JS II: client-side applications

Exhuming the terror of JavaScript

JavaScript: The Evil Parts

for (p=0;p<2;p++)
document.write(
eval("publisher_"+p))

THE HTML DOESN'T NEED TO LOAD JS AND CSS

IF THE HTML AND CSS ARE IN JS
Review: JavaScript Part 1

• JavaScript is a prototypical object-oriented language with functional and imperative features
  • Everything is an object in JavaScript (including functions)
  • You can create objects using “new NameOfConstructorFunction(...);”
  • Objects “inherit from” a prototype object associated with that class’s constructor function
    function Book(x) { this.author = x; };  b1 = new Book();
    “Book” is itself an object (it inherits the Function prototype)
    B1 is a Book object (it inherits the Book prototype)

• JavaScript uses an event queue to manage asynchronous events
  • Like browser events, timeouts, and networking events
Review: Exam-style Exercise

• What is the output of this code?

```javascript
function f() {
    console.log('beginning');
    function callback1() {
        console.log('callback1');
    }
    setTimeout(callback1, 1000); //1s
    console.log('middle');
    function callback2() {
        console.log('callback2');
    }
    setTimeout(callback2, 2000); //2s
    console.log('end');
}
f();
```
Review: Exam-style Exercise

• What is the output of this code?

```javascript
function f() {
  console.log('beginning');
  function callback1() {
    console.log('callback1');
  }
  setTimeout(callback1, 1000); //1s
  console.log('middle');
  function callback2() {
    console.log('callback2');
  }
  setTimeout(callback2, 2000); //2s
  console.log('end');
}
f();
```

Beginning
Middle
End
Callback1
Callback2

**Critical:** `setTimeout` *queues an event in the event queue*. It returns immediately

If function `f()` is called at time 0, then `callback1` is scheduled to run at time 1000, and `callback2` at time 2000 (jitter and precision notwithstanding)
Review: JavaScript Event Queue

• Much like the runtime stack and heap, JavaScript’s runtime environment includes an event queue for managing asynchronous events

• **Consider**: What would happen if we didn’t have an event queue to manage UI events?
  • Click a button, send HTTP request, *wait for response*, then let the UI update

• The event queue is a hack that allows us to design UIs without hanging and without multithreading (JS is single-threaded)
We have already seen how we can **modify the DOM** by placing random strings of HTML into various elements

- Document API: `document.getElementById("blah").innerHTML = "<p>Hi</p>"`

We can construct **arbitrarily complicated DOM subtrees** by creating DOM tree nodes manually and modifying DOM node relationships

- Create a node, assign it as a child of another node, etc.

We can use the **fetch** API to generate HTTP requests within JavaScript to reach REST API endpoints

- Then, we can use the JSON returned to inform our DOM modifications

We use frameworks like **React** to simplify DOM modifications

- They’re **slow** – Reach uses a **virtual DOM** to speed up batches of changes
Outline

• **Review: event-driven programming**

• Building DOM in JavaScript from JSON

• Fetch API and Promises

• Closures

• Frameworks
Review: Event-driven programming

let count = 0;
function callback() {
    count++;
    if (count < 2) {
        setTimeout(callback, 0);
    }
    console.log(count);
}

setTimeout(callback, 0);
// what will print?
Common mistake: for-in loops

• **for-in loops** often yield unexpected results
  • They iterate "up the prototype chain"

```javascript
> const chickens = ['Magda', 'Marilyn', 'Myrtle II'];
> for (let chicken in chickens) {
>   console.log(chicken);
> }
1
2
3
```

• ES6's for-of loops are nice, but are hard to analyze statically, so some style guides do not allow them

```javascript
> for (let chicken of chickens) {
>   console.log(chicken);
> }
Magda
Marilyn
Myrtle II
```
Iteration with `forEach` and `map`

- **`forEach` loops "do the right thing"
  - Behave like other programming languages (C, C++, Perl, Python ...)
  - We'll learn about the `=>` syntax soon (it's an anonymous function)

```javascript
const chickens = ['Magda', 'Marilyn', 'Myrtle II'];
chickens.forEach((chicken) => {
  console.log(chicken);
});
```

- **`map` is another nice option**
  - Use it to transform an array into another array

```javascript
const chickens_say = chickens.map(chicken => (`${chicken} says cluck`));
console.log(chickens_say);
// [ 'Magda says cluck', 'Marilyn says cluck', 'Myrtle II says cluck' ]
```
JavaScript: functional programming

• In **functional programming**, execution occurs **by composing functions** together *rather than* executing sequences of statements.

• Example:

```javascript
function myCalcImperative() {
    let x = [1, 2, 3, 4, 5];
    let result = 0;
    for (let i=0; i<x.length; i++) {
        if (x[i] % 2 === 0) {
            result += x[i] * 10;
        }
    }
    return result;
}
```

```javascript
function myCalcFunctional1() {
    let x = [1, 2, 3, 4, 5];
    return x.filter( n => n % 2 === 0)
        .map(a => a * 10)
        .reduce( (a,b) => a + b );
}
```
JavaScript: functional programming

• In **functional programming**, execution occurs by composing functions together *rather than* executing sequences of statements.

