Understand memory leaks in JavaScript applications

Detect and address memory issues

Ben Dolmar (Ben.Dolmar@nerdery.co)
Software Developer
The Nerdery

06 November 2012

Garbage collection can be freeing. It lets us focus on application logic rather than memory management. However, garbage collection is not magic. Understanding how it works, and how it can be tricked into maintaining memory long after it should have been released, results in faster and more reliable applications. In this article, learn about a systematic approach for locating memory leaks in JavaScript applications, several common leaking patterns, and appropriate methods to address those leaks.

Introduction

When dealing with a scripting language like JavaScript, it's easy to forget that every object, class, string, number, and method requires that memory be allocated and retained. The specifics of that allocation and its deallocation are hidden by the language and the runtime's garbage collector.

You can achieve a lot without ever considering memory management, but ignoring it can lead to significant issues in a program. Improperly cleaned-up objects can linger far longer than intended. Those objects continue responding to events and consuming resources. They can force the browser to page memory from a virtual disk drive and significantly slow down the computer (and, in extreme cases, crash the browser).

A memory leak is any object that persists after you no longer have a use or need for it. In recent years, many browsers have gotten better about reclaiming memory from JavaScript between page loads. Not all browsers behave the same way, though. Both Firefox and older versions of Internet Explorer have a history of memory leaks that would persist until the browser was closed.

Many classic patterns that historically caused memory leaks no longer leak in modern browsers. However, today there is a different trend that affects memory leakage. Many people are designing web applications that are intended to run within the context of a single page with no hard page refreshes. In that context, it's easy to retain memory from one state of the application to another when it's no longer needed, or relevant.

In this article, learn about the basic lifecycle of an object, how garbage collection determines whether an object can be freed, and how to evaluate potential leaking behaviors. Also, learn how to use the Heap Profiler in Google Chrome to diagnose memory issues. Examples show how to address memory leaks with closures, the console log, and with cycles.

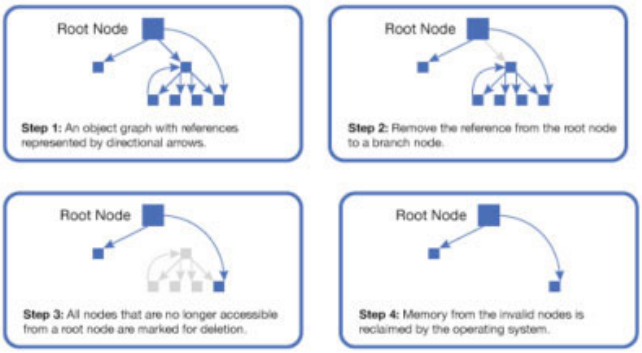
You can download the source code for the examples that are used in this article.

Object lifecycle

To understand the prevention of memory leaks, it's important to understand the basic lifecycle of an object. When an object is created, JavaScript automatically allocates an appropriate amount of memory for that object. From that point on, the object is continually evaluated by the garbage collector to see if it is still a valid object.

At regular intervals, the garbage collector sweeps through the object graph and counts the number of other objects that have a reference to each object. If an object has a count of zero (no other object has a reference to it), or if the only references to it are circular, the object's memory can be reclaimed. Figure 1 shows an example of how a garbage collector reclaims memory.

Figure 1. Reclaiming memory with garbage collection



It would be helpful to actually see the system in action, but the tools to do so are limited. One way to get a sense of how much memory your JavaScript application consumes is to use the system tools to observe the browser's memory allocation. There are several tools available that will give you the current level of use and will graph the memory usage of a process over time.

For example, if you have installed XCode on Mac OSX you can launch the Instruments application and attach its activity monitor tool to your browser for real-time analytics. On Windows®, you can use the Task Manager. If you find that the graph of memory usage over time is consistently stair-stepping up as you move around the application, you know that you have a memory leak.

Observing the browser's memory footprint is a very rough proxy for the actual memory usage of your JavaScript application. The browser data doesn't tell you which objects are being leaked, and it isn't guaranteed that the data actually matches the true footprint of your application. And, due to implementation issues in some browsers, DOM elements (or backing application-level objects)

may not be released when the corresponding element is destroyed in the page. This is particularly true for a video tag, which requires a more elaborate infrastructure for the browser to implement.

