CSE 331
Software Design & Implementation

Kevin Zatloukal
Spring 2022
Modern Web GUIs
Administrivia

- Section tomorrow should be very useful
  - focus will be on homework prep as usual
Structure of a GUI Application

• Core parts of these applications:
  – stores some data for the user
  – displays that data for the user
  – allows the user to change the data
    • causes the app to re-display

• Library provides a set of components we can use
Structure of a GUI Application

- **Model**: data and invariants
- **View / Controller**: presentation, user interaction

GUI Components? (provided by library)
Structure of a GUI Application

View / Controller sits in between model and GUI components
  – performs two key tasks…
View / Controller sits in between model and GUI components
1. Renders ("draws") the model for the user via components
Structure of a GUI Application

View / Controller sits in between model and GUI components
1. Renders (“draws”) the model for the user via components
2. Updates the model based on user interaction
   – causes the app to draw again
JS Example

register-js/index.js
Remaining Problems

• Code is extremely verbose
  – can be improved using Lambdas

• Code is not sufficiently modular
  – one JS mixes data, display, interaction

• Too much work involved with laying out elements

• Poor tool support
  – HTML is created in strings!
  – (and other issues not mentioned so far…)
ES6
From last time: Fake Classes

- JavaScript started as an OO language w/out classes

- Can do some of what we need already:

  ```javascript
  let obj = {f: (x) => x + 1};
  console.log(obj.f(2));  // 3
  ```

- Use “this” to read fields of `obj` in `obj.f`
class Foo {
    constructor(val) {
        this.secretVal = val;
    }

    secretMethod(val) {
        return val + this.secretVal;
    }
}

let f = new Foo(3);  // {secretMethod: ..., secretVal: ...}
console.log(f.secretMethod(5));  // 8
Classes

• **new** Foo creates an object already containing methods
  – also calls the constructor

• Still has the same issue with this:

```javascript
class Foo { ... }

let f = new Foo(3);
let s = f.secretMethod;
console.log(s(5));       // NaN

let t = (x) => f.secretMethod(x);
console.log(t(5));       // 8
```
JS vs Java Classes

• JS method signatures are just the name
  – JS objects are just HashMaps
  – field names are the keys

• Java methods signatures are name + arg types
  – e.g., `avg(int, int)`

• JS has only one method with a given name
  – language allows different numbers of arguments
    • missing arguments are undefined
  – can strengthen a spec by accepting a wider set of possible input types

```javascript
obj.avg(3, 5)
```
Modules

• Each file is a separate unit ("namespace")

• Only exported names are visible outside:

        export function average(x, y) { ... }

• Others can import using:

        import { average } from './filename';

  – file extension is sometimes not included
ES6 Example

register-js2/…
Remaining Problems

• Code is extremely **verbose**
  – can be improved using Lambdas

• Code is **not sufficiently modular**
  – one JS mixes data, display, interaction

• **Too much work** involved with laying out elements

• Poor **tool support**
  – No compile-time types
  – HTML is created in strings!
  – (and other issues not mentioned so far…)

UI is still in one file
TYPESCRIPT
TypeScript

• Adds type constraints to the code:
  – arguments and variables
    ```typescript
    let x: number = 0;
    ```
  – fields of classes (now declared)
    ```typescript
    quarter: string;
    ```

• tsc performs type checking
  – outputs version with type annotations removed
TypeScript Types

• Basics from JavaScript:
  number, string, boolean, string[], Object

• But also
  – specific classes Foo
  – tuples: [string, number]
  – unions: string | number
  – enums (as in Java)
  – allows null to be included or excluded (unlike Java)
  – any type allows any value
  – abbreviations: type Point = [number, number]
  – …
Simple Examples

points1.ts
points2.ts
UI Example

register-ts/…
TypeScript

- Type system is unsound
  - can’t promise to find prevent all errors
  - can be turned off at any point with any types
    - `x as Foo` is an unchecked cast to `Foo`
    - `x!` casts to non-null version of the type (useful!)

- Full description of the language at [typescriptlang.org](https://typescriptlang.org)
JSX
• Fix another problem by adding HTML as a JS type

• This is supported in .jsx files:

        let x = <p>Hi, {name}.</p>;

• Compiler can now check that this is valid HTML

• {...} replaced with string value of expression
JSX Gotchas

• Put (...) around HTML if it spans multiple lines

• **Cannot use `class=“btn”` in your HTML**
  – `class`, `for`, etc. are reserved words in JS
  – use `className=“..”, htmlFor=“..”, etc.

