React, Part 1

SWE 432, Fall 2019

Web Application Development

Logistics

- Midterm graded, returned at end of class
- HW3 due 1 week from today
- Lecture last Wed cancelled
- No class meeting this Wed, video lecture posted online

Review: HTML: HyperText Markup Language

- Language for describing *structure* of a document
- Denotes hierarchy of elements
- What might be elements in this document?

Review: HTML Example

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="main.css">
    <title>Prof Bell's Webpage</title>
</head>
<body>
<h1>
    Prof Jonathan Bell
</h1>
```

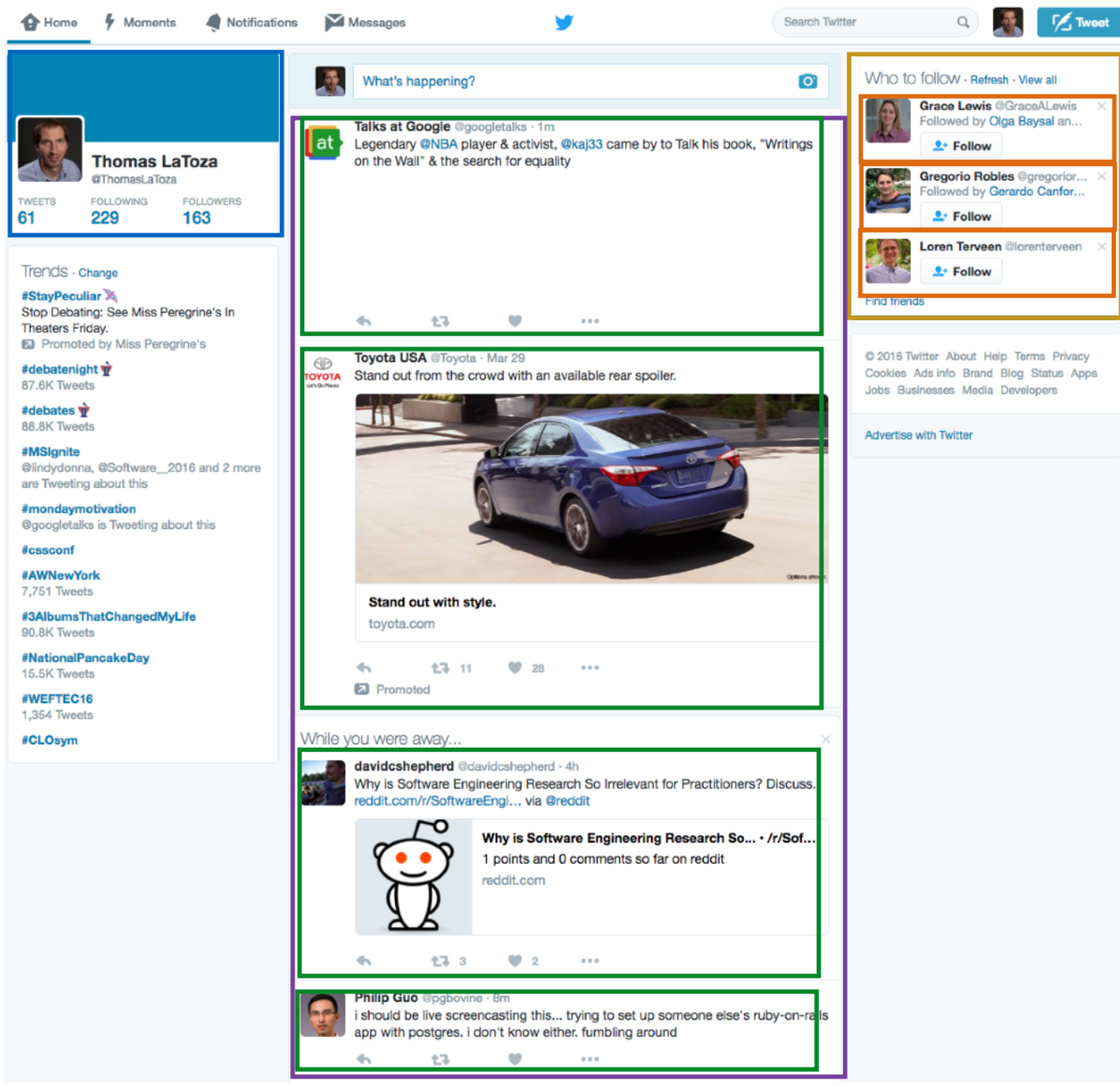
Use <h1>, <h2>, …, <h5> for headings

