developerWorks®

Memory leak patterns in JavaScript

Handling circular references in JavaScript applications

Abhijeet Bhattacharya
Kiran Shivarama Shivarama Sundar

April 24, 2007

Plugging memory leaks in JavaScript is easy enough when you know what causes them. In this article authors Kiran Sundar and Abhijeet Bhattacharya walk you through the basics of circular references in JavaScript and explain why they can cause problems in certain browsers, especially when combined with closures. After seeing some of the common memory leak patterns you should watch out for, you'll learn a variety of easy ways to work around them.

JavaScript is a powerful scripting language used to add dynamic content to Web pages. It is especially beneficial for everyday tasks such as password validation and creating dynamic menu components. While JavaScript is easy to learn and write, it is prone to memory leaks in certain browsers. In this introductory article we explain what causes memory leaks in JavaScript, demonstrate some of the common memory leak patterns to watch out for, and show you how to work around them.

Note that the article assumes you are familiar with using JavaScript and DOM elements to develop Web applications. The article will be most useful to developers who use JavaScript for Web application development. It might also serve as a reference for those providing browser support to clients rolling out Web applications or for anyone tasked with troubleshooting browser issues.

Is my browser leaking?

Internet Explorer and Mozilla Firefox are the two Web browsers most commonly associated with memory leaks in JavaScript. The culprit in both browsers is the component object model used to manage DOM objects. Both the native Windows COM and Mozilla's XPCOM use reference-counting garbage collection for memory allocation and retrieval. Reference counting is not always compatible with the mark-and-sweep garbage collection used for JavaScript. This article focuses on ways to work around memory leaks in JavaScript code. See Related topics to learn more about COM layer memory handling in Firefox and IE.

Memory leaks in JavaScript

JavaScript is a garbage collected language, meaning that memory is allocated to objects upon their creation and reclaimed by the browser when there are no more references to them. While there is nothing wrong with JavaScript's garbage collection mechanism, it is at odds with the way some browsers handle the allocation and recovery of memory for DOM objects.

Internet Explorer and Mozilla Firefox are two browsers that use reference counting to handle memory for DOM objects. In a reference counting system, each object referenced maintains a count of how many objects are referencing it. If the count becomes zero, the object is destroyed and the memory is returned to the heap. Although this solution is generally very efficient, it has a blind spot when it comes to circular (or *cyclic*) references.

What's wrong with circular references?

A circular reference is formed when two objects reference each other, giving each object a reference count of 1. In a purely garbage collected system, a circular reference is not a problem: If neither of the objects involved is referenced by any other object, then both are garbage collected. In a reference counting system, however, neither of the objects can be destroyed, because the reference count never reaches zero. In a hybrid system, where both garbage collection and reference counting are being used, leaks occur because the system fails to identify a circular reference. In this case, neither the DOM object nor the JavaScript object is destroyed. Listing 1 shows a circular reference between a JavaScript object and a DOM object.

Listing 1. A circular reference resulting in a memory leak

```
<html>
     <body>
     <script type="text/javascript">
     document.write("Circular references between JavaScript and DOM!");
     var obj;
     window.onload = function(){
 obj=document.getElementById("DivElement");
            document.getElementById("DivElement").expandoProperty=obj;
            obj.bigString=new Array(1000).join(new Array(2000).join("XXXXX"));
            };
     </script>
     <div id="DivElement">Div Element</div>
     </body>
     </html>
```

As you can see in the above listing, the JavaScript object `obj` has a reference to the DOM object represented by `DivElement`. The DOM object, in turn, has a reference to the JavaScript object through the `expandoProperty`. A circular reference exists between the JavaScript object and the DOM object. Because DOM objects are managed through reference counting, neither object will ever be destroyed.

Another memory leak pattern

In Listing 2 a circular reference is created by calling the external function `myFunction`. Once again the circular reference between a JavaScript object and a DOM object will eventually lead to a memory leak.

Listing 2. A memory leak caused by calling an external function

```
<html>
<head>
<script type="text/javascript">
document.write("Circular references between JavaScript and DOM!");
function myFunction(element)
{
 this.elementReference = element;
 // This code forms a circular reference here
```

```
 //by DOM-->JS-->DOM
 element.expandoProperty = this;
}
function Leak() {
 //This code will leak
 new myFunction(document.getElementById("myDiv"));
}
</script>
</head>
<body onload="Leak()">
<div id="myDiv"></div>
</body>
</html>
```

As these two code samples show, circular references are easy to create. They also tend to crop up quite a bit in one of JavaScript's most convenient programming constructs: closures.

Closures in JavaScript

One of JavaScript's strengths is that it allows functions to be nested within other functions. A nested, or inner, function can inherit the arguments and variables of its outer function, and is private to that function. Listing 3 is an example of an inner function.

Listing 3. An inner function

```
function parentFunction(paramA)
{
     var a = paramA;
     function childFunction()
     {
  return a + 2;
     }
     return childFunction();
}
```

JavaScript developers use inner functions to integrate small utility functions within other functions. As you can see in Listing 3, the inner function `childFunction` has access to the variables of the outer `parentFunction`. When an inner function gains and uses access to its outer function's variables it is known as a *closure*.

Learning about closures

Consider the code snippet shown in Listing 4.

