Closures
all the way through

Part of the Back2Front sessions
27 March 2015

… or how closures = functions in JS

Agenda

- Introduce closures/functions
- Mechanics of closures in JS engines
- Objects
- Sharing functionality via prototypes
- “Classes”
- Suggestions for project code
- Fun exercises
- Appendix: A known problem with today’s JS engines

First class functions

Everything in Javascript is a “first class” value.

i.e. Everything can be -

a) passed to functions as arguments,
b) assigned to variables and
c) stored in objects and arrays.

That includes numbers, strings, objects, arrays … and functions, “classes”, “methods”, … just everything.

… except operators.

```
// Let’s explore that idea.

function dist(x, y) {
    return x * x + y * y;
}

// Try dist(3,4). You should get 25.
// Now rewrite it as -

function dist(x, y) {
    debugger;
    return x * x + y * y;
}

// Run dist(3,4) again. Examine the debugger window
// and note down what you see.
```

"this" context of the call

InjectedScript.evaluate VM73:626

Scope Variables

Local

this: Window

x: 3

y: 4

Global Window

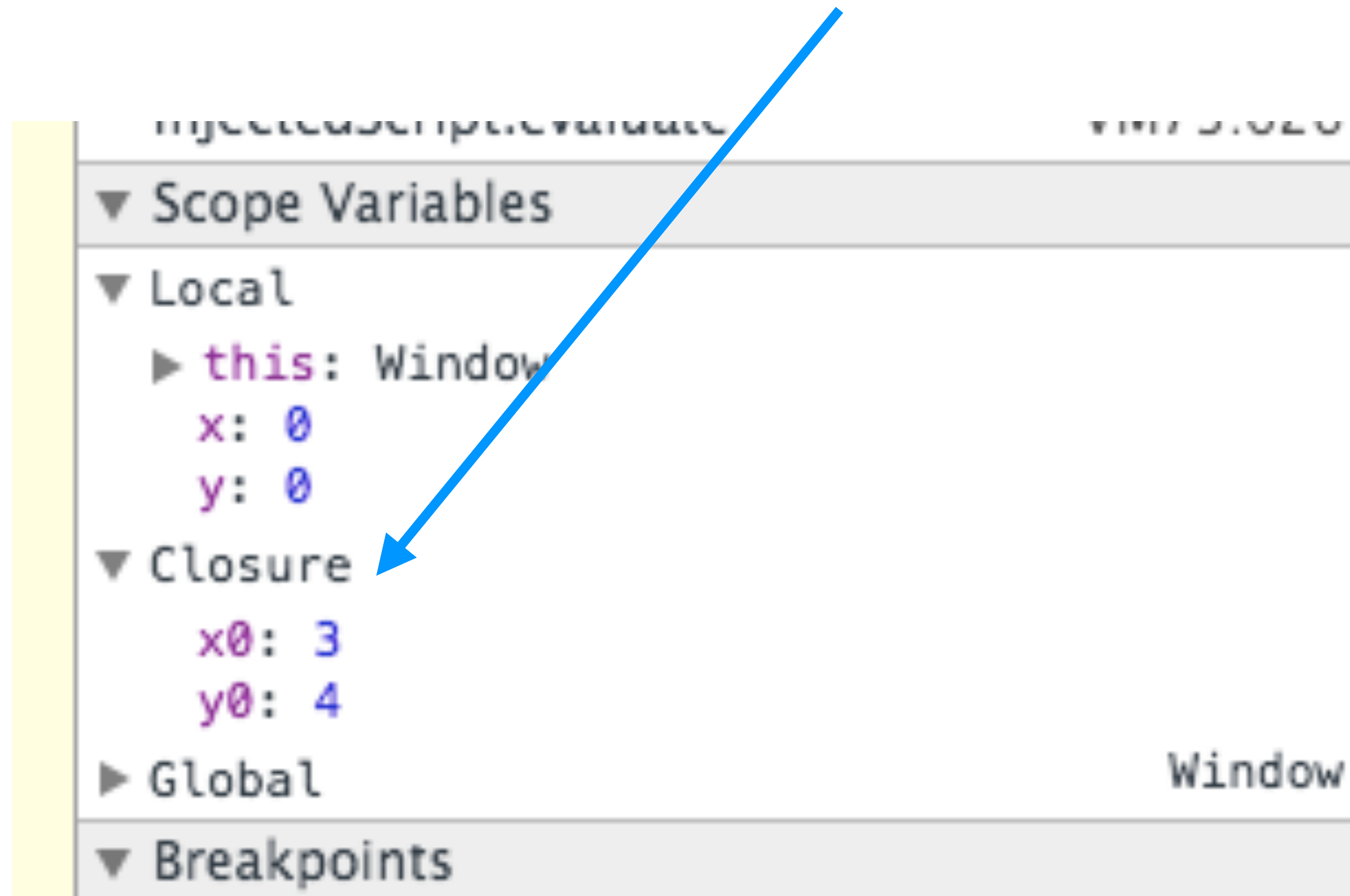
Breakpoints

Allocated local variables

```
// Now let’s make a relative distance calculator.

