Class 10: Review and Pairing

This week, students have been working with AJAX and asynchronous programming in addition to the functional programming concepts: map, filter, and reduce. These concepts were introduced in conjunction with SQL, in order to assist students in understanding the model layer of an MVC application.

Code Reviews

Trivia Crossroads

- Crossroads
    - Demo
    - Code
- Cup Game
- Black History Ipsum
- Echelon
- Web WD
- Preps
- Spirit Animal
- The Fellowship
- Vote Tracker
- Swanson
- Mortality
- Mars Site Voting
- Note Fellows

Class 06: AJAX and WRRC

During this class, you will focus on teaching students the underlying concepts of how the web works. Starting with an introduction to the WRRC (web request response cycle), you will demonstrate how HTTP works and how to accomplish simple GET requests through the use of jQuery. Students will need to understand the 3 major parts of every request (url, method, headers) and the 3 major parts of every response (status, headers, body).

Demo #1

Complete AJAX call

$.ajax() is the most configurable one

```
$.ajax({
    url: 'data/hackerIpsum.json',
    type: 'GET',
    dataType: 'json',
    success: function(data) {
      Article.loadAll(data);
      Article.all.forEach(function(a) {
        $('#articles').append(a.toHtml())
      });
      console.log(data);
    },
    error: function() {
      console.log("it failed");
    }
 });
```

Shorthand AJAX call

```
$.ajax('data/hackerIpsum.json', {
    success: function(data) {
    Article.loadAll(data);

  Article.all.forEach(function(a) {
    $('#articles').append(a.toHtml())
  });

  console.log(data);
  },
  error: function() {
    console.log("it failed");
  }
});
```

Shorthand version

It should be noted that all $.get(), $.post(), .load() are all just wrappers for $.ajax() as it's called internally.

$.get()

-- defaults type to get

```
$.get('data/hackerIpsum.json', function(data) {

  Article.loadAll(data);

  Article.all.forEach(function(a) {
    $('#articles').append(a.toHtml())
  });

  console.log(data);

});
```

$.getJSON()

-- defaults type to get

-- defaults dataType JSON

```
$.getJSON('data/hackerIpsum.json', function(data) {

    Article.loadAll(data);

  Article.all.forEach(function(a) {
    $('#articles').append(a.toHtml())
  });

  console.log(data);

});
```

$.load()

`.load()` is similar to `$.get()` but adds functionality which allows you to define where in the document the returned data is to be inserted. Therefore really only usable when the call only will result in HTML. It is called slightly differently than the other, global, calls, as it is a method tied to a particular jQuery-wrapped DOM element.

Example #1

```
<div id="divTestArea1"></div>


<script src="https://code.jquery.com/jquery-3.1.0.min.js" integrity="sha256-
cCueBR6CsyA4/9szpPfrX3s49M9vUU5BgtiJj06wt/s=" crossorigin="anonymous"></script>




<!-- Example #1 -->
<script type="text/javascript">
    $(function() {
        $("#divTestArea1").load("content.html");
    });
</script>
```

Example #2

A pretty cool trick is that you can actually pass a selector along with the URL, to only get a part of the page. In the first example, we loaded the entire file, but in the following example, we will only use the div, which contains the first sentence:

```
<!-- Example #2 -->
<div id="divTestArea2"></div>
<script type="text/javascript">
$(function() {
    $("#divTestArea2").load("content.html #divContent");
});
</script>
```

Deferred Methods

Chaining Promises

```
$.get('data/rawData.json', function(data) {
    Article.loadAll(data);

    Article.all.forEach(function(a) {
      $('#articles').append(a.toHtml())
    });

    console.log(data);

  })
  .done(function(){
    console.log("im done");
  })
  .always(function(){
    console.log("always!")
  });
```

Last Demo

```
Article.fetchAll = function() {
  if (localStorage.rawData) {
    // When rawData is already in localStorage,
    // we can load it by calling the .loadAll function,
    // and then render the index page (using the proper method on the
articleView object).
    var storedDataString = JSON.parse(localStorage.getItem('rawData'));
    Article.loadAll(storedDataString)//TODO: What do we pass in here to the
.loadAll function?

    articleView.initIndexPage()//(); //TODO: Change this fake method call to the
correct one that will render the index page.
    } else {
   // TODO: When we don't already have the rawData, we need to:
    // 1. Retrieve the JSON file from the server with AJAX (which jQuery method
is best for this?),
    $.ajax({
        type: 'GET',
        url: 'data/ipsumArticles.json',
          success: function (data) {
            // 2. Store the resulting JSON data with the .loadAll method,
            Article.loadAll(data);
            // 3. Cache it in localStorage so we can skip the server call next
time,
            localStorage.setItem('rawData', JSON.stringify(data));
            // 4. And then render the index page (perhaps with an articleView
method?).
            articleView.initIndexPage();

            console.log(data);
          }
      });
  }
}
```