• **Must have a single top-level tag:**
  – **not**: `return <p>one</p><p>two</p>;
  – usually fixed by wrapping those parts in a `div`
Remaining Problems

• Code is extremely **verbose**
  – can be improved using Lambdas

• Code is **not sufficiently modular**
  – one JS mixes data, display, interaction

• **Too much work** involved with laying out elements

• Poor **tool support**
  – No compile-time types
  – HTML is created in strings!

UI is still in one file
REACT
React

- Improve modularity by allowing custom tags

```javascript
let app = (
    <div>
        <TitleBar name="My App"/>
        <EditPane rows="80" />
    </div>);
```

- `TitleBar` and `EditPane` can be separate modules
  - their HTML gets substituted in these positions
React

- Custom tags implemented using classes

```javascript
class TitleBar extends React.Component {

- Attributes `name="My App"` passed in `props arg`

- Method `render` produces the HTML for component

- Framework joins all the HTML into one blob
  - can update in a single call to `innerHTML = ...`
React Example

register-react/…
React Components

• Each React component renders into HTML elements

```javascript
let app = (
    <div>
        <TitleBar name="My App"/>
        <EditPane rows="80" />
    </div>
);
```

• React components corresponds to portions of the document
  - TitleBar is one subtree
  - EditPane is another subtree
  - App contains the two of those
React State

• Last example was not dynamic
  – there was no model!

• Components become dynamic by maintaining state
  – stored in fields of `this.state`
  – call `this.setState({field: value})` to update

• React will respond by calling `render` again
  – will automatically update the live HTML to match
  – will only update the parts that changed
Structure of GUI Application

Each React component renders into HTML elements
Each React component includes
  • part of the model
  • part of the view (rendering of that data into components)
  • part of the controller (listeners for interaction with that view)
Structure of Example React App

- **App**
  - State:
    - quarter

- **Quarter Picker**
  - Props:
    - quarter

- **Class Picker**
  - Props:
    - quarter
  - State:
    - classes
Example 5

register-react2/…
React State

- Custom tag also has its own events

- Updating data in a parent:
  - sends parent component new data via event
  - parent updates state with `setState`
  - React calls parent’s `render` to get new HTML
    - result can include new children
    - result can include changes to child props
Structure of Example React App

App
- State:
  - quarter

Quarter Picker
- onPick
- Props:
  - quarter
- State:
  - classes

Class Picker
- onBack
- Props:
  - quarter
- State:
  - classes
Splitting the Model

- State should exist in the **lowest common parent** of all the components that need it
  - sent down to children via *props*

- Children change it via *events*
  - sent up to the parent so it can change its state

- Parent’s render creates new children with new props
Remaining Problems

• Code is extremely **verbose**
  – can be improved using Lambdas

• Code is *not sufficiently* **modular**
  – one JS mixes data, display, interaction

• Too much work involved with laying out elements

• Poor **tool support**
  – No compile-time types
  – HTML is created in strings!
Event Listener Gotchas

• Recall the issue with “this” in JavaScript.
  – **do not** write `onClick={this.handleClick}`

• Three ways to do this properly:
  1. `onClick={e => this.handleClick(e)}`
  2. `onClick={this.handleClick.bind(this)}`
  3. Make `handleClick` a field rather than a method:
     
     ```js
     handleClick: (e) => { ... };
     ```
     
     Then `this.handleClick` is okay.
React setState Gotchas

- **setState** does not update state instantly:

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

- Update occurs after the event finishes processing
  - **setState** adds a new event to the queue
  - work is performed when that event is processed

- React can batch together multiple updates
Other React Gotchas

• Model must store all data necessary to generate the exact UI on the screen
  – react may call `render` at any time
  – must produce identical UI

• Any state in the HTML components must be mirrored in the model
  – e.g., every text field’s `value` must be part of some React component’s state
  – render produces
    `<input type="text" value={...}>`
Other React Gotchas

• `render` should not have side-effects
  – only `read` `this.state` in `render`

• Never modify `this.state`
  – use `this.setState` instead

• Never modify `this.props`
  – read-only information about parent’s state

• Not following these rules may introduce bugs that will be hard to catch!
React Performance

• React re-computes the tree of HTML on state change
  – can compute a “diff” vs last version to get changes

• Surprisingly, this is not slow!
  – slow part is calls into browser methods
  – pure-JS parts are very fast in modern browsers
  – processing HTML strings is also incredibly fast
React Tools

• Use of compilers etc. means new tool set

• npm does much of the work for us
  – installs third-party libraries
  – runs the compiler(s)