CSE 331
Software Design & Implementation

Andrew Gies

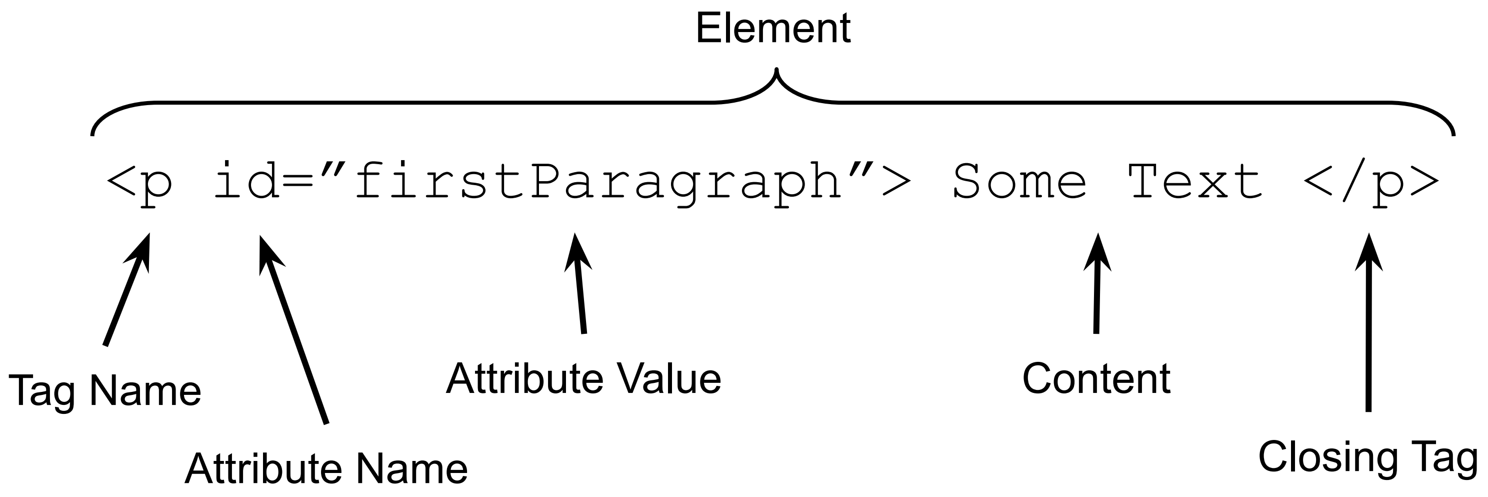
Summer 2019

React Overview

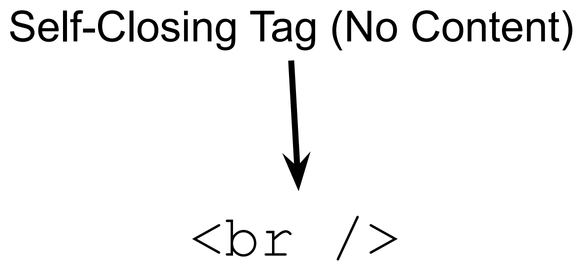
HTML, Formally

- HTML - Hypertext Markup Language
  - Not a full PL, describes document structure & content
- Consists mostly of *tags* and their contents
  - Each one has a beginning and end.
  - Can contain text (content) and other tags.
  - Each tag has a different meaning in the document.
  - Optional attributes (organized as key-value pairs)
    - Can think of them like “constructor parameters”: pieces of data that contain extra info about the tag.

Anatomy of a Tag



```
<p id="firstParagraph"> Some Text </p>
```



```
<br />
```

We'll see what <p> and <br> mean on the next slide...

A Few Useful Tags

- `<p>` - Paragraph tag, surrounds paragraph with whitespace/line breaks.
- `<div>` - “The curly braces of HTML” - used for grouping other tags. Surrounds its content with whitespace/line breaks.
- `<span>` - Like `<div>`, but no whitespace/line breaks.
- `<br />` - Forces a new line (like “\n”). Has no content.
- `<html>` and `<head>` and `<body>` - Used to organize a basic HTML document.
- `<script>` - Marks a section of non-HTML script code.
- LOTS of other tags for bullet point lists, pictures, buttons, text boxes, etc...
  - See the W3Schools HTML reference for a complete list, along with all their supported attributes.

Example 1:

Making a Clickable Button

```
<html>
  <head>
     <title>1. HTML5 Buttons</title>
  </head>
  <body>
     <script type="text/javascript">
        function sayHello() {
           alert("Hello, World!");
        }
     </script>
     <button onclick="sayHello()">Click Me!</button>
  </body>
</html>
```

JS Code is run whenever the button is clicked. In this case - just call a function that does the real "work".