Bonus if time

The ETag or entity tag is part of HTTP, the protocol for the World Wide Web. It is one of several mechanisms that HTTP provides for web cache validation, which allows a client to make conditional requests. This allows caches to be more efficient, and saves bandwidth, as a web server does not need to send a full response if the content has not cha

```
// This function will retrieve the data from either a local or remote source,
// and process it, then hand off control to the View.
```

```
Article.fetchAll = function() {
  function fetchFromDisk(){
    // DONE: When we don't already have the rawData, we need to:
    // 1. Retrieve the JSON file from the server with AJAX (which jQuery method
is best for this?),
    console.log('using ajax');
    $.getJSON( './data/hackerIpsum.json', function( data ) {
      // 2. Store the resulting JSON data with the .loadAll method,
      Article.loadAll(data);

      // 3. Cache it in localStorage so we can skip the server call next time,
      localStorage.setItem('rawData', JSON.stringify(data));

      // DONE. 4. And then render the index page (perhaps with an articleView
method?).
      articleView.initIndexPage();
    });
  }

  function fetchFromLocalStorage(){
    // When rawData is already in localStorage,
    // we can load it by calling the .loadAll function,
    // and then render the index page (using the proper method on the
articleView object).
    //DONE: What do we pass in here to the .loadAll function?
    console.log('using local storage');
    var rd = localStorage.getItem('rawData');
    var rdjson = JSON.parse(rd);
    Article.loadAll(rdjson);

    //DONE: Change this fake method call to the correct one that will render the
index page.
    articleView.initIndexPage();
  }

  $.ajax({
    url: './data/hackerIpsum.json',
    method: 'HEAD',
    success: function(data, message, xhr) {
      console.log('xhr', xhr);
      var etag = xhr.getResponseHeader('ETag');
      console.log('etag', etag);
      if (localStorage.etag){
        var locEtag = localStorage.getItem('etag');
        if (locEtag === etag && localStorage.rawData) {
          console.log('etag matches and in local storage');
          fetchFromLocalStorage();
```

```
        } else {
          fetchFromDisk();
        }
      } else {
        fetchFromDisk();
    }
      localStorage.setItem('etag', etag);
    }
  });
}
```

Functional Programming

For-Each

Since JavaScript lets us pass a function as a parameter to another function, writing a forEach function is relatively straightforward:

```
function forEach(callback, array) {
    for (var i = 0; i < array.length; i = i + 1) {
        callback(array[i], i);
    }
}
```

This function takes another function, callback, as a parameter and calls it on every item in the array.

Now, with our example, we want to run the addColour function on each item in the array. Using our new forEach function we can express that intent in just one line:

```
forEach(addColour, colours);
```

Calling a function on every item in an array is such a useful tool that modern implementations of JavaScript include it as a built in method on arrays. So instead of using our own forEach function, we could use the built in one like so:

```
var colours = [
    'red', 'orange', 'yellow',
    'green', 'blue', 'purple'
];

colours.forEach(addColour);
```

Map

Now, our forEach function is handy, but somewhat limited. If the callback function we pass in returns a value, forEach just ignores it. With a small adjustment, we can change our forEach function so that it gives us back whatever value the callback function returns. We would then have a new array with a corresponding value for each value in our original array.

Let's look at an example. Say we have an array of IDs, and would like to get the corresponding DOM element for each of them. To find the solution in a 'procedural' way, we use a for-loop:

```
var ids = ['unicorn', 'fairy', 'kitten'];
var elements = [];
for (var i = 0; i < ids.length; i = i + 1) {
    elements[i] = document.getElementById(ids[i]);
}
// elements now contains the elements we are after
```

Again, we have to spell out to the computer how to create an index variable and increment it—details we shouldn't really need to think about. Let's factor out the for-loop like we did with forEach and put it into a function called map:

```
var map = function(callback, array) {
    var newArray = [];
    for (var i = 0; i < array.length; i = i + 1) {
        newArray[i] = callback(array[i], i);
    }
    return newArray;
}
```

Now we have our shiny new map function, we can use it like so:

```
var getElement = function(id) {
  return document.getElementById(id);
};

var elements = map(getElement, ids);
```

The map function takes small, trivial functions and and turns them into super-hero functions—it multiplies the function's effectiveness by applying it to an entire array with just one call.

