Closures with JavaScript

Dominic Burford, 16 Jul 2015 CPOL

5.00 (2 votes)

Rate:

Closures with JavaScript

Introduction

Following on from my previous article Prototypal inheritance with JavaScript, another area of the Javascipt programming language that can seem confusing, especially to a developer coming from a common programming language (such as C, C++, C# and Java), is the concept of *closure*.

A *closure* defines a function and the environment in which the function is defined. This environment includes *free variables* i.e. those variables that are not local to a function nor a parameter to a function but still part of the environment of the function. For example a global variable is a free variable. Although not local to the function it nonetheless is part of the function's environment as the function can access its value.

Closures and scope

Closures are related to the concept of *scope*. In a programming language the concept of *scope* refers to the visibility and lifetime of variables and parameters.

Common programming languages such as C, C++, C# and Java denote scope using curly braces. The curly braces define a block of code, hence these languages implement what is known as block scope. A block is a function and these functions define the lifetime (or scope) of the variables defined within them, and the parameters that are passed to them.

Any programmer coming from a common language will be used to block scope, and this can cause confusion, as JavaScript does not implement block scope. Instead, JavaScript implements what is known as function scope. As you would expect from a language that defines first class functions, the scope of variables and parameters is bound to the scope of the function.

In practice, what this means is that variables and parameters defined within a function are not visible outside of the function, and that a variable defined anywhere within a function is visible everywhere within the function.

To understand this let's consider the following example.

Hide Copy Code

```
var someFunction = function(){
  var a = 3, b = 5;

  var someOtherFunction = function(){
    var b = 7, c = 11;

    //at this point a is 3, b is 7 and c is 11

    a += b + c;

    //at this point a is 21, b is 7 and c is 11
  };

  //at this point a is 3, b is 5 and c is not defined

  someOtherFunction();

  //at this point a is 21, b is 5
};
```

In most common programming languages it is often best practice to define a variable as late as possible, just before it is required by the code. In JavaScript however, because it lacks block scope, it is best practice to define variables at the top of the function.

Closures in JavaScript

Now that we understand how scope works within JavaScript, we can continue with how *closures* work, as the two are closely related to each other.

Closures are found in programming languages which are capable of defining first-class functions such as JavaScript. In JavaScript, functions can be passed as parameters to other functions and assigned to variables.

If you program in a common programming language then you will know that a function defines a *closure* i.e. all variables defined within the function are not accessible outside the function.

However, things work slightly differently in JavaScript, and this can confuse even a seasoned developer.

Simple Example

Consider the following simple example.

Hide Copy Code

```
function mainFunction() {
  var name = "Dominic";
  function displayText() {
    alert(name);
  }
  return displayText
}

var myFunc = mainFunction();
myFunc();
```

The code above displays the text *Dominic* in a JavaScript alert box. This is unintuitive at first, as variables defined in one function are not accessible to another function. What is also interesting is that the `displayText` inner function was returned from the outer function before being executed.

This is easily explained when we learn that the function `displayText` is in fact a *closure*. It defines both the function and the environment in which it was created, which in the case of JavaScript includes all local variables that are in scope at the time the *closure* was defined. So in our simple example, the function `displayText` is a *closure* that incorporates the `name` variable.

In JavaScript, inner functions have access to variables defined in their outer functions. **Whenever you define a function inside another function you are actually creating a *closure***. In other common programming languages, when you return from a function, all the local variables are no longer accessible because the stack has deallocated them. In JavaScript however, when you declare a function within another function, the local variables can remain accessible after returning from the function.

Here is another simple example that demonstrates that behaviour.

Hide Copy Code

```
var before = 100;
function testFunction() {
  alert(before); // will output 100
  alert(after); // will output 99
}
var after = 99;
test();
```

When a Javascrpt function is invoked, a new *excution context* is created. This *execution context* is the environment in which the function is invoked. In the case of the function `testFunction` above this includes both the variables `before` and `after`.

It's important to note that the local variables are not copies but references. The following example demonstrates this behaviour.

Hide Copy Code

```
function output100() {
    // Local variable that ends up within closure
    var num = 99;
    var sayAlert = function() { alert(num); }
    num++;
    return sayAlert;
}
var sayNumber = output100();
sayNumber(); // alerts 100
```

We can summarise ALL of the above by stating that a **closure is the local variables to a function - kept alive after the function has returned**.

If you are writing code using JavaScript then I suggest you take the time to fully comprehend *closures* and how they relate to JavaScript. Otherwise you will find yourself creating bugs that are difficult to diagnose.

Summary

Closures within JavaScript can be confusing, especially if coming from a common programming language. The best advice I can give is to play around with some of the code examples I have given here until you have grasped them. Feel free to leave a comment if you would like me to further elaborate on anything within this article.

License

Share

About the Author

Dominic Burford

Software Developer (Senior) Grosvenor Contracts
United Kingdom

I am a professional software engineer and architect with over eighteen years commercial development experience with a strong focus on the design and development of web and mobile applications.

I have experience of architecting scalable, distributed, high volume web applications that are accessible from multiple devices due to their responsive web design, including architecting enterprise service-oriented solutions. I have also developed enterprise mobile applications using Xamarin and Telerik Platform.

I have extensive experience using .NET, ASP.NET, Windows and Web Services, WCF, SQL Server, LINQ and other Microsoft technologies. I am also familiar with HTML, Bootstrap, Javascript (inc. JQuery and Node.js), CSS, XML, JSON, Apache Cordova, KendoUI and many other web and mobile related technologies.

I am enthusiastic about Continuous Integration, Continuous Delivery and Application Life-cycle Management having configured such environments using CruiseControl.NET, TeamCity and Team Foundation Services. I enjoy working in Agile and Test Driven Development (TDD) environments.

Outside of work I have two beautiful daughters. I enjoy cycling, running and taking the dog for long walks. I love listening to music and am a fan of Rush and the Red Hot Chilli Peppers to name a few.

Comments and Discussions

Add a Comment or Question

Search Comments Go

First Prev Next

Performace implications new

tecnocrata 16-Jul-15 18:25

Hey, great article!
I have heard that closure have performance implications
Could you please explain a bit more about that? of course if you also think that is true

Thanks!

Reply · Email · View Thread · Permalink · Bookmark

Re: Performace implications new

Dominic Burford 17-Jul-15 0:02

Whilst closures are a powerful of Javascript, creating a closure can be significantly slower than creating an inner function without a closure, and much slower still than reusing a static function. For example.

Hide Copy Code

```
function showAlertMessage() {
  var msg = 'Hello Dominic';
  window.setTimeout(function() { alert(msg); }, 100);
}
```

is slower than

Hide Copy Code

```
function showAlertMessage() {
  window.setTimeout(function() {
    var msg = 'Hello Dominic';
    alert(msg);
  }, 100);
}
```

which is slower than

Hide Copy Code

```
function alertMsg() {
  var msg = 'Hello Dominic';
  alert(msg);
}

function showAlertMessage() {
  window.setTimeout(alertMsg, 100);
}
```

So whilst closures can be a useful feature, make sure you use them wisely.

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult." - C.A.R. Hoare

Home | LinkedIn | Google+ | Twitter

Reply · Email · View Thread · Permalink · Bookmark

Refresh 1

General News Suggestion Question Bug Answer Joke Praise Rant Admin