Here is a good tutorial.

http://robertnyman.com/2008/10/09/explaining-javascript-scope-and-closures/

Even though JavaScript looks like it should have block scope because it uses curly braces { }, a new scope is created only when you execute a new function.

If you have nested functions, the inner function will have access to the containing functions variables and functions:

Example 1:

```
function saveName (firstName) {
        function capitalizeName () {
                return firstName.toUpperCase();
        }
        var capitalized = capitalizeName();
        return capitalized;
}
alert(saveName("Robert")); // Returns "ROBERT"
```

Example 2:

```
function siblings () {
        var siblings = ["John", "Liza", "Peter"];
        function siblingCount () {
                var siblingsLength = siblings.length;
                return siblingsLength;
        }
        function joinSiblingNames () {
                return "I have " + siblingCount() + " siblings:\n\n" +
siblings.join("\n");
        }
        return joinSiblingNames();
}
alert(siblings()); // Outputs "I have 3 siblings: John Liza Peter"
```

What is an anonymous function?

What does it mean to return a function?

Example 3:

```
Function message()
{

      return function(m){alert(m)};

}

a = message();

a("Hello");
```

A closure is a function having access to the parent scope, even after the parent function has closed.

Example 4:

```
function add (x) {

        return function (y) {
                return x + y;
        };
}
var add5 = add(5);
var no8 = add5(3);
alert(no8); // Returns 8
```

When the add function is called, it returns a function.

1. That function closes the context and remembers what the parameter x was at exactly that time (i.e. 5 in the code above)
2. When the result of calling the add function is assigned to the variable add5, it will always know what x was when it was initially created.
3. The add5 variable above refers to a function which will *always* add the value 5 to what is being sent in.
4. That means when add5 is called with a value of 3, it will add 5 together with 3, and return 8.

The add5 function actually looks like this:

```
function add5 (y) {

      return 5 + y;

}
```

Counter Dilemma

Suppose you want to use a variable for counting something, and you want this counter to be available to all functions.

You could use a global variable, and a function to increase the counter:

Example 5:

```
var counter = 0;

function add() {
   counter += 1;
}

add();
add();
add();

// the counter is now equal to 3
```

The counter should only be changed by the add() function.

The problem is, that any script on the page can change the counter, without calling add().

If I declare the counter inside the function, nobody will be able to change it without calling add():

Example 6:

```
function add() {
   var counter = 0;
   counter += 1;
}

add();
add();
add();

// the counter should now be 3, but it does not work !
```

It did not work! Every time I call the add() function, the counter is set to 1.

A self-invoking anonymous runs automatically/immediately when you create it and has no name, hence called anonymous.

```
(function(){
 // some code…
})();
```

Here is an application of the above self-invoking anonymous function;

Example 7:

```
function add() {
    var counter = 0;
    function plus() {counter += 1;}
    plus();
    return counter;
}

var add = (function () {
    var counter = 0;
    return function () {return counter += 1;}
})();

add();
add();
add();

// the counter is now 3
```

The variable add is assigned the return value of a self invoking function.

The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.

This way add becomes a function. The "wonderful" part is that it can access the counter in the parent scope.

This is called a JavaScript **closure.** It makes it possible for a function to have "**private**" variables.

The counter is protected by the scope of the anonymous function, and can only be changed using the add function.

A closure is a function having access to the parent scope, even after the parent function has closed.

Solution to the link problem:

```
function addLinks () {
        for (var i=0, link; i<5; i++) {
                link = document.createElement("a");
                link.innerHTML = "Link " + i;
                link.onclick = function (num) {
                        return function () {
                                alert(num);
                        };
                }(i);
                document.body.appendChild(link);
        }
}
window.onload = addLinks;
```