Text displayed inside the button.

Example 2:

Drawing on a Canvas

- `<canvas>` tag: creates a blank drawing surface that you can “draw” on with JS
  - Create lines, shapes, draw images.
  - Has `width` and `height` attributes to determine the size of the drawing surface.
- We’re using `<canvas>` in HW8 and HW9 to draw lines/paths on top of images (like a map of campus!)
- Javascript is going to need some kind of Canvas object in order to call functions and draw pictures.
  - How do we get this object?

Modifying HTML with JS

- JS exists to allow webpages - meaning the HTML inside them, to change dynamically. So JS *has* to have a way to access/change the HTML tags.
- Implementation: Every HTML element has an associated JS object that the browser maintains.
  - Can get a reference in JS by using the “id” attribute.
  - Every tag can have an ID - value is a string that uniquely identifies an element.

HTML:

```
<p id=”thePar”>Hello, World!</p>
```

JS:

```
let parObj = document.getElementByID(“thePar”);
parObj.innerHTML = “Hello, 331!”;
```

Example 2 Code

```
<html>
  <head>
     <title>2. HTML5 Canvas</title>
  </head>
  <body>
     <script type="text/javascript">
        function drawSomething() {
           let canvas = document.getElementById("theCanvas");
           let context = canvas.getContext("2d");
           context.fillStyle = "teal";
           context.fillRect(50, 50, 150, 100);
        }
     </script>
     <button onclick="drawSomething()">Draw Something Cool</button>
     <br />
     <canvas id="theCanvas" width="500" height="500"></canvas>
  </body>
</html>
```

Making the Jump to React

- Previously, we've been writing HTML, then using a small amount of JS to interact with in.
- In React: Write mostly JS, which is responsible for dynamically generating the webpage on-the-fly.
  - Fundamentally different way of thinking about websites.
  - Allows code reuse (more or less impossible in HTML)
  - Improves modularity.
  - Designed to reduce coupling, increase cohesion. (Yay!)
- Code looks different than what we've seen so far.

Starting React Theory

- The webpage is made up of *Components*: these act like fancy tags:
  - Can contain other components
  - Have attribute-like things (slightly different, we'll see in the next lecture how they work).
  - Can also contain all kinds of JS code and application data (this is the powerful thing about components).
  - Decides what it "looks like" when actually placed on the webpage.
    - Expressed in terms of other components and regular HTML tags.
- Create a component by creating a JS class that extends the `Component` class (provided by React)

Example 3:

React Boilerplate

- A simple “Hello World” application in React
  - Demonstrates all the “starting” code required to get React up-and-running.
  - Uses React’s dialect of Javascript called JSX

- React needs a “starting point” to work with when creating that application. We use index.html and index.js as that starting point.
  - “index” is traditionally used as the name of the starting point of any website. React looks for files with this name by default.
- The actual application traditionally starts in App.js

Basics of JSX

- Write HTML tags directly inside the JS code - can be treated like JS objects and put in variables, passed to functions, etc...
- Inside the “HTML”, use curly braces to switch back to Javascript - can write any expression, the value is replaced into the HTML:

```
<p>The meaning of life is {6 * 7}.</p>

let idVariable = “paragraph-element”;
<p id={idVariable}>Ima Paragraph!</p>
```

Example 4:

React Canvas

Let's convert our previous canvas code from Example 2 to use React!
Step by step from Example 3:

1. Change the `<p>` element to a `<canvas>`
2. Need to get a canvas object to draw like last time: different in React.
    a. It's React's job to manage the HTML for us, grabbing something with an ID defeats that purpose and can cause bugs.
    b. In React, we use "Ref" objects instead of ID strings, but they work similarly.
3. Write an `updateCanvasImage()` method to draw a rectangle on the canvas like before.
4. Use `componentDidMount()` to find out when React is ready for us to start drawing things, then call `updateCanvasImage()`

Example 5:

Static State

- Each component is an instance of an object, so it can have whatever instance variables it wants.
- React has a special meaning for `this.state`, however.
  - Usually, state contains an entire object inside it, which can contain any number of other variables - no limit on the amount of data inside it.
  - Any data that has an effect on what a component looks like should be stored inside state.
    - * Well, almost. It should either be inside state or inside “props” - but we haven’t seen props yet. (Coming in Example 7)
  - Can be set like a normal variable only inside the constructor during initialization.
    - To change it outside the constructor, use the `setState` method. (We’ll see this in Example 6)

Example 6:

Changing State with Buttons!

