Module 6
SaaS Clients, JavaScript

CS W169A: Software Engineering

1 Overview

Before you took this class, there's a good chance you've heard of the JavaScript programming language. In this course, we'll explore how we can enhance the client side user experience of our applications with this powerful language. The JavaScript community has blossomed in recent years, and frameworks such as React and Node.js make it possible to write full stack applications in JavaScript! We will mainly focus on the JavaScript language, AJAX, the DOM, and Events/Callbacks.

2 WWJD

Similar to Ruby, Javascript closely follows the "everything is an object" paradigm with a couple exceptions. The short list of primitives include String, Number, undefined (no value), null (different from undefined), Boolean, and BigInt. In addition, similar to Ruby, JavaScript values are dynamically typed, so variable declarations are preceded by `var` or `let`, instead of types.

That being said, Javascript has a lot of interesting idiosyncrasies worth considering. To code effectively in Javascript, it's worthwhile to familiarize yourself with these peculiarities. In this section, we'll be going over several of them with a more than familiar set of "What Would Javascript Do" questions.

2.1 Boolean

Because of how variables are dynamically and loosely typed, there are values that evaluate the True/False when cast to a Boolean, but when compared against each other, they may or may not evaluate to true.

There are seven values "Falsy" values in Javascript. "Falsy" means that they evaluate to false in conditionals: `0` (number), `0n` (BigInt), `null` (keyword), `undefined` (keyword), `false` (boolean), `NaN` (number), `""` (string). However, these values are not necessarily equal to one another.

"=="	0	0n	null	undefined	false	NaN	""
0	true	true	false	false	true	false	true
0n		true	false	false	true	false	true
null			true	true	false	false	false
undefined				true	false	false	false
false					true	false	true
NaN						false	false
""							true

To verify your understanding, complete these questions:

- `undefined == null` => true
- `NaN == NaN` => false
  A quick explanation from the ECMAScript documentation (link)
  1. If Type(x) is different from Type(y), return false.
  2. If Type(x) is Number, then
  i. If x is NaN, return false.
  ii. If y is NaN, return false
- `null == false` => true
- `0 == false` => false
- `"" == false` => false

Boolean comparisons are tricky! We'd recommend 1. Using built in compare methods instead of == or === 2. Avoid comparing different types 3. Explicitly convert types to Boolean when necessary.

2.2 Arrays

Arrays are one of the core data structures in Javascript, but they behave in interesting ways. In this exercise, we'll take a look at some of the relatively odd behaviors of traditional array operations. Don't worry if you don't get these right! This is meant to be more of an exploration. Try to justify why such behavior is the case, too!

- `[1, 2, 3] + [4, 5, 6]`
  Solution: "1,2,34,5,6" (Type: String)
  Array concatenation is a bit weird! The reason for this is that the + operator isn't defined for arrays. In fact, Javascript will convert arrays into strings, then concatenate those. To break down the fall, this is what's happening under the hood:

  ```
  [1, 2, 3] + [4, 5, 6]
  [1, 2, 3].toString() + [4, 5, 6].toString()
  "1,2,3" + "4,5,6"
  "1,2,34,5,6"
  ```

  To append to an array correctly, use the built in ‘push‘ method (docs).
- `!![]` => true
- `[] == true` => false
- `[10, 1, 3].sort()` => [1, 10, 3]
  Explanation: The default sort converts each element to a string, then does a lexicographical sort
- `[] == 0` => true
  Array Equality is a beast of its own. For more weirdness, check out this link

3 Closures

A popular paradigm in Javascript is the use of closures. Similar to higher order functions in Python, closures are combinations of a function bundled together with references to its surrounding state. Closures give you access to an outer function's scope from an inner function. Determine the output of the following code:

```
function foo(x) {
  var baz = 3;
  return function (y) {
    console.log(x + y + (baz++));
  }
}
var bar = foo(5);
bar(11);
```

Solution: 19

Explanation: When `foo(5)` is invoked, the value of `x` is set to 5. `baz` is set to 3. The nameless function taking in a parameter of `y` is returned. When is called, the value of `y` is set to 11. The input to `console.log` would therefore be `5 + 11 + (3++)`. The summation would take place first, giving 19, before the 3++ takes place.

4 Event Listeners

When designing web applications, you may often want "reactive" behavior, meaning elements on a web page (i.e. button, form, checkbox) respond to a specific user input (i.e. mouse click, key press). Event listeners help us describe such behavior programmatically. The jQuery library is one of the most popular tools for this purpose.

Write a click handler for the below `div` with id `foobar` in jQuery that changes the background color of the `div` element to `FF0000` (Red).

```
<div id="foobar">Change my background</div>
```

Solution: (This code snippet can be directly included in an HTML document)

```
<script>
$('#foobar').on("click", function() {
        $(this).css('background-color', '#FF0000');
});
</script>
```

5 AJAX Requests

AJAX (Asynchronous JavaScript and XML) is a group of tools and techniques for the development of asynchronous web applications. The goal of AJAX is that communication between an application and data server (i.e. HTTP requests) do not interfere with the experience, display, and behavior of the application.

Below, you're given a form that simulates logging in. We'd like to check whether the username/password combination works along with the account if there is one. To do so, we want an HTTP POST request to be made when this form is submitted. Write your solution with jQuery, and comment on where the callback function should be located.

```
<form method="POST" id="foo">
    <input type="text" class="user" />
    <input type="password" class="pass" />
    <input type="button" value="Log␣in" id="onSubmit" />
</form>
```

Solution: There's more than one way to do this! Our sample is below. The specific details of how inputs are communicated in the request depends on the route you're making the request to. In this case, we've assumed that the data should be as a dictionary.

```
$("#onSubmit").click(function() {
    inputs = // Grab inputs of form here
    $.ajax ({
        url: "urltoyourlogin",
       data: inputs,
       success: function() {
        $("#login").html("You␣are␣now␣logged␣in!");
       }
    });
})
```

6 Classes

Let's practice Object Oriented Programming and Inheritance in Javascript. Design 2 classes, one called "Pokemon" and another called "Charizard". The classes should do the following:

Pokemon Class:

- Constructor takes 3 parameters (HP, attack, defense)
- Constructor should create 6 fields (HP, attack, defense, move, level, type). The values of (move, level, type) should be initialized to ("", 1, "").
- Implement a `fight` method that throws an error indicating no move is specified.
- Implement a `canFly` method that checks if a type is specified. If not, throw an error. If it is, check whether the type includes "flying". If yes, return true, if not, return false.

Charizard Class:

- Constructor takes 4 parameters (HP, attack, defense, move)
- Constructor sets move and type (to "fire/flying") in addition to setting HP, attack, and defense like the superclass constructor.
- Override the `fight` method. If a move is specified, print a statement indicating the move is being used, and

return the attack field. If not, throw an error.

Solution:

```
class Pokemon {
    constructor(HP, attack, defense) {
        this.HP = HP;
        this.attack = this.attack;
        this.defense = this.defense;
        this.move = ""
        this.level = 1;
        this.type = ""
    }

    fight() {
        throw "No␣move␣specified";
    }

    canFly() {
        if (this.type) {
            if (this.type.includes("flying")) {
                return true;
            } else {
                return false;
            }
        } else {
            throw "No␣type␣specified";
        }
    }
}

class Charizard extends Pokemon {
    constructor(HP, attack, defense, move) {
        super(HP, attack, defense)
        this.move = move;
        this.type = "fire/flying";
    }

    fight() {
        if (move) {
            puts "Charizard␣used␣the␣move␣" + this.move + "!";
            return this.attack;
        } else {
            super.fight();
        }
    }
}
```