There have been several attempts to add tracking for memory allocation from within client-side JavaScript libraries. Unfortunately, none of those attempts has been particularly reliable. For example, the popular stats.js package dropped support due to inaccuracy. Generally, trying to maintain or determine this information from the client is problematic because it introduces overhead in the application and cannot be reliably determined.

The ideal solution is for browser vendors to provide a set of tools in the browser that help you monitor memory use, identify leaked objects, and determine why a particular object is still marked for retention.

Currently, the only browser to implement a memory management tool as a part of its developer tools is Google Chrome, which offers the Heap Profiler. I use the Heap Profiler in this article to test and illustrate how the JavaScript runtime handles memory.

Analyzing heap snapshots

Before creating a memory leak, look at a simple interaction where memory is properly collected. Begin by creating a simple HTML page with two buttons, as in Listing 1.

Listing 1. index.html

```
<html>
<head>
    <script src="//ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"
type="text/javascript"></script>
</head>
<body>
    <button id="start_button">Start</button>
    <button id="destroy_button">Destroy</button>
    <script src="assets/scripts/leaker.js" type="text/javascript"
charset="utf-8"></script>
    <script src="assets/scripts/main.js" type="text/javascript"
charset="utf-8"></script>
</body>
</html>
```

jQuery was included to ensure a simple syntax for managing event binding that works well across browsers and closely parallels the most common development practices. Scripts tags were added for the `leaker` class and for the main JavaScript method. In production, it is generally a better practice to consolidate your JavaScript files to a single file. For the purposes of this example, it's easier to keep the logic in separate files.

You can filter the Heap Profiler to show only instances of particular classes. To take advantage of that function, create a new class that encapsulates the behavior of the leaking object and that can be easily found in the profiler, as in Listing 2.

Listing 2. assets/scripts/leaker.js

```
var Leaker = function(){};
Leaker.prototype = {
    init:function(){

    }
};
```

Bind the start button to initialize a `Leaker` object and assign it to a variable in the global namespace. You're also going to bind the destroy button to a method that should clean up the `Leaker` object and make it ready for garbage collection, as in Listing 3.

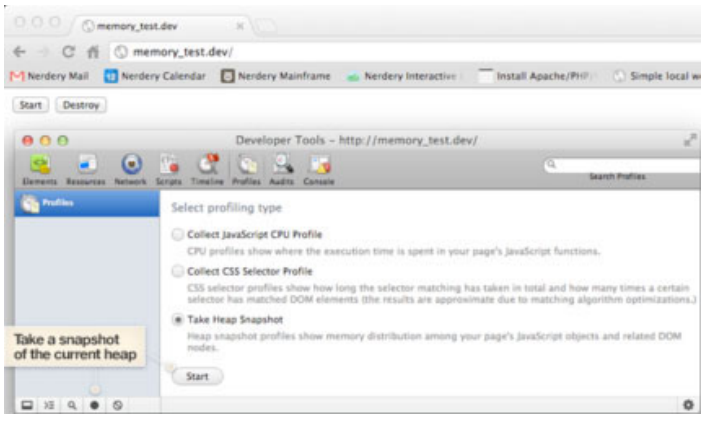
Listing 3. assets/scripts/main.js

```
$("#start_button").click(function(){
    if(leak !== null || leak !== undefined){
        return;
    }
  leak = new Leaker();
  leak.init();
});

$("#destroy_button").click(function(){
    leak = null;
});

var leak = new Leaker();
```

At this point, you're ready to create an object, observe it in memory, and then free it.

1. Load the index page in Chrome.
   Because you're loading jQuery directly from Google, an Internet connection is required to run the sample.
2. Open the developer tools by opening the View menu and selecting the Develop submenu. Select the **Developer Tools** command.
3. Go to the **Profiles** tab and take a heap snapshot, as shown in Figure 2.

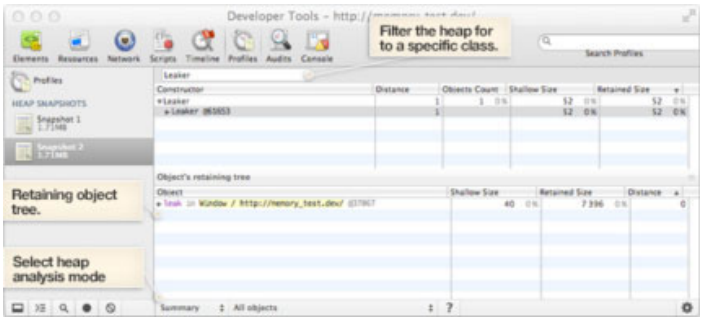
   ### Figure 2. Profiles tab

4. Return focus to the web page and select **Start**.
5. Take another heap snapshot.

6. Filter the first snapshot looking for instances of the `Leaker` class. You should find no instances. Switch to the second snapshot, and you should find one instance, as in Figure 3.

