6.170 Tutorial 7: Javascript and jQuery

Javascript

All example code located at git@github.com:6170/tutorial_examples.git inside the tutorial7 folder. Examples are separated into directories called example# which correspond to different sections of this tutorial. You should go through the notes and appropriate commented examples in parallel. Some code sections were copied into this tutorial writeup for convenience.

What is Javascript?
Throughout this course we’ve sought to obey the separation of concerns principle to keep our code understandable and modular. The model-view-controller design of Rails is one example of this. Another major example is the fact that HTML, CSS, and Javascript act in ways that complement, but do not overlap with one another: HTML is used to organize web content. CSS is used to control the presentation and styling of content. And Javascript - the final piece that we’ll be talking about today - is used to define the behavior and interactions of the web content.

Javascript is a lightweight, interpreted, object-oriented, and functional scripting language for the web. It’s the dominant scripting language on the web - in HTML5 it’s even assumed that a <script> tag is of type “text/javascript” so you don’t need to specify.

The gameplan is to first show the very basics of JS so you’re comfortable with basic syntax and where files go. Then we’ll jump into the advanced concepts that can be difficult to pick up on your own.

Here’s an example of a typical static website which uses external CSS and JS to ensure concerns are separated. Under normal circumstances, you should never use inline or embedded JS elements.

Walk through example1 code as you cover these topics.

Types, Variables, Literals

Types of Javascript Values:
- numbers - (there is no separate int and float type)
- booleans (true or false)
- strings -
- undefined - All uninitialized variables initially have the value ‘undefined’.
null - a special keyword for when to denote a null value. Similar to None in Python. JS couldn’t have only had ‘undefined’ because then you couldn’t tell the difference between undeclared variables and declared variables with special (NA, None, or nil)-like values.

Objects - objects are most similar to Python dictionaries at first glance. They have keys and corresponding values. We’ll discuss in more detail as we go through the tutorial.
  - Ex. var obj = {prop: 3, propB: “something”};

Functions - JS functions are first class objects which take arguments and define a certain behavior when they are invoked.
  - Ex. var funcy = function() {console.log(“works”);};
  - Ex. funcy(); // Execution

Variables
  - You can declare a variable using the ‘var’ keyword. Ex. var variable_name = value. This can create a global variable if the variable is declared in a global scope or a local variable if it is declared inside a local scope. Never declare a variable without the ‘var’ keyword.

Variables can also store objects and functions (Functions are first class objects).

Type Conversions
  - Number to string conversions will be performed automatically
  - For string to number conversions, use the parseInt() or parseFloat() functions

Control Flow
  - JS provides the standard for loop, while loop, do..while loop, if/elseif/else conditionals, etc. Tell students to investigate these on their own.

Copy from Example
console.log("This is the console. Your html page is running the learningjs.js script");
// Types, Variables, Literals
console.log("Types, Variables, Literals\n");
// Number literals
var my_num = 5.0;
var your_num = 99;
// String literal
var sample_string = "The sum of our numbers is ";
// Type Conversion
console.log(sample_string + (my_num + your_num));
// undefined
var some_var;
console.log(some_var);

// Literal Object
var course6005 = {name: "Elements of Software Construction", number: 6.005};
var course = {name: "Software Studio", number: 6.170};
console.log(course); // Use the Console Tool to expand.
// Function
var show_course = function(course) { // Look, I'm storing a function in a variable.
    console.log("You're in " + course.name);
}
// Function invocation
show_course(course); // What do I do? Should be simple.

// But you're used to Object having methods right? Let's make it happen.
var better_course = {
    name: "Software Studio",
    number: "6.170",
    get_prereqs: function() {
        // Traverse a graph of classes and compute all prereqs rather than hard coding
        // Populate an array to be returned (mock this by hardcoding it)
        result = [course6005,] //Look, you made an array too!
        return result
    },
    show: function() {
        console.log("You're still in " + this.name);
    }
}
console.log(better_course.get_prereqs());
better_course.show();

A Bit more about Objects:
- You saw the object called course we created right? We can call course.name and course.number to retrieve the values associated with each of those properties just like a Python dictionary. But don't object usually have methods too? Well in Javascript, the fact that a function is a first class object means we can store a function in an object, effectively making it a method. The way this happens so naturally is one great feature of JS. Check out example1 to see how.
- You'll notice that manually creating objects using the literal notation is getting tedious. We'll cover an improvement upon this when we discuss Prototypal Inheritance. For now, imagine how we could make things easier - and oh yeah, JS does not have the traditional notion of a class builtin so we need a fresh idea. :).