function relativeDist(x0, y0) {
    return function (x, y) {
        return dist(x - x0, y - y0);
    };
}

var g = relativeDist(3,4);

// Try g(0, 0). You should get 25.

// Explain how g is calculated and
// how g(0, 0) is evaluated.
```

```
// Now rewrite it as -

function relativeDist(x0, y0) {
    return function (x, y) {
        debugger;
        return dist(x - x0, y - y0);
    };
}

var g = relativeDist(3,4);

// Run g(0, 0) again. Examine the debugger window.
// What do you see? What is different from when
// we debugged the "dist" function?
```

The parent lexical context of the call



The mechanics of closures

```
// What happens when you call dist2(3,4)?

function dist2(x, y) {
    debugger;
    var x2 = x * x;
    var y2 = y * y;
    debugger;
    return x2 + y2;
}

// The runtime first allocates memory for
// x, y, x2 and y2.

// i.e. Each declared argument and declared
// “var” costs an allocation.
```

x2 & y2 are uninitialized but allocated already

Scope Variables
Local
this: Window
x: 3
x2: undefined
y: 4
y2: undefined
Global Window
Breakpoints

Allocations

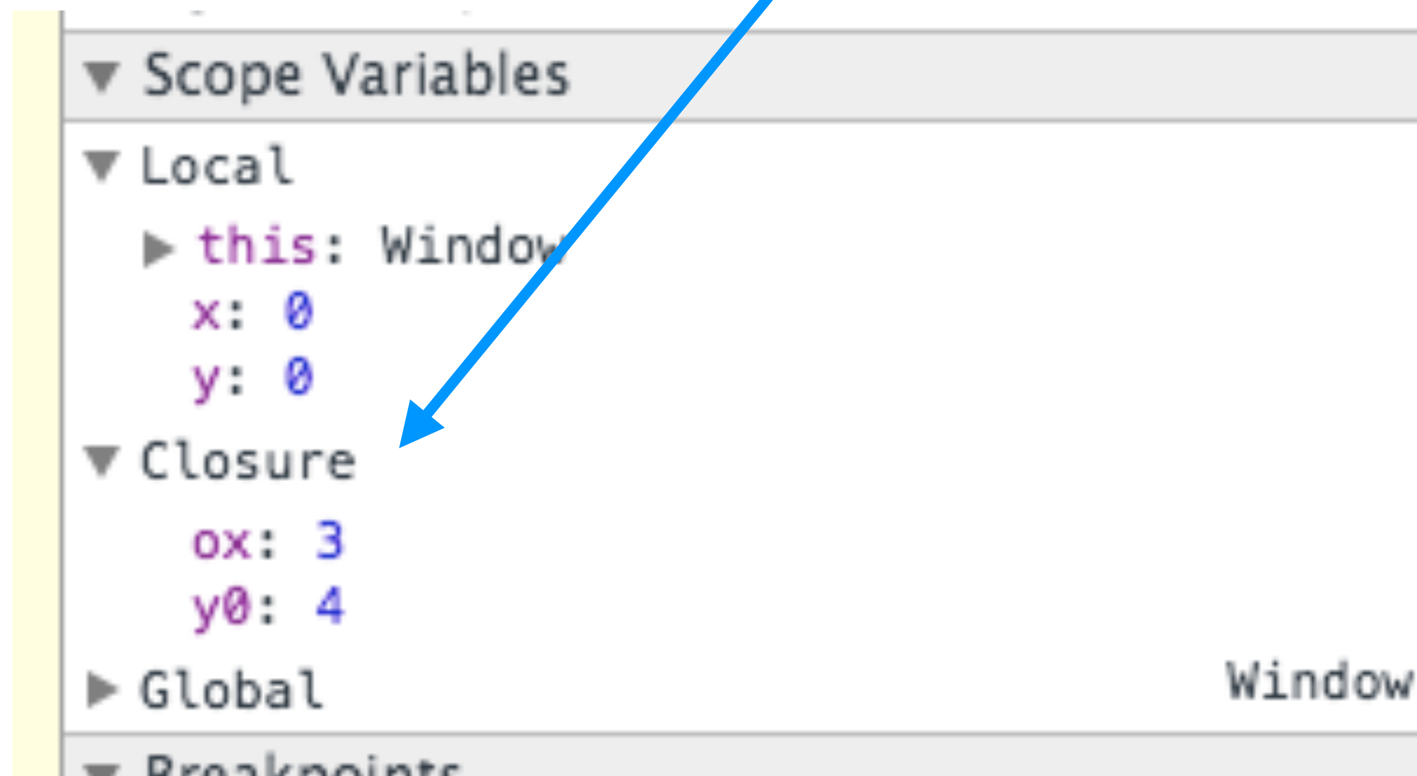
Scope Variables
Local
this: Window
x: 3
x2: 9
y: 4
y2: 16
Global Window
Breakpoints

