CS142 Midterm Review

Exam Logistics

- When: Wednesday, May 8, 7:30pm – 9:00pm
- Where: Cubberley Auditorium
- What: Covering front-end material (Up to but **not** including HTTP)
- Type: Closed book. Two double-sided 8.5x11" pages of notes are allowed

HTML

Web in browsers work on documents, traditional GUIs start with pixels

Concept: **Markup Language** - (HTML) - Content with annotations

HTML annotations - tags convey:

- Meaning of text in document (header, title, paragraph)
- Additional information (<img>)
- Some formatting (<b>) - CSS does most styling

HTML syntax

- Tags and attributes inside '<' … '>'
- Whitespace mostly not important
- Document structure contains head and body
- XHTML strict

CSS

- Key concept: Separate style from content
  - HTML describe content
  - CSS describes what it looks like

- Style sheets removed styling directive from HTML markup
- DRY principle - Don't repeat yourself
- CSS Rules
  - Selector (e.g. class, tag, id, …)
  - Declaration (property/value pairs)
- Also used in animation

CSS allows control of

- Coloring - RGB
- Size - CSS Box Model
- Position - Pixel or inches
- Visibility
- Has inheritance for some properties but not others

URLs - Uniform Resource Locators

HyperText - Text with links - Links point to names (URLs)

Naming system for the Internet

Lacked **referential integrity**

Used by the browser to fetch things

Parts of an URL

http://host.company.com:80/a/b/c.html?user=Alice&year=2008#p2

Scheme (**http:**): identifies protocol used to fetch the content.

Host name (**//host.company.com**): name of a machine to connect to.

Server's port number (**80**): allows multiple servers to run on the same machine.

Hierarchical portion (**/a/b/c.html**): used by server to find content.

Query parameters (**?user=Alice&year=2008**): provides additional parameters

Fragment (**#p2**): Have browser scroll page to fragment (html: p2 is anchor tag)

Used on the browser only; not sent to the server.

JavaScript

- Example of a **scripting language**

  ... **high-level**, **dynamic**, **untyped**, and **interpreted** programming language

  ... is **prototype-based** with **first-class functions**, …

  ... supporting **object-oriented**, **imperative**, and **functional programming**

  ... has an API for working with **text**, **arrays**, **dates** and **regular expressions**

- C-like syntax

JavaScript - The tricky parts

Variables take on type of last assignment (**dynamic** typing)

All `var` statements **hoisted** to top of scope

Object inheritance with **prototypes**

Function method with **`this`** pointer

Class defined by constructor functions

Functional programming

Closures

Hoisting - what gets printed out?

Only **declarations** are hoisted, not assignment

`let` is not hoisted

A

```
var x = 5;  // Initialize x

console.log( x+y);

var y = 7;  // Initialize y
```

B

```
var x = 5;  // Initialize x

y = 7;

console.log( x+y);

var y;  // Initialize y
```

Object Inheritance with prototypes

```
// constructor function
function MyClass () {
  var privateVariable; // private member only available within the constructor fn

  this.privilegedMethod = function () { // it can access private members
    //..
  };
}

// A 'static method', it's just like a normal function
// it has no relation with any 'MyClass' object instance
MyClass.staticMethod = function () {};

MyClass.prototype.publicMethod = function () {
  // the 'this' keyword refers to the object instance
  // you can access only 'privileged' and 'public' members
};

var myObj = new MyClass(); // new object instance

myObj.publicMethod();
MyClass.staticMethod();
```

A little bit of THIS and little bit of that

when a function executes, it gets the `this` property—a variable with the value of *the object* that invokes the function where `this` is used.

it contains the value of the object that invokes function

`this` is really just a shortcut reference for the “antecedent object”—the invoking object.

What happens when you click the button?

```
<html>
  <head>
  </head>
  <body>
    <button type="button" id="my-button">ClickMe</button>
    <script>
      var user = {
        data: [
          {name: 'Tiger Woods', age: 37},
          {name: 'Phil Mickelson', age: 43}
        ],
        clickHandler: function(event) {
          // get a random number between 0-1
          var randomNumber = ((Math.random() * 2 | 0) + 1) - 1;
          console.log(this.data[randomNumber].name);
        }
      };

      document.getElementById('my-button').onclick = user.clickHandler;
    </script>
  </body>
</html>
```

Fix:

```
<html>
  <head>
  </head>
  <body>
    <button type="button" id="my-button">ClickMe</butt
    <script>
      var user = {
        data: [
          {name: 'Tiger Woods', age: 37},
          {name: 'Phil Mickelson', age: 43}
        ],
        clickHandler: function(event) {
          // get a random number between 0-1
          var randomNumber = ((Math.random() * 2 | 0) + 1) - 1;
          console.log(this.data[randomNumber].name);
        }
      };

      document.getElementById('my-button').onclick = user.clickHandler.bind(user);
    </script>
  </body>
</html>
```

Use `bind()`, `apply()`, `call()` , or an arrow function to set the value of `this` properly

Closures

A closure is an inner function that has access to the outer (enclosing) function's variables—scope chain.