Let's jump into scoping...
Walk through example2 code as you cover these topics.

**Scoping**

When you declare a variable outside of any function, the variable is in the global scope and is available to any other code in the current document. Unlike in other languages where a local scope can be created inside a for loop or an if conditional block, in JS, declaring a variable inside a function makes it a local variable. Stated another way, Javascript does not have block scoping but rather function based scoping.

Why private variables?
1) Having variables that are available to all code running on a document can be dangerous. Other scripts (maybe not created by you - pluggins, 3rd parties) could read or write to your variables violating assumptions you have made or creating security or privacy vulnerabilities.
2) Global variables pollute the global namespace. Do your part to keep the global namespace green and pollution free. Discussed below.

```javascript
console.log("This is the console. Your html page is running the scoping.js script");

// Scoping

console.log("Scoping\n");

if (true) {
    var x = 5;                            // Global Variable
}
console.log(x);                           // Will print 5

for (var i=0; i < 10; i+=1) {
    var index = i                         // Also a global variable
}
console.log(i);
console.log(index);

// Why would we want to make a variable "private"?
// Create local variables using function scoping

var funcy = function() {
    var secret = "abcdef";                 // Local to the function funcy
}
//console.log(secret);                    // Attempt to read secret, but
secret is a local variable
```
// Namespace pollution
console.log(window);

So what we’ve learned is that in example1, all those variables we were creating were visible to
any other scripts running on the page. This is exactly what we mean when we say namespace
pollution. All global variables are attached to the one “global object” which is the window in a
web page (not to be confused with a global variable storing an object).

- At the end of scoping.js, there is a call to print the global window object. Look at the
  printout in your console and identify the global variables x, i, and index. Look at all that
  pollution. Notice, there is no window.secret though because its a local variable.

The solution is to place all of our Javascript code inside a function which is automatically
invoked. Without jQuery, you could do this like:

(function () {
    // All code
})();

With jQuery,

$(document).ready(function() {
    // All code here
});

does the same thing except it waits for the DOM to be constructed. Move on to example3 to see
an example of this in action.

Walk through example3 code as you cover these topics.

**Closure**

A closure is a function that can be passed around like an object but that also remembers the
value of all variables that were in scope when the function was created and is able to access
those variables when it is called (although those variables are no longer in scope). Nested
functions have access to the variables in their defining scope which is where the “variables that
were in scope when the function was created” part comes from.

(function() {
    console.log("This is the console. Your html page is running the
closure.js script");

    // Closure

    console.log("Closure\n");

    // Now that we know scoping, its not surprising that this works.
var init = function() {
    var name = "Make sure you understand JS scoping"; // Local variable inside function
    // inner function - only available inside init()
    var show_name = function() { // Create a function which has access to 'name'
        alert(name);
    }
    show_name();
} // show_name is invoked before the execution of init completes
init();

//This is a bit more surprising. This works because of closure
var init2 = function() {
    var name = "Closure!";
    var show_name = function() {
        alert(name);
    }
    return show_name;
}
var func_to_show_name = init2();
func_to_show_name();
/* So calling init2 returns a function which we then call. Closure enables the function stored in func_to_show_name to remember the variable name that was available when the function was defined. */

// A more Practical Example

function makeCounter(start_count, end_count) {
    var count = start_count;
    return {
        increment_count: function() {
            if (count < end_count) {
                count += 1;
            }
        },
        get_count: function() {
            return count;
        }
    }
}


```javascript
mycounter = makeCounter(1, 4);
console.log(mycounter.get_count());  // 1
mycounter.increment_count();         // 2
console.log(mycounter.get_count());
mycounter.increment_count();         // 3
mycounter.increment_count();         // 4
console.log(mycounter.get_count());
mycounter.increment_count();         // 4
console.log(mycounter.get_count());

// Check and you can see that we're not polluting the global namespace.
console.log(window);
```(})();     // Don't place anything below here.

See exercise3.5 for code sample

Object and Inheritance

You've seen that you can define objects using literal notion and how to add methods
to objects by giving the object a function as a property. Now we'll cover the getters and
setters you can use with object and a simple inheritance example that shows the use of
Object.create(parent).

Javascript allows the dot notation and the [] bracket notation when accessing and setting object
properties.

