  return items;
};
```

TodoApp – Item Clicked

```
// Called when the user checks the box next to an uncompleted item.
// The second parameter is the index of that item in the list.
doItemClick =
    (_: ChangeEvent<HTMLInputElement>, index: number): void => {
  const item = this.state.items[index];

  // Note: we cannot mutate the list. We must create a new one.
  const items = this.state.items.slice(0, index)    // 0 .. index-1
      .concat([{name: item.name, completed: true}])
      .concat(this.state.items.slice(index + 1));   // index+1 ..
  this.setState({items: items});

  // Remove the item in 5 seconds…
  setTimeout(() => this.doItemTimeout(index), 5000);
};
```

TodoApp – Item Timeout

```
// Called after an item has been removed for 5 seconds.
doItemTimeout = (index: number): void => {
  const item = this.state.items[index];

  // Note: we cannot mutate the list. We must create a new one.
  const items = this.state.items.slice(0, index)   // 0 .. index-1
      .concat(this.state.items.slice(index + 1));  // index+1 ..
  this.setState({items: items});
};
```

That's all the code in TodoApp.
If you can understand it all now,
then you're in great shape!

Debugging

A Bug's Life

- **Defect** ("the bug"): mistake made by a human
- **Error**: computation performed incorrectly
- **Failure**: mistake visible to the user

Debugging is the search
from failure back to defect

Debugging

- Debugging is <u>different</u> from coding
  - only happens when states are not as expected
    - variable has an unexpected type
    - state does not satisfy the expected assertions

- Never know how long it will take
  - only happens when you misunderstand something
  - important to start early!

Debugging

- **Debugging is <u>different</u> from coding**
  - **only happens when states are not as expected**
    - variable has an unexpected type
    - state does not satisfy the expected assertions

- **Arguably harder than coding…**

> "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

  - **write code as simply as possible**
    - if not level 0, then level 1
    - if not level 1, then level 2

Brian Kernighan

Debugging Sucks

- Rule #1 for debugging: **avoid** it

- Tips for *avoiding* debugging:
  1. Write the code as **simply** as possible

     save complication for complicated problems
  2. Apply rigorous testing and **reasoning**

     get the code right the first time
  3. Practice **defensive** programming

     catch errors as quickly as possible (reduce the search space)

- Tips for *doing* (surviving) debugging…
  – concise notes published on website

Debugging Tip #1

- Check the easy stuff first
  - make sure all the files are saved
  - restart the server
  - restart your computer
  - make sure someone didn't already fix it

- If it is one of the first 3, you will not find it debugging
  - every minute you spend until you hit save / restart is wasted

Debugging Tip #2

- Create a minimal example that demonstrates the bug
  - easier to look through everything in the debugger

- Shrink the input that fails:

Find “very happy” in “Fáilte, you are very welcome! Hi Seán! I am very very happy to see you all.”

Find “very happy” in “I am very very happy to see you all.”

not the accent characters

Find “very happy” in “very very happy”

something to do with partial match

Find “ab” in “aab”

How to Fix a Bug

- Start with a test that **fails**
  - make sure you see it fail!
  - can mistakenly write a test that worked already
- Understand why it fails
  - understand where your reasoning was wrong
- Fix the bug
- Make sure the all the tests now pass
  - new test and all previous tests

Debugging Tip #3

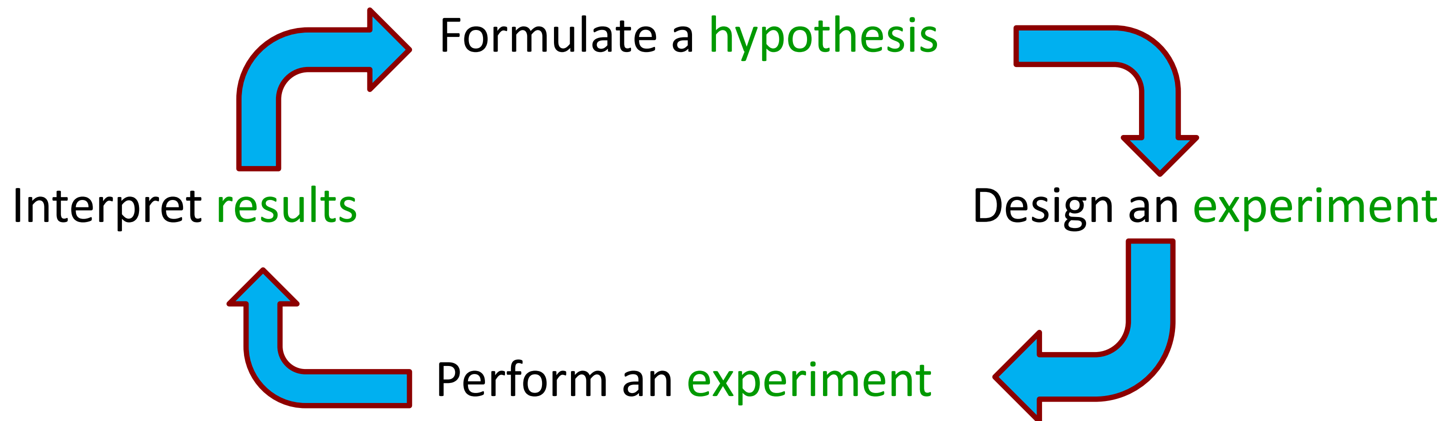
- Look for common silly mistakes
  - comparing records with `===`
  - misspelling the name of a method you were implementing
    - in Java, implementing "`equal`" instead of "`equals`"
  - passing arguments in the wrong order
- Easy for these to slip past reasoning
  - better chance of finding them with tools or testing
    - tools will miss wrong order if both arguments have the same type
  - but some will slip through

Debugging Tip #4

- Make sure it is a bug!
  - check the spec carefully
  - tricky specs can trick you
- These are the absolute worst
  - spend hours and then discover the code was right all along

Debugging Tip #5

- After 20+ min debugging, be systematic
  - don’t just try things you think might fix it
- Write down what you have tried
  - don’t try the same thing again and again
- Use the Scientific Method:



Debugging Tip #5

- **Use Binary Search to find the error**

  RI holds when the object is created

  …

  RI is violated when user clicks “submit”

- **Find an event that happens somewhere in the middle**

  RI holds when the object is created

  …

  does it hold when the user clicks on the dropdown?

  …

  RI is violated when user clicks “submit”

  - **save an alias to the object when created**

Debugging Tip #6

- Try explaining the problem to someone / something
  - can even be a rubber duck
    - Pragmatic Programmer calls this “rubber ducking”

- Talking through the problem often helps you spot it
  - this happens all the time

Debugging Tip #7

- Get some sleep!
  - the later it gets, the dumber I get
  - often don't realize it until 4–5am
- Common to wake up and instantly see the problem
- Important to start early!
  - can't do this the night it is due

Debugging Tip #8

- Get some help!
  - easy for bugs to hide in your blind spots
- After some number of hours, continuing is not helpful
  - need new ideas about where to look
- Important to start early!
  - no office hours late at night

Defensive Programming Tip #4

- If you spent 30+ min debugging, make it a **test case**
  - solid evidence that it’s a tricky case
- Bugs that happen once often come back
  - code is changed in the future
  - good chance the same error will happen in the new version
- These are called “regression tests”
  - avoid the bug coming back (“regressing”)