Going to use buttons (same buttons we've seen before) to dynamically change the state:

1. Put the `<canvas>` inside a `<div>` so we can add things to our component.
2. Add a few `<button>`s to the component next to the canvas.
3. Write a few functions to be the click functions of all the buttons.
   a. Note: In React, `onclick` was renamed to `onClick` and works slightly differently. Pass it a function object which is then called, instead of just JS code inside a string.\
   b. For reasons we'll discuss later, need to use the "arrow function" syntax instead of the regular function syntax.
4. Call `this.setState()` inside the onclick functions to change our state.
5. Use `componentDidUpdate()` to be notified of when the state has changed.

React's Rules about State

- **Do not modify state without `setState`** please :)
  - setState does more than just update the variable, it also tells React what you're changing so React can do its job
- State updates are requests - the update is **NOT** guaranteed to have completed when `setState` returns.
  - This means if you `setState` and immediately try to use it, this.state probably doesn't have the new value yet.
  - This is what `componentDidUpdate` is for - React will let you know when state has changed so you can redraw your component.
  - Rule of Thumb: treat state as if it's write-only unless you know you're currently inside `render`, `componentDidUpdate`, or something called by one of those two.

Up Next

- Props: What are they? How are they used?
- Higher-level react theory
- Breaking our demo application into reasonable modules.
- HW8 introduction and overview.

Questions?

Props: The *Other* Kind of Data

- Inside the component: show up as properties of a `props` object that's passed into the constructor.
- Outside the component: passed to a component using a syntax similar to HTML attributes.
- Read-only inside the component.
- Changes (from the parent) trigger a component update just like state changes.
- **Main Idea:**
  - State: the data is owned by the component itself.
  - Props: the data is owned by the component's parent.

Example 7:

Introducing `ColorTitle`

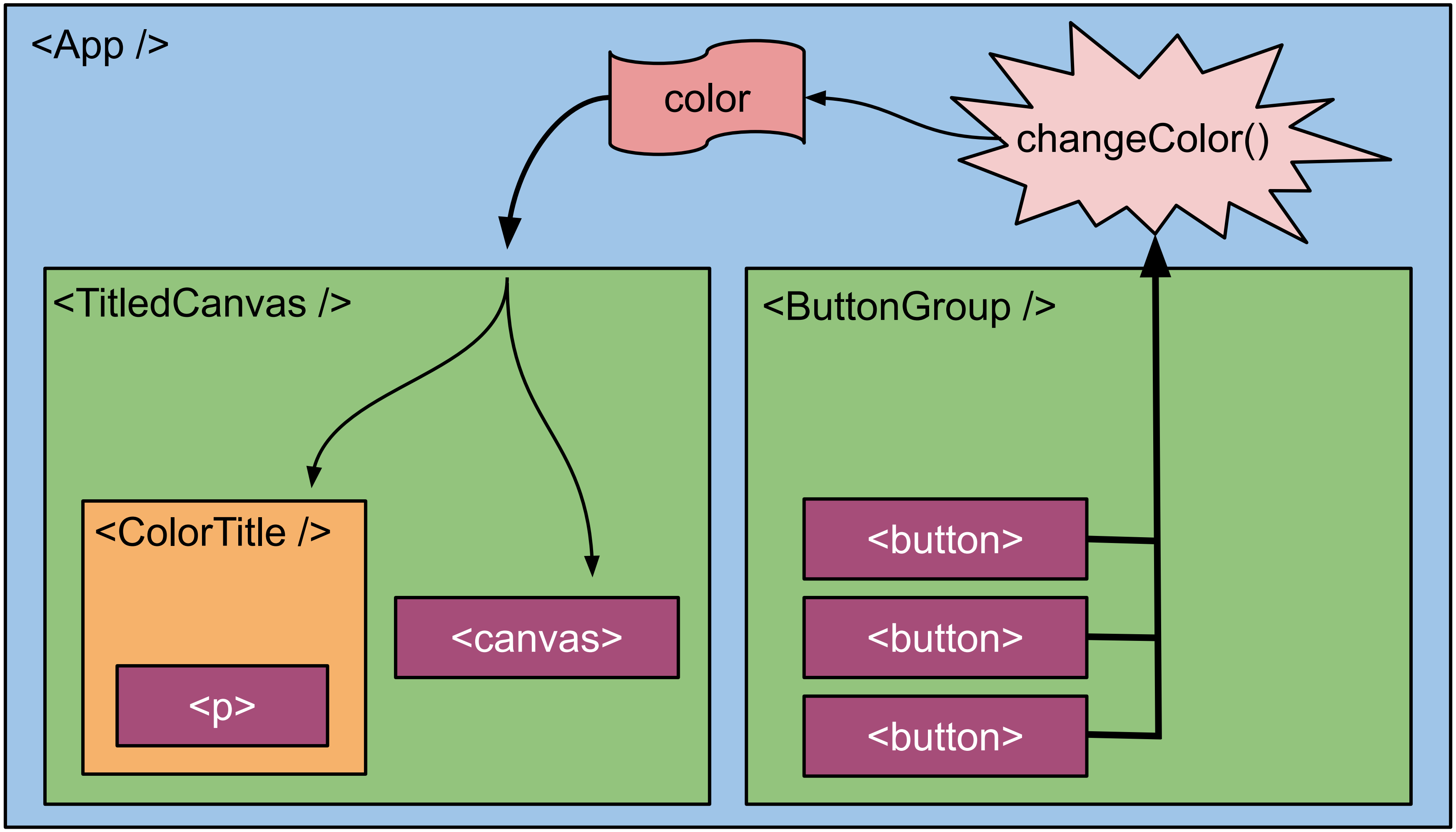
1. Let's create a new component: simply encompasses a piece of text with the current color.
   a. *In a real application, probably isn't something that makes sense to be its own component, but it's a good example.*
2. Need to somehow get the data from the `App` component inside `ColorTitle`
   a. Completely different classes/objects, so they can't just share variables.
3. Add a "color" attribute to our `<ColorTitle />` declaration, which becomes a prop inside the ColorTitle component.
4. When the "color" prop's value is changed, React automatically re-renders the component.