//Object Literal (keys assumed to be strings so quoting them is optional)
var sample = {url: "http://www.google.com", 'owner': 'google'};

//Property getters
console.log(sample.url);
console.log(sample['url']);

//Property Setters - setters return the value they just set as an object
property
sample.url = "http://www.mysite.com";
sample['owner'] = 'Daniel Jackson';
console.log(sample.prop = "value");       // Returns the set value "value" to
be console logged.
sample.prop = 3;

```
You can also delete properties from an existing object.

delete sample['prop']
delete sample.prop           //idempotent
console.log(JSON.stringify(sample));
console.log(sample);

**Prototypal Inheritance**

Javascript uses prototypal inheritance and is class-free. Prototypal inheritance is believed by some to be an innovative step ahead of classical inheritance (more concise, can simulate classical inheritance and do more).

Simply create an object that you like, create new instances of it using `Object.create(proto, [propertiesObj])`, and then customize the new objects. There is no reason to have “Classes” in the traditional sense, only objects inheriting from objects.

**Delegation**

Also known as differential inheritance. Basically, when a property not known to an object is accessed, the object delegates to its prototype object (which has its own prototype object, forming a prototype chain) to find the sought property. This is similar to how super-classing works in classical languages. The benefit of differential inheritance is that when you create an object which inherits from another, you only need to define the ways in which it differs from the object it inherited from - this significantly cuts down on the need for code reuse.

```javascript
var person = {name: "Bob",
              age: 21
          };

var zoo_keeper = Object.create(person);
//We only need to define the differences between the person and zoo_keeper objects.

zoo_keeper.name = "Frank";       //Zookeepers name will be Frank since it is earlier in the prototype chain.
zoo_keeper.enter_park = function() {
    console.log(this.name + " entered the park");
};

// When an object does not have a particular property, it checks its parents along its prototype
// chain to see if the property is defined.
console.log(zoo_keeper.name);
console.log(zoo_keeper.age); //21
console.log(zoo_keeper);
close console.log(person);

zoo_keeper.enter_park();

**Conclusion on Javascript**

---

**jQuery**
(Much of this is shamelessly stolen from the jQuery Novice to Ninja guide, which I found to be an excellent resource.)

**What is JQuery?**
It's a library designed to help speed up web development. Jquery lets you accomplish the same thing with way less code than it would take with javascript, by adding another layer of abstraction on top of javascript. It makes it easy to deal with things like:

- DOM lookup, traversal, and manipulation
- AJAX requests
- Cross-browser compatibility
- JQuery UI: Drop down menus, drag and drop elements, animations, form validation
- Lots of other stuff, via JQuery plugins

**Downsides:**
- Speed and size (which are generally not a big deal)

**How to use JQuery:**
Method 1 - You can download a single javascript file containing everything that comes out of the box forjquery, and include this file in your web page to give you access to all of jquery's predefined functions. Just add a script tag.

Method 2 - You can point your page to a remotely hosted (thanks Google) jQuery script.
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.8.0/jquery.min.js" type="text/javascript"></script>

**The jQuery Alias:**
Including jQuery in your page gives you access to a single magical function called "jQuery". Having a single function name to access the entire jQuery library prevents cluttering up your global namespace and having to worry about whether you can name your function "hide" or whether jQuery uses that function name. However, since typing "jQuery" a thousand times is annoying, it provides a shorter alias to access the library, "$". You almost always will want to use the $ instead of typing out jQuery, unless you have something like other libraries competing for the $ function name, in which case you can do something like:

(function($){
    //some code
})(jQuery);

to make jQuery available as $ without polluting the global namespace. Woohoo closures!

Parsing a jQuery statement:
Each command has four parts: the jQuery function (or its alias), selectors, actions, and parameters. If you see jQuery code that looks confusing, try to break it down into these four parts and figure out what each part is saying.

Let's start with a simple example:

$('<p>').css('color', 'blue')

Selectors -
A selector selects one of more elements on a web page. Here we've told the selector to select all the elements with paragraph tags (<p>).

Action -
Gets applied to all the elements that were selected. Here we're using jQuery's `css` action, which is used to modify a css property.

Parameters -
Arguments that get passed to the action. Here we're telling the `css` action to set the css color property to the value blue.

We'll come back to selectors, but first:

Making sure the page is ready:

Before we can interact with the HTML elements on a page, those elements need to have been loaded. The old school way to do this is to make sure the entire page (including images) had finished loading before running any scripts. But, jQuery has a built-in document ready event that will execute our code as soon as possible.

$(document).ready(function() {
alert('Document is ready.');
});

This says "When our document is ready, run our function." Your code should pretty much always go after this, and you should only need to declare it once per page.