The closure has three scope chains:

- it has access to its own scope (variables defined between its curly brackets)
- it has access to the outer function's variables and parameters
- it has access to the global variables.

Closures **store references** to the outer function's variables; they **do not store the actual value**.

interesting when the value of the outer function's variable changes before the closure is called.

You see this a lot in for loops where everything has the last value of **i**

Closures have access to the outer function's variable even after the outer function returns

Closures

```
var add = (function () {
    var counter = 0;
    return function () {return counter += 1;}
})();

add();
add();
add();

// the counter is now 3
```

The variable **add** is assigned
the return value of a self-invoking function.

The self-invoking function only runs once.
It sets the counter to zero (0), and returns a function expression.
This way add becomes a function.

The closure (return function) can access the counter in the parent scope.
It makes it possible for a function to have "**private**" variables.
The counter is protected by the scope of the anonymous function, and can only be changed using the add function.

Document Object Model (DOM)

- Browser JavaScript interface to HTML document
- Hierarchy of JavaScript object representing the document structure
- Can read & write to document (e.g. innerHTML does both)

Using the DOM

How to access specific elements

- getElementById
- getElementsByTagName
- getElementsByClassName

How to traverse the DOM

- walk up and down - parent/child
- walk sideways - siblings

How to modify the DOM

- innerHTML
- textContent
- get/setAttributes
- appendChild

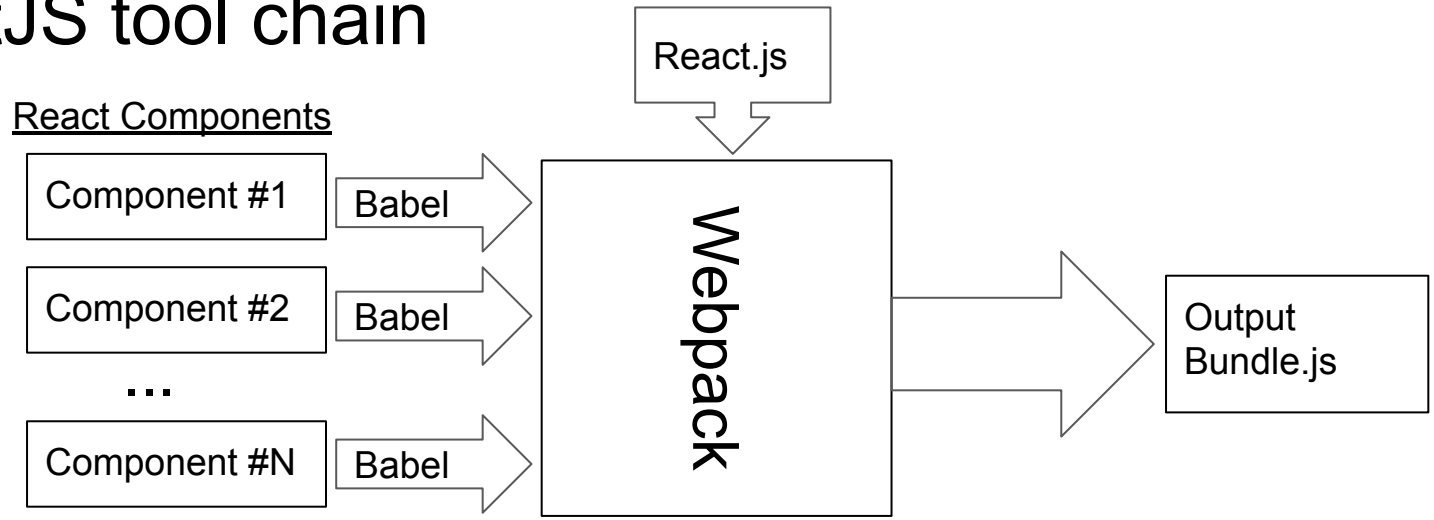
Events

- DOM communicates to JavaScript with Events
  - Mouse, keyboard, timer, network, etc.
- Event Handlers (listeners)
  - Capture phases (runs first)
  - Bubble phase (runs second)
  - stopPropagation to stop bubbling up/trickling down
- Issues with `this` binding