Example 8:

Putting it all Together

- Lots changes in this next example: but not much is new material.
- Add two other components to create a hierarchy.
- The main data, what color is currently selected, is stored in the parent (`<App />`). Passed as props to children.
- Use a callback from ButtonGroup to modify the current color.

The *Flow*



Summary

- Components are reusable blocks of code that allow modular design and proper cohesion.
- Components contain other components and HTML tags to determine how they appear on a webpage.
  - React is responsible for managing the underlying webpage.
- Data owned/controlled by a component is stored it that component's state.
- Data flows *down* from parent to child through props.
- Data flows *up* from child to parent through callbacks from the child into the parent's code.
- React notifies components of changes to their data through lifecycle methods, like `componentDidUpdate`

Objects (2)

- You can add properties

```
character.instrument = "horn"
```

- Bracket notation can be used to reference properties

```
character["instrument"] = "saxophone"
```

- And properties can be deleted

```
delete character.age
```

- Property names can be computed

```
what = "instrument"
character[what] = "tenor sax"
```

- See tutorials or references for more variations

Aside: JSON – JavaScript Object Notation

- A lightweight text format for reading and writing object data as text strings (to store in files, transmit on a network, etc.)
- Basic idea is to encode objects with a syntax very similar to JavaScript object notation. For our contact:

```
{ "name": "Lisa Simpson", " age": 7,
  "instrument": "saxophone",
  "smart": true }
```

- JSON libraries can turn JavaScript objects into JSON strings (JSON.stringify) and can parse these strings to turn them back into objects (JSON.parse)
  – See www.json.org for docs and details

Objects with methods

- Properties in a JavaScript object can include methods (functions) – just like in Java

```
let account = {
  owner: "Gandalf",
  balance: 10000,
  deposit: function(amount) {
    this.balance += amount
  }
}
```

- We call methods in the expected way:

```
account.deposit(100);
```

Objects with methods

- There is a bit of shorthand available. Instead of

```
let account = {
  …
  deposit: function(amount) {
     this.balance += amount;
  }
}
```

we can write

```
let account = {
  …
  deposit(amount) {
     this.balance += amount
  }
}
```

but the meaning is exactly the same

Creating new objects

- JavaScript has an unconventional model: it is an object-oriented language, but there are *no classes*!
  - But everything* is an object, including functions(!)

    *modulo some technical details

- JavaScript's basic model is that objects are created by functions that return new objects…
  - All objects are related to some other object by their hidden "prototype" property
  - When we look for a property in an object, if it is not found locally, we look in its prototype object, and if not found, in that object's prototype, … until we either find it or hit the top of the chain
  - So we have something that resembles inheritance but without classes

ES6 Classes

- All of this is a bit much, so ES6 added syntax for “classes”

-
  ```
  class Account {
    constructor(owner, balance) {
      this.owner = owner
      this.balance = balance
    }
    deposit(amount) {
      this.balance += amount
    }
  }
  ```
- But underneath there are only objects with prototypes pointing to other objects
  - We’ll ignore the details – see a good JS reference

What’s next?

- How JavaScript code interacts with html elements in a web page
- Then some React basics and how to structure a JavaScript application for hw8 and hw9
- Don’t miss upcoming sections and lectures!