Listing 4. A simple closure

```
<html>
<body>
<script type="text/javascript">
document.write("Closure Demo!!");
window.onload=
function  closureDemoParentFunction(paramA)
{
    var a = paramA;
    return function closureDemoInnerFunction (paramB)
    {
       alert( a +" "+ paramB);
    };
};
var x = closureDemoParentFunction("outer x");
x("inner x");
</script>
</body>
</html>
```

In the above listing `closureDemoInnerFunction` is the inner function defined within the parent function `closureDemoParentFunction`. When a call is made to `closureDemoParentFunction` with a parameter of *outer x*, the outer function variable *a* is assigned the value *outer x*. The function returns with a pointer to the inner function `closureDemoInnerFunction`, which is contained in the variable *x*.

It is important to note that the local variable *a* of the outer function `closureDemoParentFunction` will exist even after the outer function has returned. This is different from programming languages such as C/C++, where local variables no longer exist once a function has returned. In JavaScript, the moment `closureDemoParentFunction` is called, a scope object with property *a* is created. This property contains the value of *paramA*, also known as *"outer x"*. Similarly, when the `closureDemoParentFunction` returns, it will return the inner function `closureDemoInnerFunction`, which is contained in the variable *x*.

Because the inner function holds a reference to the outer function's variables, the scope object with property *a* will not be garbage collected. When a call is made on *x* with a parameter value of *inner x* -- that is, `x("inner x")` -- an alert showing "*outer x innerx*" will pop up.

Listing 4 is a very simple illustration of a JavaScript closure. Closures are powerful because they enable inner functions to retain access to an outer function's variables even after the outer function has returned. Unfortunately, closures are excellent at hiding circular references between JavaScript objects and DOM objects.

Closures and circular references

In Listing 5 you see a closure in which a JavaScript object (`obj`) contains a reference to a DOM object (referenced by the id `"element"`). The DOM element, in turn, has a reference to the JavaScript `obj`. The resulting circular reference between the JavaScript object and the DOM object causes a memory leak.

Listing 5. Event handling memory leak pattern

```
<html>
<body>
<script type="text/javascript">
document.write("Program to illustrate memory leak via closure");
window.onload=function outerFunction(){
 var obj = document.getElementById("element");
 obj.onclick=function innerFunction(){
 alert("Hi! I will leak");
 };
 obj.bigString=new Array(1000).join(new Array(2000).join("XXXXX"));
 // This is used to make the leak significant
};
</script>
<button id="element">Click Me</button>
</body>
</html>
```

Avoiding memory leaks

The upside of memory leaks in JavaScript is that you can avoid them. When you have identified the patterns that can lead to circular references, as we've done in the previous sections, you can begin to work around them. We'll use the above event-handling memory leak pattern to demonstrate three ways to work around a known memory leak.

One solution to the memory leak in Listing 5 is to make the JavaScript object `obj` null, thus explicitly breaking the circular reference, as shown in Listing 6.

Listing 6. Break the circular reference

```
<html>
<body>
<script type="text/javascript">
document.write("Avoiding memory leak via closure by breaking the circular
   reference");
 window.onload=function outerFunction(){
 var obj = document.getElementById("element");
 obj.onclick=function innerFunction()
 {
  alert("Hi! I have avoided the leak");
  // Some logic here
 };
 obj.bigString=new Array(1000).join(new Array(2000).join("XXXXX"));
 obj = null; //This breaks the circular reference
 };
</script>
<button id="element">"Click Here"</button>
</body>
</html>
```

In Listing 7 you avoid the circular reference between the JavaScript object and the DOM object by adding another closure.

Listing 7. Add another closure

```
<html>
<body>
<script type="text/javascript">
document.write("Avoiding a memory leak by adding another closure");
 window.onload=function outerFunction(){
var anotherObj = function innerFunction()
   {
   // Some logic here
   alert("Hi! I have avoided the leak");
     };
  (function anotherInnerFunction(){
  var obj =  document.getElementById("element");
  obj.onclick=anotherObj })();
     };
</script>
<button id="element">"Click Here"</button>
</body>
</html>
```

In Listing 8 you avoid the closure itself by adding another function, thereby preventing the leak.

Listing 8. Avoid the closure altogether

```
<html>
<head>
<script type="text/javascript">
document.write("Avoid leaks by avoiding closures!");
window.onload=function()
{
 var obj = document.getElementById("element");
 obj.onclick = doesNotLeak;
}
function doesNotLeak()
{
 //Your Logic here
 alert("Hi! I have avoided the leak");
}

</script>
</head>
<body>
<button id="element">"Click Here"</button>
</body>
</html>
```

In conclusion

This article has explained how circular references can lead to memory leaks in JavaScript, particularly when combined with closures. You've seen several common memory leak patterns involving circular references and some easy ways to work around them. See Related topics to learn more about the topics discussed in this introductory article.

Related topics

- "JavaScript and the Document Object Model" (Nicholas Chase, developerWorks, July 2002): Introduces the DOM to JavaScript developers.
- "Crossing borders: Closures" (Bruce Tate, developerWorks, January 2007): A primer on the many uses of closures (based on Ruby but conceptually applicable to JavaScript).
- "Finite state machines in JavaScript, Part 1: Design a widget" (Edward J. Pring, developerWorks, January 2007): A fun exercise using closures and other advanced features of JavaScript.
- "A re-introduction to javascript" (Simon Wilson, Mozilla.org): A closer look at JavaScript and its features.
- "Using XPCOM in JavaScript without leaking" (David Baron, Mozilla.org): Explains why and how Firefox uses reference counting for memory allocation.
- "Memory leakage in Internet Explorer -- revisited" (Volkan Ozcelik, The Code Project, November 2005): A tutorial introduction to the common causes of memory leaks in JavaScript, with reference to IE.
- IBM product evaluation versions: Get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.