x2 & y2 are initialized

```
// So var declarations are as though they’re
// at the beginning of the scope. And var
// initializations occur at the place where
// they’re declared.

function dist2(x, y) {
    var x2, y2;
    x2 = x * x;
    y2 = y * y;
    return x2 + y2;
}
```

```
// An inner function captures its outer function’s
// allocations.

function relativeDist(ox, oy) {
    var y0 = oy;
    return function (x, y) {
        debugger;
        return dist(x - ox, y - y0);
    };
}

var g = relativeDist(3,4);
```

The parent lexical context of the call



```
// Problem: Analyze what happens in this case.

function counter(n) {
    var funcs = [];
    for (var i = 0; i < 3; ++i) {
        funcs[i] = function () {
            console.log("index is", i);
        };
    }
    funcs[0]();
    funcs[1]();
    funcs[2]();
}

// Step 1: What allocations are for counter?
// Step 2: What is in the "Closure" of the
//         inner functions? (Use "debugger;")
```

```
// Problem: Analyze what happens in this case.

function counter(n) {
    var funcs = [];
    for (var i = 0; i < 3; ++i) {
        funcs[i] = function () {
            console.log(“index is”, i);
        };
        funcs[i]();
    }
}

// Step 1: What allocations are for counter?
// Step 2: What is in the “Closure” of the
//         inner functions? (Use “debugger;”)
```

Objects

```
function relativeToOrigin(x0, y0) {
    return {
        dist: function (x, y) {
            debugger;
            return dist(x - x0, y - y0);
        },
        vector: function (x, y) {
            debugger;
            return {x: x - x0, y: y - y0};
        }
    };
}

var g = relativeToOrigin(3,4);

// Run g.dist(0, 0) and g.vector(0, 0)
// Run g[“dist”](0,0) and g[“vector”](0,0)
// Look at “this” and “Closures”.
```

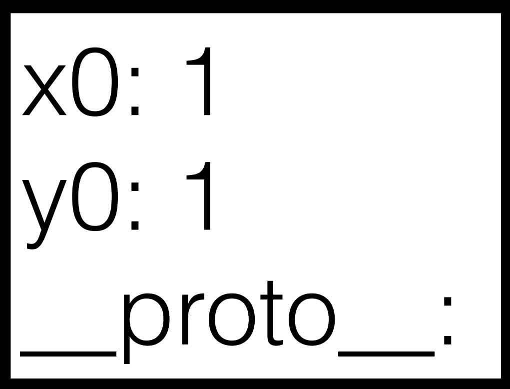
```
function relativeToOrigin(ox, oy) {
    return {
        x0: ox,
        y0: oy,
        dist: function (x, y) {
            debugger;
            return dist(x - this.x0, y - this.y0);
        },
        vector: function (x, y) {
            debugger;
            return {x: x - this.x0, y: y - this.y0};
        }
    };
}

var g = relativeToOrigin(3,4);

// Run g.dist(0, 0) and g.vector(0, 0)
// Look at “this” and “Closures”.
```

```
// Remember - “methods” are just functions
// and hence just normal values.

var distFunc = g.dist;
distFunc.call(g, 0, 0);

// What happens if you call distFunc(0, 0)?

distFunc.call({x0:3, y0:4}, 0, 0);

x0 = 3;
y0 = 4;
distFunc(0, 0);

// i.e. any “method” can be invoked on
// any object.
```

