JavaScript Closure

Wikipedia

In programming languages, **closures** (also **lexical closures** or **function closures**) are a technique for implementing lexically scoped name binding in languages with first-class functions.

Operationally, a closure is **a data structure storing a function together with an environment**: a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or storage location the name was bound to at the time the closure was created.

A closure—unlike a plain function—allows the function to access those *captured variables* through the closure's reference to them, even when the function is invoked outside their scope.

Functions as First Class Objects

- Can assign functions to variables
- Can pass functions as arguments
- Can have functions as return values
- Can define function literals

Nested functions with non-local variables provide an implementation challenge.

Lexical Scope

- Name refers to local lexical environment (variable hiding)
- First class nested functions need to carry record of environment variables (closure)

Free variables

Not local nor a parameter, but rather a placeholder for a non-local symbol that will be replace later based on the context.

Scope in JavaScript

- Scope is declared by functions, not blocks
- Global scope acts as one big function encompassing entire page

- Variable declarations are in scope from point of declaration to end of function
- Named functions are in scope within entire function within which they are declared: **hoisting**

				1	**function outer() {**
			2		var a = 1;
		3			function inner() { }
	4				var b = 2;
					if (a==1) {
5					var c = 3;
					}
					}

Four ways of Function invocation

Every function has two implicit members: this and arguments.

“this” an object implicitly associated with the function at invocation: the functional context.

Invoked as:

- a function: context is the global object (window)
- a method: context is object owning method
- a constructor: context is a new object
- a call to “call()” or “apply()”: context is user specified

Function and Method invocation

```
// Normal function call: context is the window
function getContext() { return this; }
getContext();

// Context is still the window
var contextGetter = getContext;
contextGetter();

// Invoke as method: context is owning object
var anObject = { contextGetter: getContext };
anObject;
anObject.contextGetter();
```

Constructor Function Invocation (new)

- A new, empty object is created and passed to the function as the context (this)
- The object is returned if not overridden by an explicit return

```
function Person(name) { this.name = name; }
var victor = new Person(“Victor”);
victor;
```

apply and call

apply takes two parameters: context object, array of invocation arguments

call takes the context object as a parameter, and the arguments directly

(see code sample)

Standard Closure example: Increment

```
function startAt(x) {
   function incrementBy(y) {
       return x + y;
   }
   return incrementBy;
}

var closure1 = startAt(1);
var closure2 = startAt(5);

closure1(1);
closure2(1);
```

Closure example

Callbacks, recursion…

http://stackoverflow.com/questions/2728278/what-is-a-practical-use-for-a-closure-in-javascript

```
for (var i = 0; i < 5; i++) {
    this.setTimeout(function () {
        console.log("Value of i was " + i + " when this timer was set" )
    }, 10000);
};
```

IIFE

Immediately Invoked Function Expression

- Minimize pollution of global namespace
- Pattern for “private” members, modules

```
(function() {
    var x = 1;
})();
```

Closure example take 2

```
for (var i = 0; i < 5; i++) {

    (function (i) {
        this.setTimeout(function () {
            console.log("Value of i was " + i + " when this timer was set" )
        }, 1000);
    })(i);

};
```

Java, stack, lambdas