   Figure 3. Snapshot instance



7. Return focus to the web page and select **Destroy**.
8. Take a third heap snapshot.
9. Filter the third snapshot looking for instances of the `Leaker` class. You should find no instances.
   Alternately, with the third snapshot loaded, switch analysis modes from Summary to Comparison and compare the third and second snapshots. You should see a delta of -1 (one instance of the `Leaker` object was released between the two snapshots).

Hooray! Garbage collection works. Now it's time to break it.

Memory leak 1: Closures

One easy way to prevent an object from being garbage-collected is to have an interval or timeout that references the object in its callback. To see this in action, update the leaker.js class as in Listing 4.

Listing 4. assets/scripts/leaker.js

```
var Leaker = function(){};

Leaker.prototype = {
    init:function(){
        this._interval = null;
        this.start();
    },

    start: function(){
        var self = this;
        this._interval = setInterval(function(){
            self.onInterval();
        }, 100);
    },

    destroy: function(){
        if(this._interval !== null){
            clearInterval(this._interval);
        }
    },

    onInterval: function(){
```

```
        console.log("Interval");
    }
};
```

Now, when you repeat steps 1-9 in the section above, you should see that in snapshot three the `Leaker` object has persisted and the interval continues running forever. So what happened? Any local variable that is referenced in a closure is retained by the closure for as long as the closure exists. To ensure that the callback for the `setInterval` method executed with access to Leaker instance's scope, the `this` variable was assigned to the local variable `self`, which was used to trigger `onInterval` from within the closure. When `onInterval` fires, it has access to any instance variables in the `Leaker` object including self. However, the `Leaker` object is not garbage collected for as long as the event listener exists.

To clean up the issue, trigger the `destroy` method that was added to the `leaker` object before nulling the stored reference to it, by updating the click handler for the destroy button, as in Listing 5.

Listing 5. assets/scripts/main.js

```
$("#destroy_button").click(function(){
    leak.destroy();
    leak = null;
});
```

Destroying objects and object ownership

It's good practice to have a standard method that is responsible for making an object eligible for garbage collection. The primary purpose of a destroy function is to centralize the responsibility for cleaning up anything that the object has done that will:

- Prevent its reference count from dropping to 0 (for example, removing problematic event listeners and callbacks and unregistering from any services).
- Consume unnecessary CPU cycles, such as intervals or animations.

The `destroy` method is often a necessary step toward cleaning up an object, but it is rarely sufficient. Other objects that retain a reference to the destroyed object could, in theory, call methods on it after the instance has been destroyed. Because this situation can lead to very unpredictable results, it's critical that a destroy method be called only when the object really is about to go away.

Generally, destroy methods are best when there is one clear owner for an object responsible for its lifecycle. This situation occurs frequently in hierarchical systems, such as views and controllers in MVC frameworks or a scene graph for a canvas rendering system.

Memory leak 2: Console log

One particularly obscure way to retain an object in memory is to log it to the console. Listing 6 updates the `Leaker` class to show an example of this.

Listing 6. assets/scripts/leaker.js

```
var Leaker = function(){};

Leaker.prototype = {
    init:function(){
        console.log("Leaking an object: %o", this);
    },

    destroy: function(){

    }
};
```

You can demonstrate the effect of the console by taking the following steps.

1. Load the index page.
2. Click **Start**.
3. Go to the console and verify that the Leaking object was traced.
4. Click **Destroy**.
5. Go back to the console and type `leak` to log the current contents of the global variable. The value should be null at this point.
6. Take another heap snapshot and filter for the Leaker object.
   You should have one `Leaker` left.
7. Go back to the console and clear it.
8. Take one more heap profile.
   The one remaining leaker should have been cleaned up after the console was cleared.