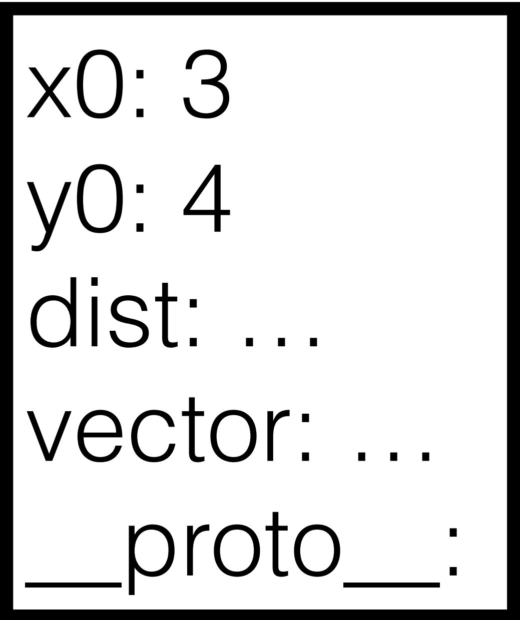
Sharing functionality via Prototypes

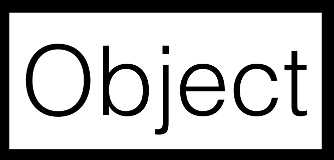
```
// Introducing Object.create().

function relativeToOrigin(ox, oy) {
    return {
        x0: ox,
        y0: oy,
        dist: function (x, y) {
           return dist(x - this.x0, y - this.y0);
        },
        vector: function (x, y) {
           return {x: x - this.x0, y: y - this.y0};
        }
    };
}

var g = relativeToOrigin(3,4);
var h = Object.create(g, {x0: {value: 1}, y0: {value: 1}});
h.dist(0, 0);

// Examine h. Checkout its "__proto__" property.

h.dist.call(h, 0, 0);
h.dist.call(g, 0, 0);
```

h



g





There is a short hand for all that

```
function ReferenceFrame(ox, oy) {
    this.x0 = ox;
    this.y0 = oy;
}

ReferenceFrame.prototype = {
    dist: function (x, y) {
        return dist(x - this.x0, y - this.y0);
    },
    vector: function (x, y) {
        return {x: x - this.x0, y: y - this.y0};
    },
    secret: “Decepticons hidden here!”
};

var g = new ReferenceFrame(3,4);

// Examine g. Checkout its “__proto__” and “constructor” properties.

console.log(g.secret);
ReferenceFrame.prototype.secret = “Decepticons found!”;
console.log(g.secret);
```

```
// Remember - Everything is first class … even
// these “classes”.

var R2O = ReferenceFrame;
var g1 = new R2O(3,4);
var obj = {r2o: R2O};
var g2 = new obj.r2o(3,4);
var g3 = new (function cat() {this.msg = “Meow!”;})();
var g4 = new g2.constructor(3,4);


// Examine g1, g2, g3. Checkout their “__proto__”
// and “constructor” properties.

console.assert(g1.constructor === ReferenceFrame);
console.assert(g2.constructor === ReferenceFrame);
```

Some suggestions for your project code

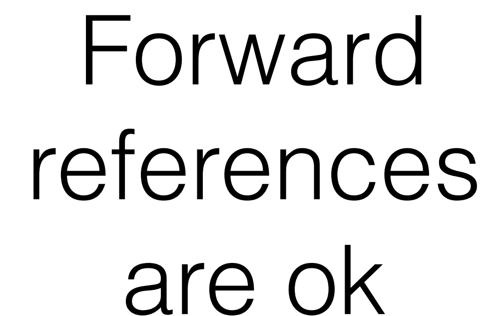
- Use stand alone classes - i.e. stay away from “inheritance hierarchy” - as far as you can.

- A normal JS object can serve as a “module”. Use modules to organize shared code.

