JavaScript Closures

Closures allow JavaScript programmers to write better code. Creative, expressive, and concise. We frequently use closures in JavaScript, and, no matter your JavaScript experience, you will undoubtedly encounter them time and again.

What is a closure?

A closure is an inner function that has access to the outer (enclosing) function’s variables—scope chain. The closure has three scope chains: it has access to its own scope (variables defined between its curly brackets), it has access to the outer function’s variables, and it has access to the global variables.

The inner function has access not only to the outer function’s variables, but also to the outer function’s parameters. Note that the inner function cannot call the outer function’s arguments object, however, even though it can call the outer function’s parameters directly.

You create a closure by adding a function inside another function.

A Basic Example of Closures in JavaScript:

function showName (firstName, lastName) {

​ var nameIntro = "Your name is ";

*// this inner function has access to the outer function's variables, including the parameter​*

​ function makeFullName () {

​ return nameIntro + firstName + " " + lastName;

}

​ return makeFullName ();

}

showName ("Michael", "Jackson");

*// Your name is Michael Jackson*

Closures are used extensively in Node.js; they are workhorses in Node.js’ asynchronous, non-blocking architecture. Closures are also frequently used in jQuery and just about every piece of JavaScript code you read.

**A Classic jQuery Example of Closures:**

$(function() {

​ var selections = [];

$(".niners").click(function() {

*// this closure has access to the selections variable​*

selections.push (this.prop("name"));

*// update the selections variable in the outer function's scope​*

});

});

Closures’ Rules and Side Effects

Closures have access to the outer function’s variable even after the outer function returns:

One of the most important and ticklish features with closures is that the inner function still has access to the outer function’s variables even after the outer function has returned. Yep, you read that correctly. When functions in JavaScript execute, they use the same scope chain that was in effect when they were created. This means that even after the outer function has returned, the inner function still has access to the outer function’s variables. Therefore, you can call the inner function later in your program. This example demonstrates:

function celebrityName (firstName) {

var nameIntro = "This celebrity is ";

*// this inner function has access to the outer function's variables, including the parameter​*

function lastName (theLastName) {

return nameIntro + firstName + " " + theLastName;

}

return lastName;

}

​var mjName = celebrityName ("Michael"); // At this juncture, the celebrityName outer function has returned.​

*​// The closure (lastName) is called here after the outer function has returned above​*

*​// Yet, the closure still has access to the outer function's variables and parameter​*

mjName ("Jackson"); *// This celebrity is Michael Jackson*

# Closures store references to the outer function’s variables

they do not store the actual value.  Closures get more interesting when the value of the outer function’s variable changes before the closure is called. And this powerful feature can be harnessed in creative ways, such as this private variables example first demonstrated by Douglas Crockford:

function celebrityID () {

var celebrityID = 999;

*// We are returning an object with some inner functions​*

*// All the inner functions have access to the outer function's variables​*

return {

getID: function () {

*// This inner function will return the UPDATED celebrityID variable​*

*// It will return the current value of celebrityID, even after the changeTheID function changes* it​

return celebrityID;

},

setID: function (theNewID) {

*// This inner function will change the outer function's variable anytime​*

celebrityID = theNewID;

}

}

}

​var mjID = celebrityID (); // At this juncture, the celebrityID outer function has returned.​

mjID.getID(); // 999​

mjID.setID(567); // Changes the outer function's variable​

mjID.getID(); // 567: It returns the updated celebrityId variable

# Closures Gone Awry

Because closures have access to the updated values of the outer function’s variables, they can also lead to bugs when the outer function’s variable changes with a for loop. Thus:

*// This example is explained in detail below (just after this code box).​*

​function celebrityIDCreator (theCelebrities) {

var i;

var uniqueID = 100;

for (i = 0; i < theCelebrities.length; i++) {

theCelebrities[i]["id"] = function () {

return uniqueID + i;

}

}

return theCelebrities;

}

​var actionCelebs = [{name:"Stallone", id:0}, {name:"Cruise", id:0}, {name:"Willis", id:0}];

​var createIdForActionCelebs = celebrityIDCreator (actionCelebs);

​var stalloneID = createIdForActionCelebs [0];

console.log(stalloneID.id()); // 103

In the preceding example, by the time the anonymous functions are called, the value of i is 3 (the length of the array and then it increments). The number 3 was added to the uniqueID to create 103 for ALL the celebritiesID. So every position in the returned array get id = 103, instead of the intended 100, 101, 102

The reason this happened was because, as we have discussed in the previous example, the closure (the anonymous function in this example) has access to the outer function’s variables by reference, not by value. So just as the previous example showed that we can access the updated variable with the closure, this example similarly accessed the i variable when it was changed, since the outer function runs the entire for loop and returns the last value of i, which is 103.

To fix this side effect (bug) in closures, you can use an **Immediately Invoked Function Expression** (IIFE), such as the following:

function celebrityIDCreator (theCelebrities) {

var i;

var uniqueID = 100;

for (i = 0; i < theCelebrities.length; i++) {

theCelebrities[i]["id"] = function (j) {

*// the j parametric variable is the i passed in on invocation of this IIFE​*

return function () {

return uniqueID + j;

*// each iteration of the for loop passes the current value of i into this IIFE and it saves the correct value to the array​*

} ()

*// BY adding () at the end of this function, we are executing it immediately and returning just the value of uniqueID + j, instead of returning a function.​*

} (i);

*// immediately invoke the function passing the i variable as a parameter*​

}

​

return theCelebrities;

}

​

​var actionCelebs = [{name:"Stallone", id:0}, {name:"Cruise", id:0}, {name:"Willis", id:0}];

​

​var createIdForActionCelebs = celebrityIDCreator (actionCelebs);

​

​var stalloneID = createIdForActionCelebs [0];

 console.log(stalloneID.id); // 100​

​

​var cruiseID = createIdForActionCelebs [1]; console.log(cruiseID.id); // 101