```
    <h2>Welcome, students!</h2>
        <p>
            <a href="https://www.youtube.com/watch?v=dQw4w9WgXcQ">See how to make
this page</a>
        </p>
    <h2>
        Some funny links
    </h2>
    <p>
        <ul>
            <li><a href="http://www.homestarrunner.com">Homestar Runner</a></li>
            <li><a
href="http://www.wb3w.net/The%20Original%20Hamsterdance.htm">Hamster Dance</a></li>
        </ul>
    </p>
    <h3>
        About Prof Bell
    </h3>
    <p>
        Prof Bell's office is at 4422 Engineering Building. His email address is <a
href="mailto:bellj@gmu.edu">bellj@gmu.edu</a>.
    </p>
    <p>
        Last updated: September 4th, 1999
    </p>
</div>
</body>
</html>
```

https://seecode.run/#-KQgR7vG9Ds7IUJS1kdq

Review: Components

- Web pages are complex, with lots of logic and presentation
- How can we organize web page to maximize modularity?
- Solution: Components
  - Templates that correspond to a specific widget
  - Encapsulates related logic & presentation using language construct (e.g., class)



Today

- REACT'ing to change
- Big demo of React

- If time
  - Extend demo in small groups

*React*ing to change

- What happens when state of component changes?
  - e.g., user adds a new item to list
- Idea
  1. Your code updates this.state of component when event(s) occur (e.g., user enters data, get data from network) using this.setState(newState)
  2. Calls to this.setState *automatically* cause *render* to be invoked by framework
  3. Reconciliation: Framework *diffs* output of render with *previous* call to render, updating only part of DOM that *changed*

What is state?

- All internal component data that, when changed, should trigger UI update
  - Stored as single JSON object this.state
- What isn't state?
  - Anything that could be computed from state (redundant)
  - Other components - should build them in render
  - Data duplicated from properties.

Properties vs. State

- Properties should be **immutable**.
  - Created through attributes when component is instantiated.
  - Should *never* update within component
  - Parent may create a new instance of component with new properties

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

- State **changes** to reflect the current state of the component.
  - Can (and should) change based on the current internal data of your component.

Working with state

- Constructor should initialize state of object

```
constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
```

- Use this.setState to update state

```
    this.setState({
      date: new Date()
    });
```

- Doing this will (asynchronously) eventually result in render being invoked
  - Multiple state updates may be batched together and result in a single render call

Partial state updates

- State is an object, may contain whatever properties you add
- Can use setState to update only parts of state you'd like to update

```
fetchPosts().then(response => {
  this.setState({
    posts: response.posts
  });
});