- The “module pattern” can be used to hide “private” code. Modules will be standard in ES6.

```
// The “module pattern”.

var VectorMath = (function (exports) {

    function length(vec) { // is private to scope
        return vec.x * vec.x + vec.y * vec.y;
    }

    function dist(v1, v2) {
        return length(relative(v1, v2));
    }

    function relative(v1, v2) {
        return {x: v2.x - v1.x, y: v2.y - v1.y};
    }

    exports.dist = dist;
    exports.relative = relative;
    return exports;

}({}));
```

```
// The “module pattern”.

var VectorMath = (function (exports) {

    function length(vec) { // is private to scope
        return vec.x * vec.x + vec.y * vec.y;
    }

    function dist(v1, v2) {
        return length(relative(v1, v2));
    }

    function relative(v1, v2) {
        return {x: v2.x - v1.x, y: v2.y - v1.y};
    }

    exports.dist = dist;
    exports.relative = relative;
    return exports;

}({}));
```



```javascript
// The “module pattern”.

var VectorMath = (function (exports) {

    // Even this early reference is ok.
    exports.dist = dist;
    exports.relative = relative;

    function length(vec) { // is private to scope
        return vec.x * vec.x + vec.y * vec.y;
    }

    function dist(v1, v2) {
        return length(relative(v1, v2));
    }

    function relative(v1, v2) {
        return {x: v2.x - v1.x, y: v2.y - v1.y};
    }

    return exports;

}({}));
```

Fun exercise:
What are classes, really?

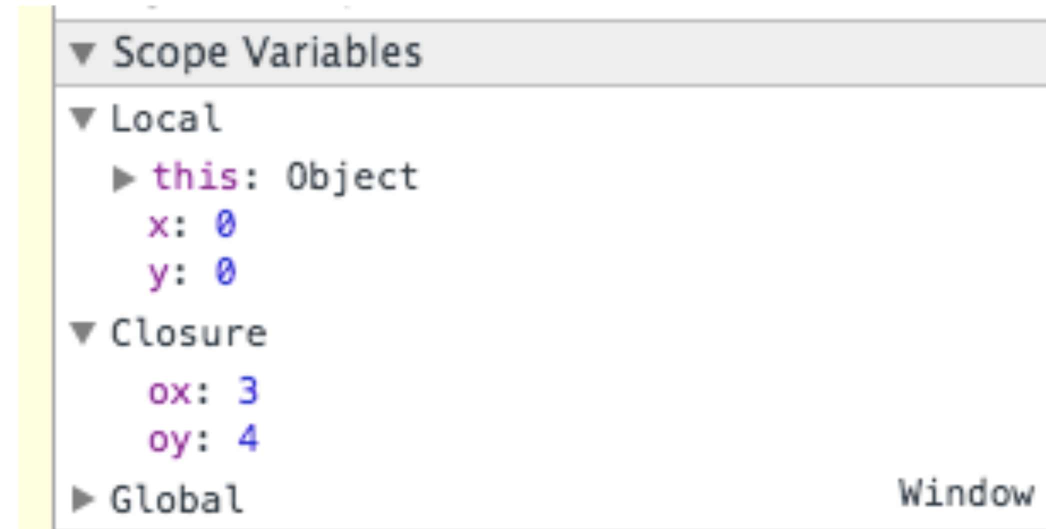
Fun exercise:
How do you model
inheritance?

Fun exercise:
How do you model
multiple inheritance?

