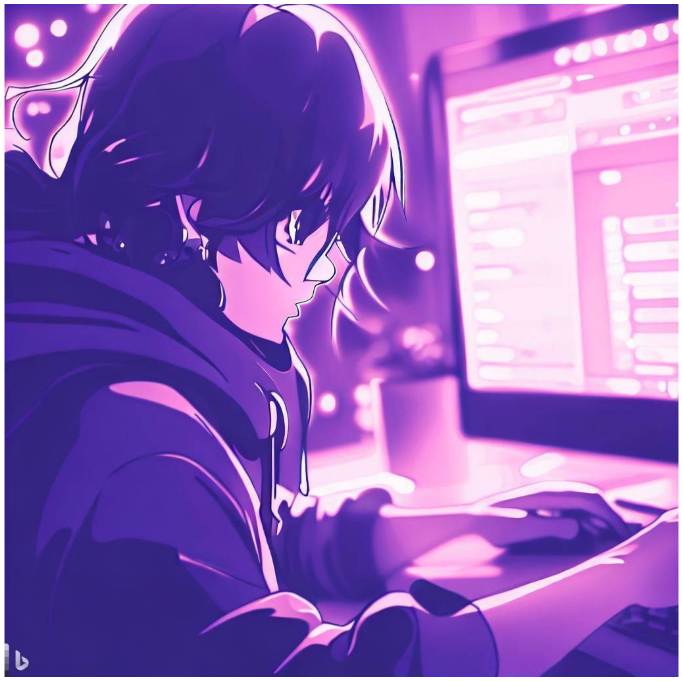


CSE 331

Stateful UI in React

Kevin Zatloukal

Administrivia

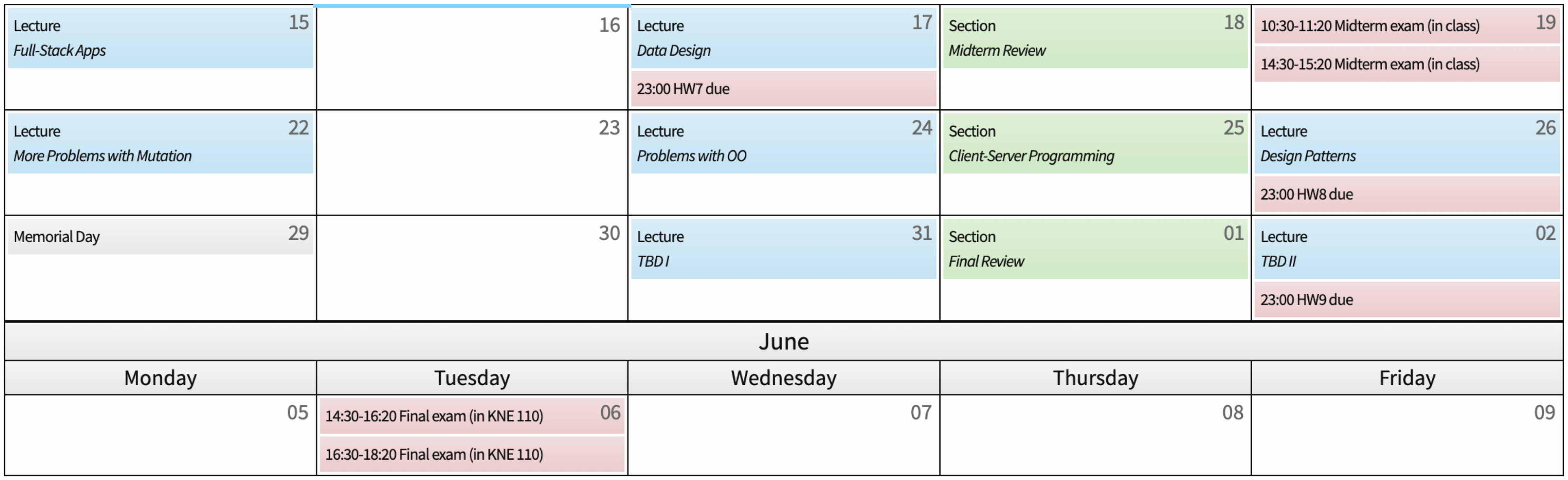
- HW7 released yesterday
  - it is probably longer than HW5-6
  - start early!