• Example (alternate):

```javascript
function myCalcFunctional2() {
  let x = [1, 2, 3, 4, 5];
  let result = 0;

  let filteredX = [];
  x.forEach(n => if (n % 2 == 0) filteredX.push(n));
  filteredX.forEach(n => n *= 10);
  filteredX.forEach(n => result += n);
  return result;
}
```
JavaScript: Functional programming??

• Consider this scenario:
  • User clicks a button
  • Browser sends a POST request
  • Browser waits for server response
  • Browser renders changes in DOM tree

• Looks something like this in JavaScript:

```javascript
document.getElementById("myButton").onclick = function () {
  makePostRequest("api.lol.com/api/path",
                  {"subject": "hello", "text": "world"},
                  onSuccessFunction,
                  onFailureFunction);
}
```
JavaScript: Functional programming??

• Consider this scenario:
  • User clicks a button
  • Browser sends a POST request
  • Browser waits for server response
  • Browser renders changes in DOM tree

• Looks something like this in JavaScript:

```javascript
document.getElementById("myButton").onclick = function () {
  makePostRequest("api.lol.com/api/path", {
    "subject": "hello", "text": "world"},
    function( data ) { /* do some changes to DOM based on data */},
    function( data ) { /* do some failure handling */};
}
```
JavaScript: Functional programming

• **Note**: the onclick function handler runs when the button gets clicked
  • *We don’t want the browser to “freeze” when the user clicks the button*
  • **Instead**, we *schedule events* in the event queue to occur:
    • Immediately make a POST request
    • Schedule a onSuccessFunction to run when the request finishes
    • Schedule a onFailFunction to run if the request fails (e.g., server doesn’t respond)
  • This way, we don’t have to wait around for the server on every UI event

```javascript
document.getElementById("myButton").onclick = function () {
    makePostRequest("api.lol.com/api/path",
    {
        "subject": "hello", "text": "world"},
    onSuccessFunction, onFailFunction);
};
```
Outline

• Review: event-driven programming

• Building DOM in JavaScript from JSON

• Fetch API and Promises

• Closures

• Frameworks
DOM Review

```
<html>
  <head></head>
  <body>
    <div id="content">
      <p>Drew DeOrio</p>
      <p>Jason Flinn</p>
    </div>
  </body>
</html>
```
Overview

• Turn JSON data into a webpage that looks like this:

```json
[{
   "url": "/u/1/",
   "username": "awdeorio"
},
{
   "url": "/u/2/",
   "username": "jflinn"
}]
```

```html
<html>
  <head></head>
  <body>
    <div id="content">
      <p class="user">awdeorio's link <a href="/u/1/">here</a></p>
      <p class="user">jflinn's link <a href="/u/2/">here</a></p>
    </div>
  </body>
</html>
```
myObj.forEach(user => {
    // make <p>
    let myNewParagraph = document.createElement('p');
    // make it <p class="user">
    myNewParagraph.setAttribute('class', 'user');

    //Make the string to go inside
    let myString = document.createTextNode(`User "+user.username+"'s link `);

    // Make the link
    let newLink = document.createElement('a');
    newLink.setAttribute('href', user.url);
    // make Link text "here"
    let newLinkText = document.createTextNode("here");

    // put link text in link
    newLink.appendChild(newLinkText);

    // put myString and newLink inside the paragraph
    myNewParagraph.appendChild(myString);
    myNewParagraph.appendChild(newLink);

    // append this DOM subtree into the content Div
    myDivElement.appendChild(myNewParagraph);
});
Outline

• Review: event-driven programming

• Building DOM in JavaScript from JSON

• Fetch API and Promises

• Closures

• Frameworks
Fetch

- JavaScript provides a “fetch” API for making HTTP requests
- These do not require browser reloads

- Idea: use fetch to make requests of a REST API
  - API responds with JSON object
  - Use JSON to update DOM in some way

- Example: client request “fetch post ID 50”
  Server responds OK “{ ‘author’ : ‘kevin’, ‘content’: ‘hello’}”
Fetch API

• How JavaScript requests data from a REST API
• JavaScript function that loads data from the server
• Returns a "promise"

```javascript
fetch("https://dijkstra.eecs.umich.edu/kleach/eecs485/su20/s/users.json")
    .then(handleResponse)
    .then(handleData);
```
Promises

• Promise: a function that returns some time in the future
  • Put a function call on the event queue

• Needed for fetch because request/response is not instant
  • Recall: JavaScript is single-threaded. We don’t want the UI to freeze up while the request is being serviced

```javascript
fetch("https://dijkstra.eecs.umich.edu/kleach/eecs485/su20/s/users.json")
  .then(handleResponse)
  .then(handleData);
```
Outline

