— Front-end training —

Scope and Context

What is Scope

Scope is a set of variables one have access to.

Scope is the set of rules that determines where and how a variable (identifier) can be looked-up.

JavaScript has functional scope.

Local and Global Scopes

Variables declared outside of any function are called **global**, because they are visible throughout the program

Each function has its own scope, and any variable declared within that function is only accessible from that function (**local**)

Declaring variables without `var` makes them global automatically.

Scope Chain

Functions can be created inside other functions, producing several degrees of locality.

```
function first(){
  second();
  function second(){
    third();
    function third(){
      fourth();
      function fourth(){
        // do something
      }
    }
  }
}
first();
fourth();
```

Closures

Closures are functions that refers to variables from outer scopes.

```
var sayHello = function (name) {
   var text = 'Hello, ' + name;
   return function () {
     console.log(text);
   };
};
```

Module Pattern

One of the most popular types of closures is what is widely known as the module pattern

```
var Module = (function(){
  var privateProperty = 'foo';

  function privateMethod(args){
    // do something
  }

  return {

    publicProperty: '',

    publicMethod: function(args){
      // do something
    },

    privilegedMethod: function(args){
      return privateMethod(args);
    }
  };
})();
```

Let me to the Scope

- Language-defined: `this` and `arguments`.
- Formal parameters: are scoped to the body of the function.
- Function declarations: `function foo() {}`.
- Variable declarations: `var foo;`.

Quiz Time...

```
console.log(foo);

var foo = 7;

function foo() {};
```

Hoisting

ECMA: Variables are created when the execution scope is entered.

Regardless of where a variable is **declared**, it will be, *hoisted* to the top of current scope.

```
var foo; // declaration
foo = 'bar'; // initialization
```

Hoisting (cont.)

```
foo();
function foo() {
  console.log(bar);
  if (false) {
   var bar = 1;
  }
  console.log(baz);
  return;
  var baz = 1;
}
```

Hoisting (cont.)

It doesn't matter whether the line with the variable declaration would ever be executed.

```
foo();
function foo() {
  var bar, baz;
  if (false) {
    bar = 1;
  }
  return;
  baz = 1;
}
```

Hoisting (cont.)

```
function foo() {
  bar(); // Uncaught TypeError: bar is not a function(…)
  if (true) {
    function bar() { return 'baz'; }
  }
}
```

Always declare stuff at the top and never in a loop or conditional.

Context

One of the most confused mechanisms in JavaScript.

Context is most often determined by how a function is invoked.

Context is always the value of the `this` keyword.

Wat is Zis

```
function showThis() {
  console.log(this);
}
```

- `window` if you simply call function: `showThis(); // -> window`
- Object, if you call function as method:

  ```
  var obj = { fn: showThis };
  obj.fn(); // -> Object { fn: function }
  ```

- new object, if function used as constructor: `new showThis(); // -> showThis {}`
- DOM element, if you call function as event listener: `document.body.onclick = showThis; // ... -> <body...`

Call me baby

```
showThis.call(context/*, comma-separated list of arguments */);
showThis.apply(context/*, list of arguments as array */);

var petro = {name: 'Petro'};

showThis.call(petro); // Object {name: "Petro"}
```

Method Kidnapping

```
var gangsta = {
   money: 20,
   takeMoney: function(victim) {
      this.money += victim.giveMoney();
   }
}

var john = {
   money: 75,
   giveMoney: function() {
      var amount = this.money;
      this.money = 0;
      return amount;
   }
}

gangsta.takeMoney(john);

gangsta.takeMoney.call(john, bruce);
```

Method Borrowing

```
sum(1, 9); // 10
sum(1, 2, 3); // 6

function sum() {
  return [].reduce.call(arguments, function(comp, curr) {
    return comp + curr;
  }, 0);
}

function isWordPalindrome(word) {
  return word === [].slice.call(word).reverse().join('');
}
```

Callbacks have this problem

```
...
getData: function() {
  $.getJSON('/api/assets', this.onGetData);
},
onGetData: function(data) {
  this.stopLoader();
 . ..
}
...
// somewhere in $ library
xhr.onreadystatechange = function() {
  // some checks, blah, blah
  ourCallback(xhr.responseText);
}
```

```
getData: function() {
  var self = this;

  $.getJSON('/api/assets', function(data) {
    self.onGetData(data);
  });
}
```

Closures

```
function bind(fn, ctx) {
   return function() {
     return fn.apply(ctx, arguments);
   };
}

var getValue = function() { return this.value }
var charlie = { value: 'charlie' };
var getCharlieValue = bind(getValue, charlie);

getCharlieValue(); // -> 'charlie'
getCharlieValue.call({ value: 'miranda' }); // -> 'charlie'
getCharlieValue.call(null); // -> 'charlie'
```

Bind that this

`Function.prototype.bind`

Bind does not call function immediately, it returns a new function with "bound" context.

```
// ...
var getCharlieValue = getValue.bind(charlie);
```

Bind that callback

```
...
getData: function() {
  $.getJSON('/api/assets', this.onGetData.bind(this));
},
...
```

Thank You!

Questions?

References

- http://www.adequatelygood.com/JavaScript-Scoping-and-Hoisting.html
- http://ryanmorr.com/understanding-scope-and-context-in-javascript/
- http://javascriptplayground.com/blog/2012/04/javascript-variable-scope-this/
- http://adripofjavascript.com/blog/drips/using-javascripts-array-methods-on-strings.html