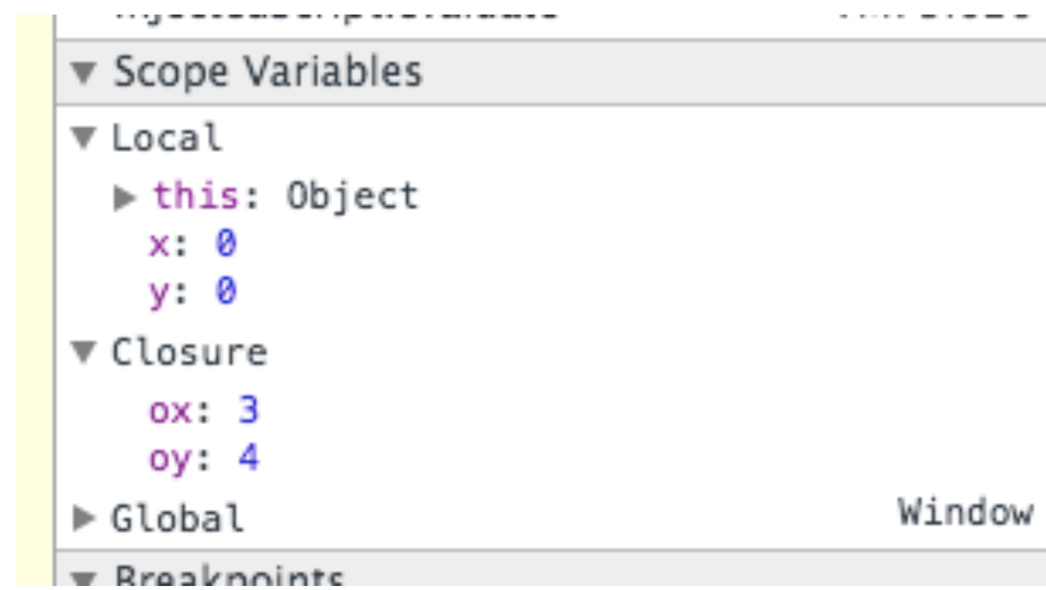
Appendix:
A known problem with today's JS engines

```javascript
function relativeToOrigin(ox, oy) {
    return {
        x0: ox,
        y0: oy,
        dist: function (x, y) {
            debugger;
            return dist(x - ox, y - oy);
        },
        vector: function (x, y) {
            debugger;
            return {x: x - this.x0, y: y - this.y0};
        }
    };
}

var g = relativeToOrigin(3,4);

// Run g.dist(0, 0). What do you see in “Closures”?
// Run g.vector(0, 0). What do you see in “Closures”?
// Notice an issue?
```

g.dist



g.vector



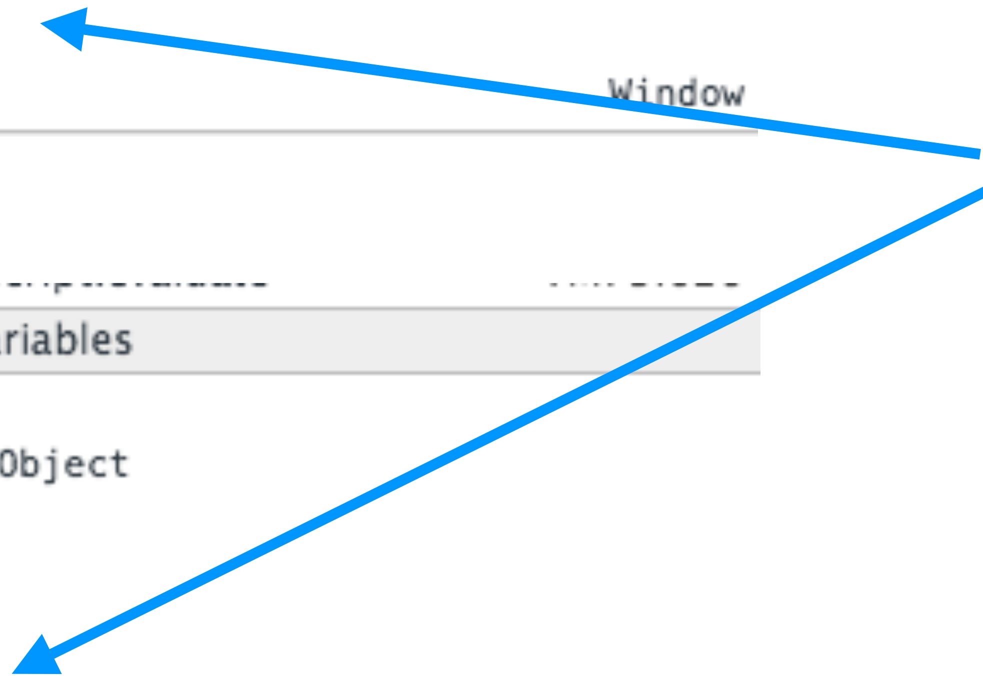
```
function relativeToOrigin(ox, oy) {
    function unused() {
        return ox + oy;
    }
    return {
        x0: ox,
        y0: oy,
        dist: function (x, y) {
            debugger;
            return dist(x - this.x0, y - this.y0);
        },
        vector: function (x, y) {
            debugger;
            return {x: x - this.x0, y: y - this.y0};
        }
    };
}

var g = relativeToOrigin(3,4);

// Run g.dist(0, 0). What do you see in “Closures”?
// Run g.vector(0, 0). What do you see in “Closures”?
```

Note that dist and vector don’t refer to ox and oy

Even then …

g.dist

g.vector



Memory leak!