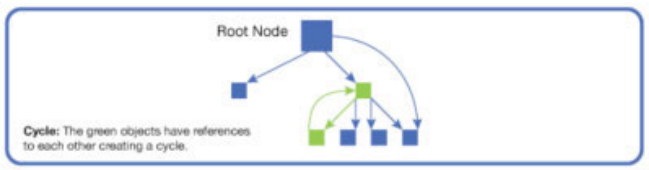
The effects of console logging on the overall memory profile could potentially be a very significant issue that many developers don't even consider. Logging the wrong object can keep large chunks of data in memory. It's important to note that this also applies to:

- Objects that are logged during an interactive session in the console where the user is typing in JavaScript.
- Objects that are logged by the `console.log` and `console.dir` methods.

Memory leak 3: Cycles

A cycle happens when two objects reference each other in such a way that both objects retain each other, as in Figure 4.

Figure 4. References creating a cycle



Listing 7 shows a simple code example.

Listing 7. assets/scripts/leaker.js

```
var Leaker = function(){};

Leaker.prototype = {
    init:function(name, parent){
        this._name = name;
        this._parent = parent;
        this._child = null;
        this.createChildren();
    },

    createChildren:function(){
        if(this._parent !== null){
            // Only create a child if this is the root
            return;
        }
        this._child = new Leaker();
        this._child.init("leaker 2", this);
    },

    destroy: function(){

    }
};
```

The instantiation of the root object would be modified, as in Listing 8.

Listing 8. assets/scripts/main.js

```
leak = new Leaker();
leak.init("leaker 1", null);
```

If you do a heap analysis after creating and destroying the objects, you should see that the garbage detector detected the circular reference and freed the memory when you selected the destroy button.

However, if a third object is introduced that retains the child, the cycle results in a memory leak. For example, create a `registry` object as in Listing 9.

Listing 9. assets/scripts/registry.js

```
var Registry = function(){};

Registry.prototype = {
    init:function(){
        this._subscribers = [];
    },

    add:function(subscriber){
        if(this._subscribers.indexOf(subscriber) >= 0){
            // Already registered so bail out
            return;
        }
        this._subscribers.push(subscriber);
    },

    remove:function(subscriber){
```

```
        if(this._subscribers.indexOf(subscriber) < 0){
            // Not currently registered so bail out
            return;
        }
              this._subscribers.splice(
                  this._subscribers.indexOf(subscriber), 1
              );
    }
};
```

The `registry` class is a simple example of an object that lets other classes register with it and then remove themselves from the registry. While this particular class doesn't do anything with the registry, this is a common pattern in event dispatchers and notification systems.

Import that class into the index.html page before the leaker.js, as in Listing 10.

Listing 10. index.html

```
<script src="assets/scripts/registry.js" type="text/javascript"
charset="utf-8"></script>
```

Update the `Leaker` object to register itself with the registry object (presumably for notification about some unimplemented event). This creates an alternate path from the root node for the child leaker to be retained and because of the cycle, the parent will also be retained, as in Listing 11.

Listing 11. assets/scripts/leaker.js

```
var Leaker = function(){};
Leaker.prototype = {

    init:function(name, parent, registry){
        this._name = name;
        this._registry = registry;
        this._parent = parent;
        this._child = null;
        this.createChildren();
        this.registerCallback();
    },

    createChildren:function(){
        if(this._parent !== null){
            // Only create child if this is the root
            return;
        }
        this._child = new Leaker();
        this._child.init("leaker 2", this, this._registry);
    },

    registerCallback:function(){
        this._registry.add(this);
    },

    destroy: function(){
        this._registry.remove(this);
    }
};
```

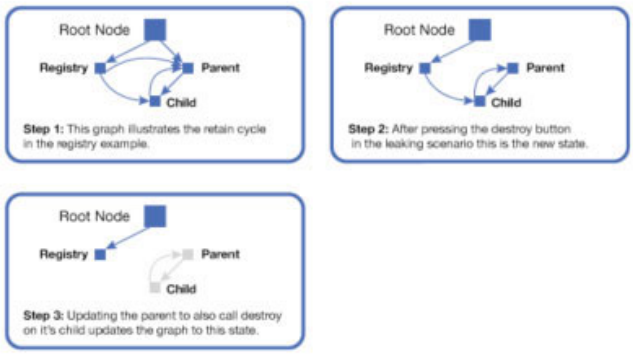
Finally, update main.js to set up the registry and pass a reference to the registry to the parent `leaker` object, as in Listing 12.