fetchComments().then(response => {
  this.setState({
    comments: response.comments
  });
});
```

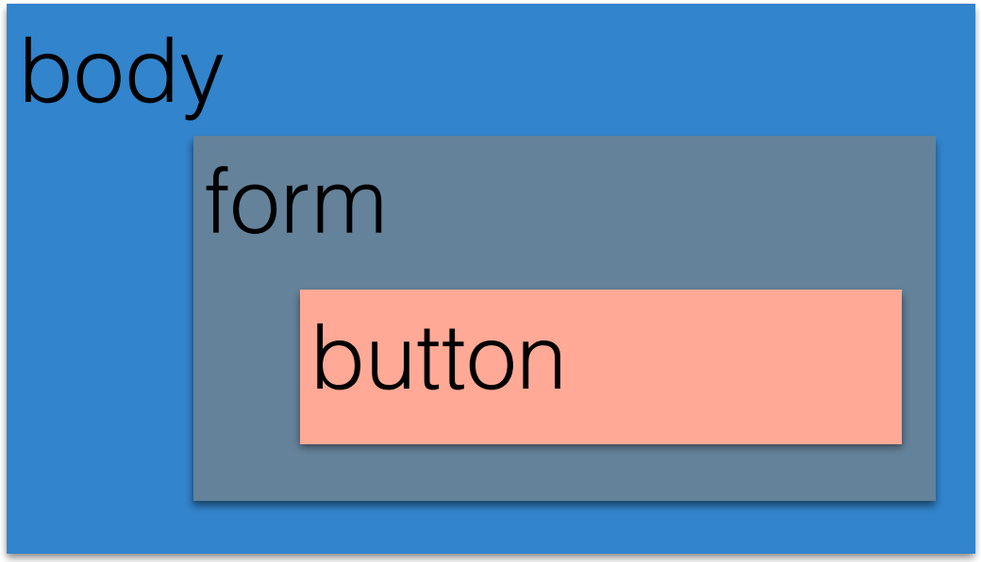
Handling events

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(prevState => ({ isToggleOn: !prevState.isToggleOn }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

ReactDOM.render(
  <Toggle />, document.getElementById('root')
);
```

https://reactjs.org/docs/handling-events.html

Event Dispatching

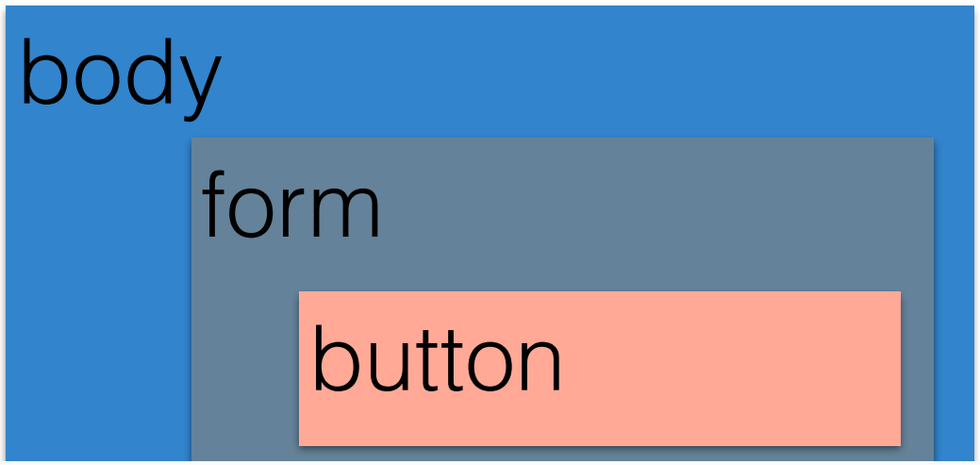
- Each event target can have (0…n) listeners registered for any given event type, called in arbitrary order
- What happens with nested elements?




```
Listener1: body onClick
Listener2: form onClick
Listener3: button onClick
```

What happens when we click in `button`?

Event Bubbling



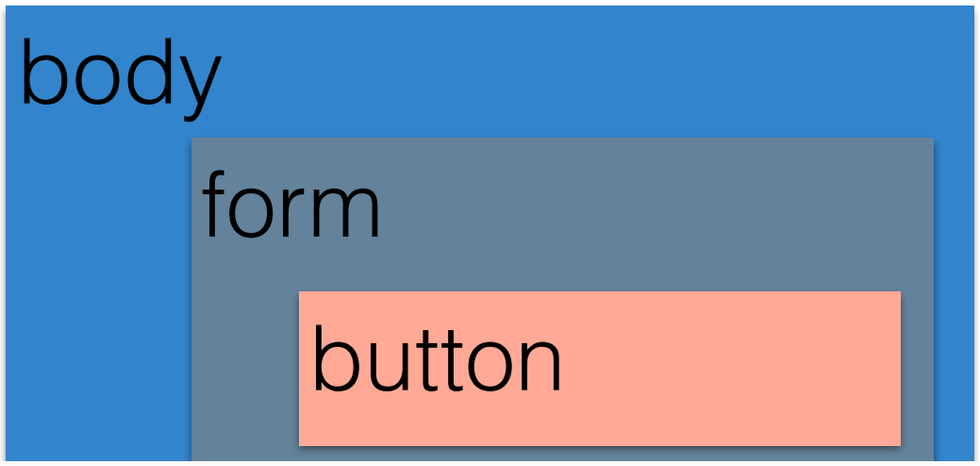
What happens when we click in **`button`**?