• Review: event-driven programming

• Building DOM in JavaScript from JSON

• Fetch API and Promises

• Closures

• Frameworks
Closures

```javascript
function outer() {
    let x = 0;
    function inner() {
        x++;
        console.log(x);
    }
    return inner;
}

let f = outer();
f();
f();
f();

Notice: f is a function pointer to “inner” (the thing returned by “outer”)
Closures

• Notice that the inner function has access to outer function's variables
• Lexically scoped name binding
• This is called a closure

```javascript
function showUsers() {
    const entry = document.getElementById('JSEntry');
    //...
    function handleData(data) {
        //...
        entry.appendChild(node);
    }
    fetch(/* ... */); // ...
}
```
Closures

function showUsers() {
    const entry = document.getElementById('JSEntry');

    function handleResponse(response) { /*... */ }

    function handleData(data) {
        // ...
        entry.appendChild(node);
    }

    fetch(/*...*/)
        .then(handleResponse)
        .then(handleData)
        .catch(error => console.log(error));
}

• The inner function has a longer lifetime than the outer function
• handleData() has access to entry even though showUsers() has already returned!
Closures in the interpreter

1. Objects created for entry, handleResponse, handleData

2. `fetch` function executes
   1. Enqueue callbacks `handleResponse`, `handleData`
   2. `fetch` returns before response arrives

3. .... wait for response

4. Later, response arrives and `handleResponse` executes

5. Later, JSON data is ready and `handleData` executes

```javascript
function showUsers() {
    const entry = /*...*/;

    function handleResponse(response) {
        /*...*/
    }

    function handleData(data) {
        // ...
        entry.appendChild(node);
    }

    fetch(/*...*/)
        .then(handleResponse)
        .then(handleData)
}
```
Why closures important in web development

• Callback Functions are everywhere
  • Event handlers: click
  • Promise handlers: .then()

• These functions need to remember their context
  • Remember that execution keeps going past a fetch() call
Anonymous functions

- These callback functions are used only once

```javascript
function showUsers() {
    const entry = document.getElementById('JSEntry');

    function handleResponse(response) {/*/... */}

    function handleData(data) {/*/...*/}

    fetch('/api/v1/users/')
        .then(handleResponse)
        .then(handleData);
}
```
Anonymous functions

• Refactor to use *anonymous functions*

```javascript
function showUsers() {
    const entry = document.getElementById('JSEntry');

    function handleResponse(response) {/*...*/}

    function handleData(data) {/*...*/}

    fetch('/api/v1/users/')
        .then(function(response) {
            //...
        })
        .then(function(data) {
            //...
        })
}
```
Anonymous functions

• Works exactly the same way as when the functions had names
• Also called a lambda function or function literal

```javascript
function showUsers() {
    const entry = document.getElementById('JSEntry');

    fetch('/api/v1/users/')
        .then(function(response) {
            //...
        })
        .then(function(data) {
            //...
    })
}
Anonymous functions in ES6

• ES6 provides a convenient syntax for anonymous functions
• "Arrow functions"

```javascript
function showUsers() {
    const entry = document.getElementById('JSEntry');

    fetch('/api/v1/users/')
        .then((response) => {
            //...
        })
        .then((data) => {
            //...
        })
}
```
Anonymous functions in ES6

• Anatomy of an anonymous function
  • Inputs
  • Body
  • Arrow

• Creates a function object on the heap
  • Just like "regular" functions

• Long format
  \[
  (\text{INPUTS}) \Rightarrow \{ \\
  \quad \text{// BODY} \\
  \}
  \]

• Short cut for body with one function call
  \[
  \text{INPUT} \Rightarrow \text{my\_function(INPUT)}
  \]
Outline

• Review: event-driven programming

• Building DOM in JavaScript from JSON

• Fetch API and Promises

• Closures

• Frameworks
A problem with raw JavaScript

• Large JavaScript applications quickly become unwieldy
• All functions act on the DOM, DOM acts like a giant global variable
• Difficult to decompose program into abstractions

```javascript
function showUser() {
    fetch()
        .then(function(data) {
            const users = data.results;
            users.forEach((user) => {
                const node = document.createElement('p');
                // ...
                node.appendChild(textnode);
                entry.appendChild(node);
            });
        });
    // ...
```
React

• React is a framework built by Facebook
• Build encapsulated components that manage their own state
  • Compose them to make complex UIs
• Efficient updates to the DOM

• https://reactjs.org/
React

• Components
  • Functional: usually stateless
  • Class-type: usually stateful

• Tree of composable components -> DOM
Virtual DOM

• Components rendering cause other components to render
• This would cause lots of DOM updates, which are slow
  • Because the actual screen changes
• Solution: a Virtual DOM

• Periodically reconcile virtual DOM with real DOM
  • Avoids unnecessary changes

• For lots of details: https://reactjs.org/docs/reconciliation.html
React documentation

• Required reading for project 3: 
  https://reactjs.org/docs/hello-world.html

• Live example 
  https://codepen.io/awdeorio/pen/yzXjzZ?editors=1010