Listing 12. assets/scripts/main.js

```
   $("#start_button").click(function(){
  var leakExists = !(
       window["leak"] === null || window["leak"] === undefined
   );
  if(leakExists){
      return;
  }
  leak = new Leaker();
  leak.init("leaker 1", null, registry);
});

$("#destroy_button").click(function(){
    leak.destroy();
    leak = null;
});

registry = new Registry();
registry.init();
```

Now, when you do a heap analysis, you should see that every time you select the start button two new instances of the `Leaker` object are created and retained. Figure 5 shows the flow of the object references.

Figure 5. Memory leak due to retained references



On the surface it might seem like a contrived example, but it's actually quite common. Event listeners in more classically object-oriented frameworks frequently follow patterns that look like Figure 5. This kind of pattern can also dovetail with issues caused by closures and console logs.

Though there are several ways to address this kind of issue, in this case the easiest change is for the `Leaker` class to be updated to destroy its children objects when it is destroyed. For the example, updating the `destroy` method, as in Listing 13, would be sufficient.

Listing 13. assets/scripts/leaker.js

```
destroy: function(){
    if(this._child !== null){
        this._child.destroy();
    }
    this._registry.remove(this);
}
```

Sometimes a cycle exists between two objects that do not have a strong enough relationship for one of them to assume responsibility for the other object's lifecycle. In such a case, the object that established the relationship between the two objects should assume responsibility for breaking the cycle when it is destroyed.

Conclusion

Even though JavaScript is garbage collected, there are still many ways that you can retain unwanted objects in memory. Most modern browsers have improved at cleaning up memory, but the available tools to evaluate your application's memory heap are still limited—except for Google Chrome. By starting with simple test cases, it's fairly easy to evaluate potential leaking behaviors and determine whether there is leakage.

It is impossible to accurately gauge memory use without testing. It's all too easy to allow circular references to preserve large portions of the object graph. Chrome's Heap Profiler is a valuable tool for diagnosing memory issues; it's a good idea to use it regularly as you develop. Have concrete expectations for when you expect specific resources in the object graph to be freed, and then verify them. Any time that you see a result you don't expect, investigate it.

Planning for the cleanup of an object when you create the object is far easier than trying to graft a cleanup stage into the application later. Always have a plan to remove any event listener, and stop any interval that you create. Be cognizant of the memory use in your application and you will have more reliable and better performing applications.

Downloads

Description	Name	Size
Article source code	JavascriptMemoryManagementSource.zip	4KB

Resources

Learn

- Chrome Developer Tools: Heap Profiling: Take this tutorial to learn how to use the Heap Profiler to uncover memory leaks in your applications.
- "Memory leak patterns in JavaScript" (developerWorks, April 2007): Learn the basics of circular references in JavaScript and why they can cause problems in certain browsers.
- "Finding Memory Leaks": Read how readily leaks can be diagnosed even with no knowledge of the source.
- "JavaScript Memory Leaks": Get more information about causes and detection of memory leaks.
- "JavaScript and the Document Object Model" (developerWorks, July 2002): Read about the JavaScript approach to DOM and the building of a web page to which the user can add notes and edit note content.
- A re-introduction to JavaScript: Take a closer look at JavaScript and its features.
- developerWorks Web development zone: Find articles covering various web-based solutions. See the Web development technical library for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- developerWorks technical events and webcasts: Stay current with technology in these sessions.
- developerWorks on-demand demos: Watch demos ranging from product installation and setup for beginners to advanced functions for experienced developers.
- developerWorks on Twitter: Join today to follow developerWorks tweets.

Get products and technologies

- Developer Channel: Get Google Chrome releases as well as the latest version of Developer Tools.
- IBM product evaluation versions: Download or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2, Lotus, Rational, Tivoli, and WebSphere.

Discuss

- developerWorks community: Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Ben Dolmar

Ben Dolmar has been programming professionally since 2001 and has a long list of technology expertise, including ActionScript, iOS, JavaScript, PHP, Ruby, and graphic design. After graduating from the University of Wisconsin Madison in 1997 with a dual major in journalism and political science, Ben served as director of production at Faith Inkubators from 1998 to 2007. Since joining The Nerdery in 2007, he has helped launch over 400 projects. In 2012, Ben was promoted to principal software engineer. He has spoken on ActionScript at SWFCamp and presented at SocialMediaDev Camp in Chicago on evolving web standards and HTML5.