Since this is so common, there is a shortcut way to say document ready:

$(function() { alert('Document is ready.'); });

This doesn't look much like the jQuery function + selector + action + parameter pattern we promised, but beneath all the syntactic shortcuts we still have:

- **jQuery function** = $
- **selector** = `document`
- **action** = `ready`
- **parameter** = `function() { alert('Document is ready.'); }`

### More about selectors:

$(<selectors go here>)

**Selecting by element type:** pass the element's HTML name as a string parameter to the $ function. For example, to select all table row elements (which have the `<tr>` tag), you would use $`('tr')`.

**Selecting by id:** use the # followed by the element's id, so something with id='data' would be $`('#data')`

**Selecting by class:** use . followed by the element's class name, so something with class='blue' would be $`('.blue')`

We can combine selectors in a variety of ways:
- combining selectors with a space between to narrow down the selection (selects the intersection)

$`('#data tbody tr')` would select the table rows inside the table body of the element with id data.

- using selector:filter to filter out elements you don’t want

$`('#data tbody tr:even')` keeps every even-indexed element in the original selection, so if you initially had 8 elements selected when you did $`('#data tbody tr')`, this would give you the ones indexed 0, 2, 4, and 6.
-combining selectors with commas between to select all of them (selects the union)
$('p,div,h1') selects all the paragraphs, divs, and h1 headings.

**Event Handlers**
Events are actions and user interactions that occur on the web page. When an event happens, we say that it has *fired*. When we write some code to handle the event, we say we *caught* the event. When the user moves the mouse, or clicks a button, or when a browser window is resized, or the scroll bar moved, each of these fires an event that we can catch, and act on. We already saw one event handler, for the document-ready event. When the document said “I’m ready”, it fired an event, which our jquery statement caught, and acted on.

Let’s look at another event handler, the click handler.

```javascript
$('#hideButton').click(function()
    { $('#uglything').hide();
});
```

When #hideButton gets clicked, our function gets executed and hides #uglything.

**this**
When an event fires, we will often want to refer to the element that fired it. For example, we might want to modify the button that the user has clicked in some way. We can refer to the javascript object that fired the event using the keyword *this*. To convert it from a javascript object to a jquery object, we wrap it in the jquery selector, giving us `$($(this))`. This is easier and cleaner than trying to re-select the element that fired the event.

```javascript
$('#hideButton').click(function() {
    $(this).hide(); // a curious disappearing button.
});
```

**Ajax**
Making Ajax calls is easy with jQuery! (So easy that I can’t remember the other way to do them...)

```javascript
$.ajax({
    type: 'GET',
    url: 'getDetails.php',
    data: { id: 142 },
    success: function(data) {
        // grabbed some data!
    }
});
```
Or the even shorter way (but be careful if you need to handle failures!):

$.get(url, data, callback, dataType);
$.post(url, data, callback, dataType);

Events and scope

(You can skip this if you run out of time, but I seem to remember from last year that my final project group had all sorts of issues with javascript/jquery/callbacks and scope, probably because none of us really understood this part.)

You can define a variable as global by declaring it outside of any loops or constructs. Global variables will be accessible from anywhere in your code. Likewise, a variable that you declare inside a construct (such as a function or an object) is said to be local to that construct.

This seems simple, but it can become messy when we start defining callback methods for our Ajax requests, because the callback will often be run in a different context than the one where it was defined. Suppose we have defined a namespace “widget” and we have an event inside the widget namespace. If you try to refer to this in a callback, expecting it to point to your widget namespace, you’ll be unpleasantly surprised: it might be undefined, or it might refer to something else entirely. For example:

```javascript
var WIDGET = {};
WIDGET.delay = 1000;
WIDGET.run = function() {
    alert(this.delay); // 1000 ... good!
    $(p).click(function() {
        alert(this.delay); // undefined! bad!
    });
};
```

When a p tag is clicked, our event handler runs in a different context than the widget object itself. So this.delay will most likely not exist (and if it does, it’s a different variable to what we wanted anyway!). There are a few ways we can deal with this, but without being too JavaScripty, the easiest way is to store the widget’s scope in a variable:

```javascript
var WIDGET = {};
WIDGET.delay = 1000;
WIDGET.run = function() {
    alert(this.delay); // 1000 ... good!
    var _widget = this;
    $(p).click(function() {
        alert(_widget.delay); // 1000 ... good!
    });
};
```
alert(_widget.delay) // 1000 ... yes!
});
};

By setting _widget to point to this *in the context of the widget*, we'll always be able to refer back to it, wherever we are in the code. In JavaScript, this is called a closure. The general convention is to name it _this (though some people also use self). If it’s used in a namespacing object, it’s best to name it with an underscore (_), followed by the widget name. This helps to clarify the scope we’re operating in.