

CS 252:

*Advanced Programming Language Principles*



# Scoping in JavaScript

Prof. Tom Austin

San José State University

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

```
a = makeListOfAdders([1, 5]);
```

```
a[0](42); // 43
```

```
a[1](42); // 47
```

```
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);  
  });  
}
```

this refers to the  
scope where the  
function is called

```
var bugs = new Rabbit("Bugs",  
  ["carrots", "lettuce", "souls"]);  
console.log(bugs.name); // "Bugs"  
console.log(bugs.myFoods);  
                        // Nothing prints
```

# Execution Contexts

Comprised of:

- A variable object
  - Container for variables & functions
- A scope chain
  - The variable object plus 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 (aka *activation objects*) include
  - 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
- Constructors (functions called w/ `new`):
  - the new object being created.
- Special cases:
  - `call`, `apply`, `bind`
  - in-line event handlers on DOM elements

# 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.