Like forEach, map is so handy that modern implementations have it as a built-in method for array objects. You can call the built-in method like this:

```
var ids = ['unicorn', 'fairy', 'kitten'];
var getElement = function(id) {
  return document.getElementById(id);
};

var elements = ids.map(getElement);
```

Reduce

Now, map is very handy, but we can make an even more powerful function if we take an entire array and return just one value. That may seem a little counter-intuitive at first—how can a function that returns one value instead of many be more powerful? To find out why, we have to first look at how this function works.

To illustrate, let's consider two similar problems:

- Given an array of numbers, calculate the sum; and
- Given an array of words, join them together with a space between each word.[1]

Now, these might seem like silly, trivial examples—and they are. But, bear with me, once we see how this reduce function works, we'll apply it in more interesting ways.

So, the 'procedural' way to solve these problems is, again, with for-loops:

```
// Given an array of numbers, calculate the sum
var numbers = [1, 3, 5, 7, 9];
var total = 0;
for (i = 0; i < numbers.length; i = i + 1) {
    total = total + numbers[i];
}
// total is 25

// Given an array of words, join them together with a space between each word.
var words = ['sparkle', 'fairies', 'are', 'amazing'];
var sentence = '';
for (i = 0; i < words.length; i++) {
    sentence = sentence + ' ' + words[i];
}
// ' sparkle fairies are amazing'
```

These two solutions have a lot in common. They each use a for-loop to iterate over the array; they each have a working variable (total and sentence); and they both set their working value to an initial value.

Let’s refactor the inner part of each loop, and turn it into a function:

```
var add = function(a, b) {
    return a + b;
}

// Given an array of numbers, calculate the sum
var numbers = [1, 3, 5, 7, 9];
var total = 0;
for (i = 0; i < numbers.length; i = i + 1) {
    total = add(total, numbers[i]);
}
// total is 25

function joinWord(sentence, word) {
    return sentence + ' ' + word;
}

// Given an array of words, join them together with a space between each word.
var words = ['sparkle', 'fairies', 'are', 'amazing'];
var sentence = '';
for (i = 0; i < words.length; i++) {
    sentence = joinWord(sentence, words[i]);
}
// 'sparkle fairies are amazing'
```

Now, this is hardly more concise but the pattern becomes clearer. Both inner functions take the working variable as their first parameter, and the current array element as the second. Now that we can see the pattern more clearly, we can move those untidy for-loops into a function:

```
var reduce = function(callback, initialValue, array) {
    var working = initialValue;
    for (var i = 0; i < array.length; i = i + 1) {
        working = callback(working, array[i]);
    }
    return working;
};
```

Now we have a shiny new reduce function, let's take it for a spin:

```
var total = reduce(add, 0, numbers);
var sentence = reduce(joinWord, '', words);
```

Like forEach and map, reduce is also built in to the standard JavaScript array object. One would

use it like so:

```
var total = numbers.reduce(add, 0);
var sentence = words.reduce(joinWord, '');
```

Filter

The filter() method creates a new array with all elements that pass the test implemented by the provided function.

filter() calls a provided callback function once for each element in an array, and constructs a new array of all the values for which callback returns a value that coerces to true. callback is invoked only for indexes of the array which have assigned values; it is not invoked for indexes which have been deleted or which have never been assigned values. Array elements which do not pass the callback test are simply skipped, and are not included in the new array.

Filtering out all small values

The following example uses filter() to create a filtered array that has all elements with values less than 10 removed.

```
function isBigEnough(value) {
    return value >= 10;
}

var filtered = [12, 5, 8, 130, 44].filter(isBigEnough);
// filtered is [12, 130, 44]
```

Another example:

```
var a = [5, 4, 3, 2, 1];
smallvalues = a.filter(function(x) { return x < 3 });   // [2, 1]
everyother = a.filter(function(x,i) { return i%2==0 }); // [5, 3, 1]
```

Another example:

Fortunately, in JavaScript, arrays have the handy filter method which we can use to do the filtering for us instead of manually looping through the array ourselves.

```
var sidekicks = [
    { name: "Robin",     hero: "Batman"   },
    { name: "Supergirl", hero: "Superman" },
    { name: "Oracle",    hero: "Batman"   },
    { name: "Krypto",    hero: "Superman" }
];

var batKicks = sidekicks.filter(function (el) {
    return (el.hero === "Batman");
});

// Outputs: [
//    { name: "Robin",  hero: "Batman"   },
//    { name: "Oracle", hero: "Batman"   }
// ]
console.log(batKicks);

var superKicks = sidekicks.filter(function (el) {
    return (el.hero === "Superman");
});

// Outputs: [
//    { name: "Supergirl", hero: "Superman" },
//    { name: "Krypto",    hero: "Superman" }
// ]
console.log(superKicks);
```

The filter method accepts a callback function. In that callback function we examine each element of the array indidually, and return true if we want the element to pass the filter.