Model-View-Controller (MVC) Pattern

- **Model**: manages the application's data
  - JavaScript objects. Photo App: User names, pictures, comments, etc.
- **View**: what the web page looks like
  - HTML/CSS. Photo App: View Users, View photo with comments
- **Controller**: fetch models and control view, handle user interactions,
  - JavaScript code. Photo App: DOM event handlers, web server communication

ReactJS tool chain



Babel - Transpile language features (e.g. ECMAScript, JSX) to basic JavaScript

Webpack - Bundle modules and resources (CSS, images)

Output loadable with single `script` tag in any browser

components/ReactAppView.js - ES6 class definition

```
import React from 'react';

class ReactAppView extends React.Component {
  constructor(props) {
    super(props);
    ...
  }
  render() { ...
};

export default ReactAppView;
```

Inherits from React.Component. `props` is set to the attributes passed to the component.

Require method render() - returns React element tree of the Component's view.

ReactAppView render() method

```
<div>
   <label>Name: </label>
   <input type="text" ... />
   <h1>Hello {this.state.yourName}!</h1>
</div>
```

```
render() {
    let label = React.createElement('label', null,'Name: ');
    let input =  React.createElement('input',
           { type: 'text', value: this.state.yourName,
               onChange: (event) => this.handleChange(event) });
    let h1 = React.createElement('h1',  null,
                          'Hello ', this.state.yourName, '!');

    return React.createElement('div', null, label, input, h1);
}
```

Returns element tree with `div` (`label`, `input`, and `h1`) elements

Use JSX to generate calls to createElement

```
render() {
   return (
      <div>
        <label>Name: </label>
        <input
           type="text"
           value={this.state.yourName}
           onChange={this.handleChange}
        />
        <h1>Hello {this.state.yourName}!</h1>
      </div>
    );
}
```

Component state and input handling

```
import React from 'react';

class ReactAppView extends React.Component {
  constructor(props) {
    super(props);
    this.state = {yourName: ""};
  }
  handleChange = (event) => {
    this.setState({ yourName: event.target.value });
  }
  ....
```

Make `<h1>Hello {this.state.yourName}!</h1>` work

- **Calls to `setState` cause React to call `render()` again**

Single Page Applications

- Run app as a single page fetched from web server
- Support deep-linking
  - Bookmarking places in app
  - Sharing places in app
  - Maintain the app's context state in the URL, or...
  - Provide a share button to generate deep linking URL

http://www.example.org/dirmod?sid=789AB8&type=gen&mod=Core+Pages&gid=A6CD4967199

versus

http://www.example.org/show/A6CD4967199

Passing parameters with React Router

- Parameter passing in URL

```
<Route
   path="/Book/:book/ch/:chapter"
   component={BookChapterComponent}
/>
```

- Parameters put in `prop.match` of the component

```
function BookChapterComponent({ match }) {
  return ( <div>
             <h3>Book: {match.params.book}</h3>
             <h3>Chapter: {match.params.chapter}</h3>
           </div> );
}
```

```
<Link to="/Book/Moby/ch/1">
     Moby
</Link>
```

Book: Moby

Chapter: 1

Responsive implementation

- Build components to operate at different screen sizes and densities
  - Use relative rather than absolute
  - Specify sizes in device independent units
- Use CSS breakpoints to control layout and functionality
  - Layout alternatives
  - App functionality conditional on available screen real estate
- Mobile first - popular strategy
  - Expand a good mobile design to use more real estate

Web Apps

Design + implementation

- Consistency, providing context, fast response (don’t make the user wait)

Style guides and design templates (example: Google’s *Material Design*)

- Covers the look and feel of the app
- Use design patterns. Be consistent
- Follow a familiar structure

Grid layout is a popular strategy

Consider Internationalization & Accessibility - users are not all the same!

Testing the web app

- Unit testing
  - Each test targets a particular component and verifies it does what it claims it does
  - Requires mock components for the pieces that component interacts with
  - Example: Load an React component and run tests against it
    - Need to mock everything this component touches (DOM, libraries, models, etc.)
- End-to-End (e2e) testing
  - Run tests against the real web application
  - Scripting interface into browser used to drive web application
  - Example: Fire up app in a browser and programmatically interact with it.
    - WebDriver interface in browsers useful for this
- Metric: Test Coverage
  - Does every line of code have a test?

Good luck!