CSE 341
Lecture 27

JavaScript scope and closures

slides created by Marty Stepp

http://www.cs.washington.edu/341/

Recall: Scope

- **scope**: The enclosing context where values and expressions are associated.
  - essentially, the visibility of various identifiers in a program

- **lexical scope**: Scopes are nested via language syntax; a name refers to the *most local* definition of that symbol.
  - most modern languages (Java, C, ML, Scheme, JavaScript)

- **dynamic scope**: A name always refers to the *most recently executed* definition of that symbol.
  - Perl, Bash shell, Common Lisp (optionally), APL, Snobol

Lexical scope in Java

- In Java, every block ({ }) defines a scope.

```
public class Scope {
    public static int x = 10;

    public static void main(String[] args) {
        System.out.println(x);
        if (x > 0) {
            int x = 20;
            System.out.println(x);
        }
        int x = 30;
        System.out.println(x);
    }
}
```

Lexical scope in JavaScript

- In Java, there are only two scopes:
  - **global scope**: global environment for functions, vars, etc.
  - **function scope**: every function gets its own inner scope

```
var x = 10;                        // foo.js
function main() {
    print(x);
    x = 20;
    if (x > 0) {
        var x = 30;
        print(x);
    }
    var x = 40;
    var f = function(x) { print(x); }
    f(50);
}
```

Another scope example

```
function f() {
    var a = 1, b = 20, c;
    print(a + " " + b + " " + c);        // 1 20 undefined

    // declares g (but doesn't call immediately!)
    function g() {
        var b = 300, c = 4000;
        print(a + " " + b + " " + c);    // 1 300 4000
        a = a + b + c;
        print(a + " " + b + " " + c);    // 4301 300 4000
    }

    print(a + " " + b + " " + c);        // 1 20 undefined
    g();
    print(a + " " + b + " " + c);        // 4301 20 undefined
}
```

Lack of block scope

```
for (var i = 0; i < 10; i++) {
    print(i);
}
print(i);  // 11
if (i > 5) {
    var j = 3;
}
print(j);
```

- any variable declared lives until the end of the function
  - lack of block scope in JS leads to errors for some coders
  - this is a "bad part" of JavaScript (D. Crockford)

The future: let statement

```
var x = 5;      // this code doesn't work today
var y = 0;
var z;
let (x = x + 10, y = 12, z = 3) {
  print(x + " " + y + " " + z);      // 15 12 3
}
print(x + " " + y + " " + z);  // 5 0 undefined
print(let (x = 2, y = 3) x + " " + y);   // 2 3
print(x + " " + y);                       // 5 0
```

- upcoming versions of JS will have block scope using `let`
  - https://developer.mozilla.org/en/New_in_JavaScript_1.7

    (this code does not work yet!)

Implied globals

```
name = value;
```

```
function foo() {
    x = 4;
    print(x);
}   // oops, x is still alive now (global)
```

- if you assign a value to a variable without `var`, JS assumes you want a new *global* variable with that name
  - hard to distinguish
  - this is a "bad part" of JavaScript (D.Crockford)

The global object

- technically *no* JavaScript code is "static" in the Java sense
  - *all* code lives inside of some object
  - there is *always* a `this` reference that refers to that object

- all code is executed inside of a **global object**
  - in browsers, it is also called `window`; in Rhino: `global()`
  - global variables/functions you declare become part of it
    - they use the global object as `this` when you call them

- *"JavaScript's global object [...] is far and away the worst part of JavaScript's many bad parts."* -- D. Crockford

Global object and this keyword

```
function printMe() {
    print("I am " + this);
}
```

```
> var teacher = {...};  // from past lecture
> teacher.print = printMe;
> teacher.print();
I am Prof. Tyler Durden
> print();
I am [object global]
```

Recall: Closures

- **closure**: A first-class function that binds to free variables that are defined in its execution environment.

- **free variable**: A variable referred to by a function that is not one of its parameters or local variables.
  - **bound variable**: A free variable that is given a fixed value when "closed over" by a function's environment.

- A *closure* occurs when a function is defined and it attaches itself to the free variables from the surrounding environment to "close" up those stray references.

Closures in JS

```
var x = 1;
function f() {
    var y = 2;
    return function() {
        var z = 3;
        print(x + y + z);
    };
    y = 10;
}
var g = f();
g();                // 1+10+3 is 14
```

- a function closes over free variables as it is declared
  - grabs references to the names, not values (sees updates)

Declare-and-call pattern

```
(function(params) {
    statements;
})(params);
```

- declares and immediately calls an anonymous function
  - used to create a new **scope** and **closure** around it
  - can help to avoid declaring global variables/functions
  - used by JavaScript libraries to keep global namespace clean

Declare-and-call example

```
// old: 3 globals

var count = 0;
function incr(n) {
  count += n;
}
function reset() {
  count = 0;
}
incr(4);  incr(2);
print(count);
```

```
// new: 0 globals!
(function() {
    var count = 0;
    function incr(n) {
        count += n;
    }
    function reset() {
        count = 0;
    }
    incr(4);  incr(2);
    print(count);
})();
```

- declare-and-call protects your code and avoids globals
  - avoids common problem with namespace/name collisions

Common closure bug

```
var funcs = [];
for (var i = 0; i < 5; i++) {
    funcs[i] = function() { return i; };
}
```

```
> funcs[0]();
5
```

```
> funcs[1]();
5
```

- Closures that bind a loop variable often have this bug.
  - Why do all of the functions return 5?

Fixing the closure bug

```
var funcs = [];
for (var i = 0; i < 5; i++) {
    funcs[i] = (function(n) {
        return function() { return n; }
    })(i);
}
```

```
> funcs[0]();
1
> funcs[1]();
2
```

Objects with public data

```
// BankAccount "invariant": balance >= 0
function BankAccount(name, balance) {
    this.name = name;
    this.balance = Math.max(0, balance);
}
BankAccount.prototype.withdraw = function(amt) {
    if (amt > 0 && amt <= this.balance) {
        this.balance -= amt;
    }
};
```

- clients can directly modify a BankAccount's balance!

```
var ba = new BankAccount("Fred", 50.00);
ba.balance = -10;  // ha ha
```

Objects with private data

```
// BankAccount invariant: balance >= 0
var BankAccount = (function() {
    var name, balance;
    var ctor = function(nam, bal) {
        name = nam;
        balance = Math.max(0, bal);
    };
    ctor.prototype.withdraw = function(amt) {
        if (amt > 0 && amt <= balance) {
            balance -= amt;
        }
    };
    ctor.prototype.getName = function() {return name;}
    ctor.prototype.getBalance = function() {return balance;}
    return ctor;
})();
```

Memoization and "private" data

```
var functionName = (function() {
    1. create "memory" to store results.
    2. create inner function to implement the
                 behavior, using memory as a cache.
    3. return the inner function.
})();
```

- since functions define a scope, we can wrap a function in another one to make its memory a "private" variable
  - only the inner function can see `memory`, since it encloses over `memory` as parts of its closure (bound variable)

** NOTE: Underscore library can do memoization for you ...*

Memoization example

```
var fib = (function() {
    memory = {1:1, 2:1};      // initial memory
    return function(n) {
        var mem = memory[n];
        if (typeof(mem) !== "undefined") {
            return mem;       // re-use past result
        }
        // not in memory; must compute
        var result = fib(n-1) + fib(n-2);
        memory[n] = result;   // remember
        return result;
    };
})();
```