- Work through 3 versions of an ADT
  - changing representations and specifications

- Finished all material on correctness
  - tools, testing, reasoning, and defensive programming
    - proof by calculation, cases, induction
    - Floyd logic, arrays
    - AF and RI

Remaining Work

- Last four weeks include
  - midterm and final exam
  - HW8 and HW9 (full app on your own)



15 Lecture *Full-Stack Apps*	16	17 Lecture *Data Design*; 23:00 HW7 due	18 Section *Midterm Review*	19 10:30-11:20 Midterm exam (in class); 14:30-15:20 Midterm exam (in class)
22 Lecture *More Problems with Mutation*	23	24 Lecture *Problems with OO*	25 Section *Client-Server Programming*	26 Lecture *Design Patterns*; 23:00 HW8 due
29 Memorial Day	30	31 Lecture *TBD I*	01 Section *Final Review*	02 Lecture *TBD II*; 23:00 HW9 due

June

Monday	Tuesday	Wednesday	Thursday	Friday
05	06 14:30-16:20 Final exam (in KNE 110); 16:30-18:20 Final exam (in KNE 110)	07	08	09

Recall: Array Loop Expectations

In 331, expect you to (eventually) be able to

1. Write invariant that is a simple weakening of Post
   - problems of lower complexity

HW8–9

2. Write the code, given the invariant
   - problems of moderate complexity
3. Check correctness, given code with invariant
   - problems of higher complexity
   - (not possible without invariant)

exams

Remaining Work

- HW8 and HW9 focus on practical skills
  - build full stack apps
  - some help in HW8
  - no help in HW9

- Tests focus on theoretical knowledge
  - e.g., checking correctness of complex loops
  - midterm is *practice* for the final
    - covered all the material already (HW1–7)
    - midterm worth about the same points as HW8 & HW9

Midterm

- Midterm exam has 4 problems covering
  1. Correctness of a complex loop
  2. Writing a loop correctly given the invariant
  3. Writing code correctly given no invariant
  4. Testing a complex loop

- Study HW5-6 and related section material
  - some other example tests on the web site
  - not necessarily representative of our problems

    tests are from other instructors, in different quarters

Stateful UI in React
(React Components)

UI in HW1-4

- UI so far was static
  - `index.tsx` calls `render` to show a fixed UI

    UI was different based on query params

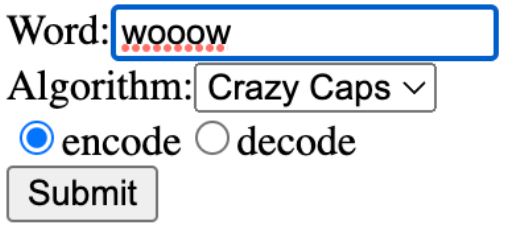
    but never changed once rendered

- Made the UI change by reloading the page
  - change the query params, so it renders something different

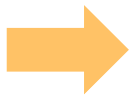
UI in HW1-4

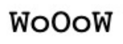
- Made the UI change by reloading the page
  - change the query params, so it renders something different

`http://localhost:8080/` `http://localhost:8080/?word=wooow&...`



Word: wooow
Algorithm: Crazy Caps
[✓ encode] decode
Submit





```
WoOoW
```

```
const word = params.get("word");
if (word === null) {
  root.render(<MakeForm/>);
} else {
  root.render(<ShowResults word={word} ../>);
}
```

UI in HW1-4

- Reloading is not great as a user experience
  - page reloads are slow
  - page reloads can lose state (e.g., content of text fields)

- Better to re-render the page without a reload

React Functions

- React let us create custom tags
  - e.g., from HW2

    ```
    root.render(<SquareElem square={sq}/>);
    ```

  - acts like the call

    ```
    root.render(SquareElem({square: sq}));
    ```

  - where SquareElem is function taking a record argument

    ```
    function SquareElem(props: {square: Square}): JSX.Element
    ```

- HTML returned by the function is displayed
  - “`SquareElem`” tag is in the HTML
  - `render` spots it, calls the function, and replaces the tag

React Components

- **Can do the same with a class (a React Component):**

```
class HiElem extends Component<{name: string}, {}> {
  render = (): JSX.Element => {
    return <p>Hi, {this.props.name}</p>;
  };
}
```

- **Use via** `<HiElem name={“Fred”}/>`
  - **React instantiates the class and calls its** `render` **method**

- **React calls render to get the HTML to display**
  - **constructor stores argument in a field called “**`props`**”**

    props type is `SqProps`

React Components

- Can do the same with a class (a React Component):