Called →

```
Listener1: body onClick
Listener2: form onClick
Listener3: button onClick
```

This is the default behavior

Event Capturing



What happens when we click in **`button`**?

Called →

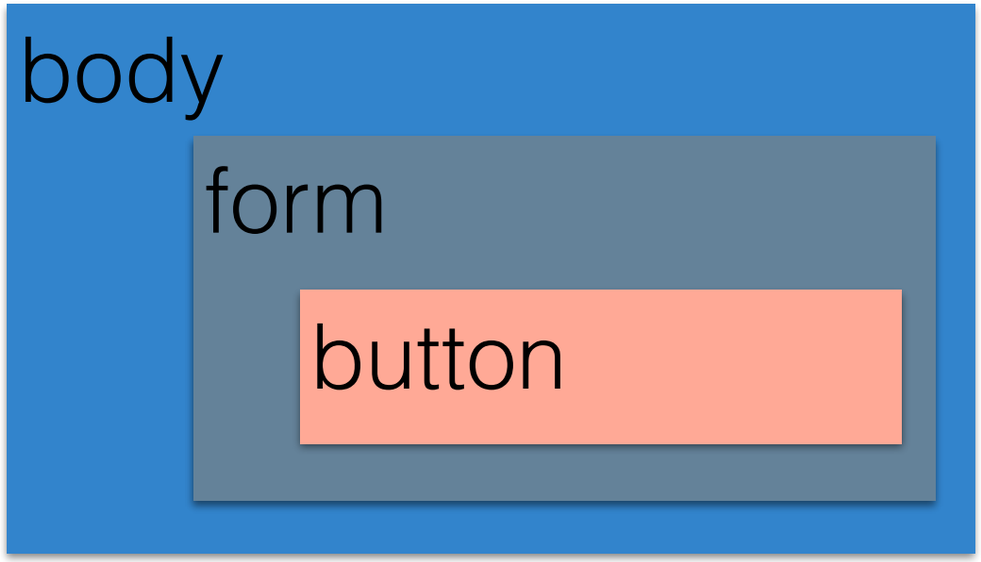
```
Listener1: body onClick
Listener2: form onClick
Listener3: button onClick
```

Enable event capturing when you register your listener:

```
element.addListener('click', myListener, true);
```

Event Dispatching

- An individual listener can stop bubbling/capturing by calling
- `event.stopPropagation();`
  - Assuming that `event` is the name of your handler's parameter




```
Listener1: body onClick
Listener2: form onClick
Listener3: button onClick
```

Functional components

- What happens if you have a simple component that **only** has properties and no state.
- Can use the function syntax to create a component.
- Component that is only a render method.

```
function Square(props) {
  return (
    <button className="square" onClick={props.onClick}>
      {props.value}
    </button>
  );
}
```

Nesting components

- UI is often composed of nested components
  - Like containers in HTML, corresponds to hierarchy of HTML elements
  - But…now each element is a React component that is generated
- Parent *owns* instance of child
  - Occurs whenever component instantiates other component in render function
  - Parent configures child by passing in properties through attributes

Nesting components

```
render() {
    return (
        <div>
            <PagePic pagename={this.props.pagename} />
            <PageLink pagename={this.props.pagename} />
        </div>
    );
}
```

Establishes ownership by creating in render function.

