Advanced JavaScript

by Vlad Ungureanu

Agenda

Advanced JavaScript

- Scope
- Bind
- Call
- Closures
- Coding Standards
- Best Practices

LEARN stuff IO

Scope

- In JavaScript, scope refers to the current context of your code.

- Scopes can be globally or locally defined.

```
// Scope A - global
var myFunction = function () {
        // Scope B – myFunction scope
        var name = 'Vlad'; // defined in Scope B
        var myOtherFunction = function () {
                // Scope C: `name` is accessible here since Scope C is wrapped in Scope B
         };
};
```

Scope - This

- Each scope binds a different value for ‘this’ of this depending on how the function is invoked.
- We’ve all used the this keyword, but not all of us understand it and how it differs when invoked.
- By default this refers to the outer most global object, the window.

```
var myFunction = function () {
        console.log(this); // this = global, [object Window]
};
myFunction();
      var myObject = {};
      myObject.myMethod = function () {
       console.log(this); // this = Object { myObject }
};
var nav = document.querySelector('.nav'); // <nav class="nav">
var toggleNav = function () {
        console.log(this); // this = <nav> element
}; nav.addEventListener('click', toggleNav, false);
```

Bind

- Bind creates a new function that, when called, has its this keyword set to the provided value.

- We pass our desired context, this (which is myObj), into the .bind() function.

- When the callback function is executed, this references myObj.

```
Function.prototype.bind = function (scope) {
    var fn = this;
    return function () {
        return fn.apply(scope);
    };
}
```

Bind

```
var myObj = {
   specialFunction: function () {
   },
   anotherSpecialFunction: function () {
   },
   render: function () {
         this.getAsyncData(function () {
          this.specialFunction();
          this.anotherSpecialFunction();
   }.bind(this));
}};
```

Call

- The .call() and .apply() methods allow you to pass in a scope to a function, which binds the correct this value.

```
var links = document.querySelectorAll('nav li');
      for (var i = 0; i < links.length; i++) {
        (function () {
          console.log(this);
        }).call(links[i]);
}
```

Closures

- A closure is an inner function that has access to the outer (enclosing) function's variables—scope chain.
- The closure has three scope chains: it has access to its own scope (variables defined between its curly brackets), it has access to the outer function's variables, and it has access to the global variables.
- The inner function has access not only to the outer function's variables, but also to the outer function's parameters.
- Note that the inner function cannot call the outer function's *arguments* object, however, even though it can call the outer function's parameters directly.
- The most simple way to think of a closure is a function that can be stored as a variable (referred to as a "first-class function"), that has a special ability to access other variables local to the scope it was created in.

Closures

```
function showName (firstName, lastName) {
var nameIntro = "Your name is ";
   // this inner function has access to the outer function's variables, including the parameter
        function makeFullName () {
                return nameIntro + firstName + " " + lastName;
        }

        return makeFullName ();
}

showName ("Michael", "Jackson");
```

Closures

```
$(function() {

    var selections = [];
    $(".niners").click(function() { // this closure has access to the selections variable
      selections.push (this.prop("name")); // update the selections variable in the outer
    function's scope
    });

});
```

Coding Standards

- Use white spaces and general spacing in accordance to Java coding standards for readability
- Either declare ‘var’ specifically for all variables or only for the first one
- Declare functions as variables with the same name for traceability and self calling
- ALWAYS evaluate for the best, most accurate result - the above is a guideline, not a dogma
- Use Clean Code conventions for naming variables, objects and methods

LEARN stuff IO

Best Practices

- Avoid Global Variables
  - Minimize the use of global variables.
  - This includes all data types, objects, and functions.
  - Global variables and functions can be overwritten by other scripts.
  - Use local variables instead, and learn how to use closures.
- Declarations on Top
- Initialize Variables
- Never Declare Number, String, or Boolean Objects
  - Always treat numbers, strings, or boolean as primitive values. Not as objects.
  - Declaring these types as objects, slows down execution speed, and produces nasty side effects:
- Don't Use new Object()
  - Use {} instead of new Object()
  - Use "" instead of new String()
  - Use 0 instead of new Number()
  - Use false instead of new Boolean()
  - Use [] instead of new Array()
  - Use /()/ instead of new RegExp()
  - Use function (){} instead of new function()

LEARN stuff IO

Best Practices

- Beware of Automatic Type Conversions
  - Beware that numbers can accidentally be converted to strings or NaN (Not a Number).
  - JavaScript is loosely typed. A variable can contain different data types, and a variable can change its data type

- Use === Comparison
  - The == comparison operator always converts (to matching types) before comparison.
  - The === operator forces comparison of values and type

- Avoid Using eval()

- Understand Closures

- Pick a framework and learn it

THANK YOU!

Vlad Costel Ungureanu
ungureanu_vlad_costel@yahoo.com