C152 – Programming Language Paradigms
Prof. Tom Austin, Fall 2014

JavaScript Scope

Lab Solution Review
(in-class)

Remember that JavaScript has first-class functions.

```
function makeAdder(x) {
  return function (y) {
    return x + y;
} }
var addOne = makeAdder(1);
console.log(addOne(10));
```

Warm up exercise: Create a `makeListOfAdders` function.

input: a list of numbers

returns: a list of adders

```
function makeListOfAdders(lst) {
  var arr = [];
  for (var i=0; i<lst.length; i++) {
    var n = lst[i];
    arr[i] = function(x) { return x + n; }
  }
  return arr;
}

var adders =
     makeListOfAdders([1,3,99,21]);
adders.forEach(function(adder) {
  console.log(adder(100));
});
```

Prints:

121

121

121

121

```
function makeListOfAdders(lst) {
  var arr = [];
  for (var i=0; i<lst.length; i++) {
    arr[i]=function(x) {return x + lst[i];}
  }
  return arr;
}

var adders =
     makeListOfAdders([1,3,99,21]);
adders.forEach(function(adder) {
  console.log(adder(100));
});
```

Prints:

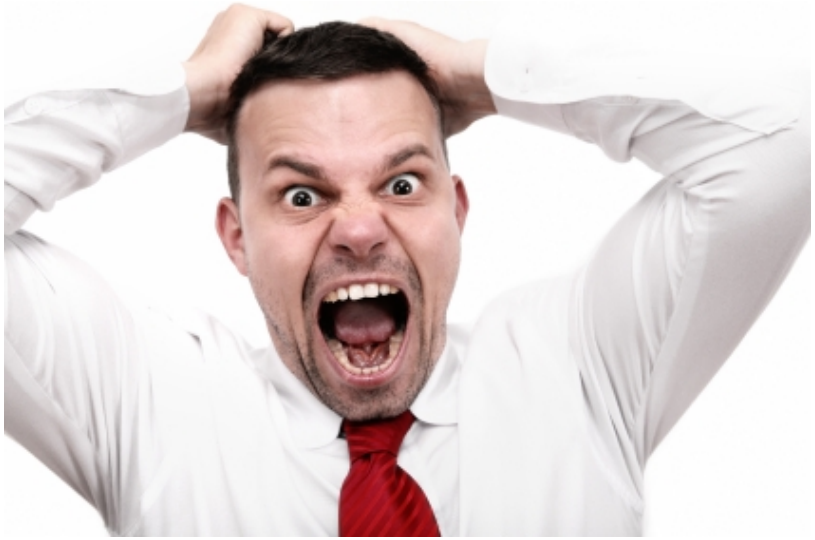
NaN

NaN

NaN

NaN

What is going on in this wacky language???!!!



JavaScript does *not* have block scope.

So while you see:

```
for (var i=0; i<lst.length; i++)
  var n = lst[i];
```

the interpreter sees:

```
var i, n;
for (i=0; i<lst.length; i++)
  n = lst[i];
```

In JavaScript, this is known as *variable hoisting*.

Faking block scope

```
function makeListOfAdders(lst) {
  var i, arr = [];
  for (i=0; i<lst.length; i++) {
   (function() {
    var n = lst[i];
    arr[i] = function(x) {return x + n;}
   })();
  }
  return arr;
}
```

Function creates new scope

A JavaScript constructor

```
name = "Monty";
function Rabbit(name) {
  this.name = name;
}
var r = Rabbit("Python");

console.log(r.name);
                        // ERROR!!!
console.log(name);
                        // Prints "Python"
```

Forgot `new`

A JavaScript constructor

```
function Rabbit(name, favFoods) {
  this.name = name;
  this.myFoods = [];
  favFoods.forEach(function(food) {
    this.myFoods.push(food);
  });
}
var bugs = new Rabbit("Bugs",
          ["carrots", "lettuce", "souls"]);
console.log(bugs.name); // "Bugs"
console.log(bugs.myFoods);
                          // Nothing prints
```

`this` refers to the scope where the function is called

Execution Contexts

Scoping is determined by *execution contexts,* each of which is comprised of:

- A variable object
  - Container of data for the execution context
  - Container for variables, function declarations, etc.
- A scope chain
  - The variable object plus the parent scopes
- A context object (`this`)

Global context

- Top level context.
- Variable object is known as the *global object*.
- `this` refers to global object

Function contexts

- Variable objects are known as *activation objects*. An activation object includes:
  - Arguments passed to the function
  - A special `arguments` object
  - Local variables
- What is `this`? It's complicated…

What does this refer to?

- Normal function calls: the global object
- Object methods: the object
- Constructers (functions called with `new`): the new object being created.
- Special cases:
  - Can be changed with `call` and `apply` functions
  - Can be changed with `bind` method (Ecmascript 5)
  - For an in-line event handler (on the web), refers to the relevant DOM element

apply, call, and bind

```
x = 3;

function foo(y) {
  console.log(this.x + y);
}
foo(100);

foo.apply(null, [100]);   // Array passed for args
foo.apply({x:4}, [100]);
foo.call({x:4}, 100);     // No array needed

var bf = foo.bind({x:5}); // Create a new function
bf(100);
```

Lab: Intro to JavaScript

Continued from last time: In today's lab, you will explore both the functional and object-oriented aspects of JavaScript.

See Canvas for details.