Sets pagename property of child to value of pagename property of parent

The data flows down

- State that is common to multiple components should be owned by a common ancestor
  - State can be passed into descendants as properties
- When this state can manipulated by descendants (e.g., a control), change events should invoke a handler on common ancestor
  - Handler function should be passed to descendents

https://reactjs.org/docs/state-and-lifecycle.html#the-data-flows-down

The data flows down

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
    this.state = {temperature: '', scale: 'c'};
  }

  handleCelsiusChange(temperature) {
    this.setState({scale: 'c', temperature});
  }

  render() {
    const scale = this.state.scale;
    const temperature = this.state.temperature;
    const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) : temperature;

    return (
      <div>
        <TemperatureInput
          scale="c"
          temperature={celsius}
          onTemperatureChange={this.handleCelsiusChange} />
      </div>
    );
  }
}
```

Single page app

- In a single page app, there is only one single HTML page loaded by browser.
- When new views are opened by user or data arrives from server, client side JavaScript code generates new views.

- How could you build a single page app using React?

What's wrong with this code?

```
class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = { seconds: 0 };
    this.interval = setInterval(() => this.tick(), 1000);
  }

  tick() {
    this.setState(prevState => ({
      seconds: prevState.seconds + 1
    }));
  }

  render() {
    return (
      <div> Seconds: {this.state.seconds} </div>
    );
  }
}

ReactDOM.render(<Timer />, mountNode);
```

Component lifecycle

```
class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = { seconds: 0 };
  }

  tick() {
    this.setState(prevState => ({
      seconds: prevState.seconds + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>
        Seconds: {this.state.seconds}
      </div>
    );
  }
}

ReactDOM.render(<Timer />, mountNode);
```

Component lifecycle

```
class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = { seconds: 0 };
  }

  tick() {
    this.setState(prevState => ({
      seconds: prevState.seconds + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>
        Seconds: {this.state.seconds}
      </div>
    );
  }
}

ReactDOM.render(<Timer />, mountNode);
```

ReactDOM.render(...)
[component created]
constructor(...)
render()
componentDidMount()

tick()
render()
...

[component rendered again by parent]
componentWillUnmount()
[component created]
...

Controlled Components

```
class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'Please write an essay about your favorite DOM element.'
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('An essay was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name: <textarea value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

https://reactjs.org/docs/forms.html

Controlled Components

- Single source of truth
- Whenever a control changes its value
  - React is notified
  - State is updated
- Whenever state is updated
  - If necessary, render function executes and generates control with new value

Reconciliation

```
<Card>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</Card>
```

```
<Card>
  <p>Paragraph 2</p>
</Card>
```

- Process by which React updates the DOM with each new render pass
- Occurs based on order of components
  - Second child of Card is destroyed.
  - First child of Card has text mutated.

https://reactjs.org/docs/reconciliation.html

Reconciliation with Keys

- Problem: what if children are dynamically generated and have their own state that must be persisted across render passes?
  - Don’t want children to be randomly transformed into other child with different state
- Solution: give children identity using keys
  - Children with keys will always keep identity, as updates will reorder them or destroy them if gone

Keys

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Todo in React

```
class App extends Component {
  constructor(props) {
    super(props);
    this.state = { items: [], text: '' };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={this.handleSubmit}>
          <input
            onChange={this.handleChange}
            value={this.state.text}
          />
          <button>
            Add #{this.state.items.length + 1}
          </button>
        </form>
      </div>
    );
  }
```

```
  handleChange(e) {
    this.setState({ text: e.target.value });
  }

  handleSubmit(e) {
    e.preventDefault();
    if (!this.state.text.length) {
      return;
    }
    const newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState(prevState => ({
      items: prevState.items.concat(newItem),
      text: ''
    }));
  }
}

class TodoList extends Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

In Class Activity

- Start with React todo demo app:
  - https://codesandbox.io/s/adoring-ishizaka-bf3ym?fontsize=14

- Add a “delete item” button
- Add an “edit” button that converts an item into a text field, then lets you save changes to it