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

class HiElem extends Component<HiProps, {}> {
  render = (): JSX.Element {
    return <p>Hi, {this.props.name}</p>;
  };
}
```

- Can define a shorthand for the type

No sensible reason to make Components without state

- Component is a generic type
  – first type parameter is the type of "`props`"
  – second type parameter is for "state"…

React Components

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

class HiElem extends Component<HiProps, HiState> {
  constructor(props: HiProps) {
    super(props);
    this.state = {curName: this.props.name};
  }
```

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

    initial value set in the constructor

    never *directly* modified after that

React Components

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

class HiElem extends Component<HiProps, HiState> {

  render = (): JSX.Element {
    return <p>Hi, {this.state.curName}</p>;
  };
```

- `render` **can use both** `this.props` **and** `this.state`
  - **difference is that state can be changed**

    props never change
  - **React will automatically re-render when state changes**

    re-render happens shortly after the state change

React Components

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

class HiElem extends Component<HiProps, HiState> {
  …
  setName = (newName: string): void => {
    this.setState({curName: newName});
  };
}
```

- **Must call `setState` to change the state**
  - **directly modifying `this.state` is a (painful) bug**
- **React will automatically re-render when state changes**
  - **this is the (only) reason to use a Component**

React Components

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

class HiElem extends Component<HiProps, HiState> {
  …
  setName = (newName: string): void => {
    this.setState({curName: newName});
  };
}
```

- **Must call** `setState` **to change the state**
  - **directly modifying** `this.state` **is a (painful) bug**

- **Only need to supply the fields that have changed**
  - **all the other fields will stay as they were before**

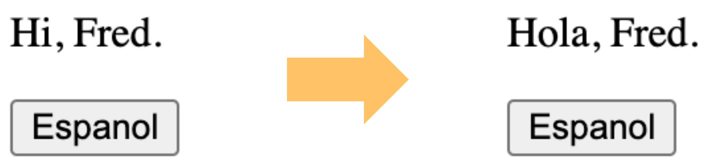
React Components

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

class HiElem extends Component<HiProps, HiState> {
  constructor(props: HiProps) {
    super(props);
    this.state = {curName: this.props.name};
  }

  render = (): JSX.Element {
    return <p>Hi, {this.state.curName}</p>;
  };

  setName = (newName: string): void => {
    this.setState({curName: newName});
  };
}
```

React Components

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

class HiElem extends Component<HiProps, HiState> {
  …
  setName = (newName: string): void => {
    this.setState({curName: newName});
  };
}
```

- **How could `setName` be called?**
  - **typically happens in a handler for an HTML event**



React Component with an Event Handler

- Pass method to be called as argument
  - value of `onClick` attribute is our `makeSpanish` method

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

- Browser will invoke that method when button is clicked

```
makeSpanish = (evt: MouseEvent<HTMLButtonElement>) => {
   this.setState({greeting: "Hola"});
 };
```

  - Call to `setState` causes a re-render (in a bit)

React Component with an Event Handler

```
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.makeSpanish}>Espanol</button>
      </div>);
  };

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

React Components are Like ADTs

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

- “Props” are part of the specification (arguments)
  - public interface, used by clients

```
root.render(<Hi name={“Fred”}/>);  // pass in name
```

- “State” is the concrete representation
  - private choice of data structures, hidden from clients

```
constructor(props: HiProps) {
  super(props);
  this.state = {greeting: “Hi”};   // initial state
}
```

React Components are Like ADTs

- Can have RIs on state as well

```
// RI: 0 <= index < options.length
type OptionState = {
  options: string[],
  index: number
};
```

- Good idea to write a `checkRep` here also!

React Components are Level 3

- **Like ADTs, methods are sharing state**
  - **change in one method is read in other methods**

- **Debugging will be harder!**

- **Move complex parts into separate functions**
  - **class is ideally just be render and simple event handlers**
    - move everything complex into helper functions
    - e.g., calculation of new state can be a helper function
  - **harder to reason about and test Level 3, so keep it simple**

- **Write code to check your invariants**
  - **ensure the new state is valid before calling** `setState`

React Components are Like ADTs

- HTML on the screen is a (hidden) part of the state
  - components work with React to manage this state

- `render` method is like an AF
  - defines the correct HTML to display for the given state

- Components have an invariant like an RI

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

React Components are Like ADTs

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

	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 Components are Like ADTs

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

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

  – **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`)**

   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**

   More on this later…

Example: To-Do List