LC 101

Unit 3 - JavaScript Quirks

April 6, 2017

Scoping

- What will this code do?

```
// Oops, these buttons will do something unexpected.
for (var i = 0; i < 3; ++i) {
    var button = $('<button></button>').text(i).click(function() {
        console.log(i);
    });
    var listItem = $('<li></li>').append(button);
    $('#bad-buttons ul').append(listItem);
}
```

- It creates buttons labeled ‘0’, ‘1’, and ‘2’, but they all print 3 when clicked!

Scoping

- A loop does not create a new scope!
  - The `text(i)` is called as the loop runs, so the buttons are labeled ‘0’, ‘1’, and ‘2’
  - But the event handler anonymous function is not called until the button is clicked
    - At that time, the value of `i` in the function’s closure is its value after the loop has completed (3 in this case)
    - All the event handlers functions have the same closure and are referring to the same `i`

```
for (var i = 0; i < 3; ++i) {
    var button = $('<button></button>').text(i).click(function() {
        console.log(i);
    });
...
```

Scoping

- How can we fix this?
  - We need to create a different closure for each event handler function by creating a new scope around its creation

Scoping

- When **`forEach`** calls a function for each value, that function call creates a new scope

```
// But this works like we expect.
var indexes = [0, 1, 2];
indexes.forEach(function(i) {
    var button = $('<button></button>').text(i).click(function() {
        console.log(i);
    });
    var listItem = $('<li></li>').append(button);
    $('#good-buttons-1 ul').append(listItem);
});
```

Scoping

- Or we can create a new scope by creating *and calling* a function in the loop

```
// We can "fix" the first version by creating a new scope for each button.
for (var i = 0; i < 3; ++i) {
   var createAndAddButton = function(j) {
           var button = $('<button></button>').text(j).click(function() {
           console.log(j);
       });
       var listItem = $('<li></li>').append(button);
       $('#good-buttons-2 ul').append(listItem);
   };
   createAndAddButton(i);
}
```

Scoping

- It doesn’t have to be a new function, just a new function call

```
// And this also creates a new scope for each button.
function createButton(i) {
    return $('<button></button>').text(i).click(function() {
        console.log(i);
    });
}

for (var i = 0; i < 3; ++i) {
    var listItem = $('<li></li>').append(createButton(i));
    $('#good-buttons-3 ul').append(listItem);
}
```

Scoping

- But this is most often fixed by creating and calling an *anonymous wrapper*

```
// We can also do this by creating and calling an anonymous wrapper.
for (var i = 0; i < 3; ++i) {
    (function(j) {
            var button = $('<button></button>').text(j).click(function() {
            console.log(j);
        });
        var listItem = $('<li></li>').append(button);
        $('#good-buttons-3 ul').append(listItem);
    })(i);
}
```

Equals

- The normal == operator will *cast* values to the same type
  - `"2" == 2` is true
- The === operator will not cast
  - `"2" === 2` is false

```
console.log("2" == 2);   // true
console.log("2" === 2);  // false
```

Truthy and Falsy

- In JavaScript, the following values are considered *falsy*
  - `false`, `0`, `""`, `null`, `undefined`, `NaN` (not a number)
- Everything else is considered *truthy*

- This means we can't just use `if (something) {...}` to check if something is null or undefined
  - Will also catch if something is zero, empty string, NaN
  - Need to instead do something like

```
if (something !== undefined && something !== null) {...}
```

Prototypes

- JavaScript uses prototypal inheritance (the only language that does?)
  - Every object has another object as its prototype
    - Usually the `Object.prototype` object
    - Actually, you *can* create objects with a null prototype, but it is not typically done

```
var fruits = { apple: 1 };
var fruitsAndVegetables = Object.create(fruits);
fruitsAndVegetables.carrot = 2;
for (var key in fruitsAndVegetables) {
   console.log(key);
}
// outputs 'carrot' and 'apple'
```

in vs hasOwnProperty

- The `in` operator will return true if the property exists in the object or anywhere in its prototype chain
- The `hasOwnProperty` function will return true only if the property exists in the object itself

```
console.log('apple' in fruitsAndVegetables);                // true
console.log(fruitsAndVegetables.hasOwnProperty('apple'));  // false
```

in vs hasOwnProperty

- This sometimes causes unexpected properties in a **`for...in`** loop
  - Will often see code that uses **`hasOwnProperty`** to filter properties in loops

```
for (var key in fruitsAndVegetables) {
   if (fruitsAndVegetables.hasOwnProperty(key)) {
       console.log(key);
   }
}
// only outputs 'carrot', not 'apple'
```

this and that

- In object oriented languages such as Java and C#, the **`this`** keyword refers to the object instance
  - Similar to **`self`** in Python (but do not need to include in method declarations)
- In JavaScript, this does **not** refer to the object instance, it refers to the *calling scope*

this and that

```
function Foo() {
    this.whatIsThis = function() {
        console.log(this);
    }
}

var foo = new Foo();
foo.whatIsThis();  // this will be the foo object in whatIsThis

var oops = foo.whatIsThis;
oops();  // this will be the global object (in non-strict mode)
        // or undefined (in strict mode)
```

this and that

- This is often “fixed” by assigning another variable to it in the constructor

```
function Bar() {
   var self = this;

   this.whatIsThis = function() {
       console.log(self);  // self will always be bar
   }
}

var bar = new Bar();
var ok = bar.whatIsThis;
ok();
```

More

- For more JavaScript quirks, check out http://bonsaiden.github.io/JavaScript-Garden/