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`
Classes

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., \( \text{avg}(\text{int}, \text{int}) \)

• 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

\( \text{obj.avg}(3, 5) \)
Modules

• Each file is a separate unit (“namespace”)

• Only exported names are visible outside:

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

• Others can import using:

    ```javascript
    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
    ```
    let x: number = 0;
    ```
  – fields of classes (now declared)
    ```
    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
    • \(x \text{ as } \text{Foo}\) is an unchecked cast to \(\text{Foo}\)
    • \(x!\) casts to non-null version of the type (useful!)

• Full description of the language at
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
UI Modularity

• **Key idea**: break the *visible* UI into pieces that can become separate components

---

Component Tree

• App
  • Title Bar
  • Side Bar
  • Main Body
    – children…
UI Modularity

• **Key idea**: break the *visible* UI into pieces that can become separate components
  – each component should know how to turn itself into GUI components (panels, buttons, etc.)

• **Problem**: How do all the pieces get put together?
  – the GUI must be **one tree**, not many
REACT
React

• Improve modularity by allowing custom tags

```jsx
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