PEDRO SIMONETTI
pedrosimonetti@gmail.com

A proof of concept pure JavaScript Firebug

2009

Summary

1. Introduction

This document discuss about the viability of making a pure JavaScript port of the Firebug code base, with the goal of creating a almost complete version of Firebug to Internet Explorer 6 and 7.

2. Background

When I discovered Firebug, back in 2006, I was working with web mapping Ajax applications, and I must say that it made my life much easier. In 2008, working with the OpenLayers library, I've notice that the library, when in “development mode”, enabled Firebug Lite automatically. Similar to what Dojo does. Since the development of Firebug Lite was abandoned, the OpenLayers team made some minimal modifications to the code to adapt it with the library.

In march 2008, I released a JavaScript framework called jProton, and I began writing some development tools to help the creation of the framework. A console tool is very handy, and soon the Firebug Lite was there, in the middle of the code base. And there was me hacking around the Firebug Lite code as the folks from OpenLayers, and Dojo.

Months later, after some modifications in the code I thought: “Why not port the Firebug to pure JavaScript?”. Okay, we know this is not possible because the really magic behind Firebug relies on Firefox's core. Even so, I started studying the Firebug code to investigate how much possible it is. Then I realized that the Firebug Lite was basically a very small version the console.js file of the Firebug, and that a lot of Firebug code could theoretically be ported to a cross-browser implementation. The real problem is the lack of documentation, because the Firebug source goes around 35000 lines of code (thanks FWG for the effort with the documentation, specially Honza's tutorials).

Meanwhile, I was very excited to hear about a new version of Firebug Lite, leaded by Azer Koçulu. Finally Firebug Lite got some proper development, and we start to see cool new features coming for non-Firefox browsers. Then I thought: “Wow this is great!”, and start wondering “How far can we get? How many features is possible to implement?”. Soon, I asked myself again: “Why not port the Firebug to pure JavaScript?”.

So, someday in November 2008, I started working in a proof of concept version of a cross-browser library independent pure JavaScript Firebug. My understanding of the Firebug code is very superficial yet, so this is not exactly a “port” of the real code, it's more like a “pseudo-port”, but I've tried to capture the essence of Firebug original code.

The current version is not stable and complete, but I'm communicating earlier about this experiment, as suggested in the meeting notes, so all can share opinion about the achieved result.

3. Guidelines

Here are some guidelines I've learn digging the Firebug code. Many of this guidelines are also recommended by JavaScript performance experts.

3.1. Basic guidelines for a Firebug port version

- Performance in IE6 is critical. IE6 is the slower and buggier browser in the world. If we need hacks and ugly workarounds to achieve a good performance, let's do it. I know, it's a sin, it's ugly, but our soul will be forgiven.
- A port should be extensible

3.2. Performance guidelines

- Use innerHTML instead of DOM methods
- Use HTML and CSS to render all the user interface, using JavaScript only when extremely necessary.
- Use arrays to concatenate long sequences of strings
- Use object literals instead of classes, when you don't need several different instances of an object.
- Lazy load the code, and lazy process the code
- Cache all references to accessed elements
- Avoid using long namespaces, or cache references to it
- Use event delegation
- Avoid extra operations
  - Events
    - If you know that only you will attach an event to an element (for example, the elements of the Firebug User Interface), attach the event direct to the `element`, instead of using `attachEvent` or `addEventListener`, or a general purpose method like `addEvent`. Example:

```
vSplitter.onmousedown = onVSplitterMouseDown;
hSplitter.onmousedown = onHSplitterMouseDown;
```

- If the element is a link (the “a” element) you can use the href attribute to set the click event behavior, using the “javascript:namespace.fn()” pattern. It's ugly, and the browser will show the link in the status bar, but you don't have to worry with memory leak, removing events, and so on.

- If performance is critical, consider using a `onmousedown` instead of `onclick` to gain a some extra miliseconds.

- DOM changes

  - ClassName

    - Try to avoid using multiple classes in DOM elements. If you know an element will have only a class, you can read/write the value directly, instead of using general purpose functions.

    - In the case you need multiple classes, the optimal function to check a class existence is:

```
function hasClass(e, name) {
    return (" "+e.className+" ").indexOf(" "+name+" ") != -1;
}
```

      But in performance critical cases, it may be worth to use the `indexOf` trick directly:

```
if ((" "+e.className+" ").indexOf(" "+name+" ") != -1)
```

    - You could use the same principle to check multiple values like:

```
if ((" meta title script ").indexOf(" "+e.nodeName+" ") != -1)
```

  - CSS

    - The best is to avoid JavaScript writing CSS values

    - When it isn't possible, and you need to change several CSS properties at the same time, consider the possibility of using the `cssText` property. It will override all inline styling, but you can set how many properties you want with a single DOM change.

4. What is implemented

I estimate that only 15-20% of a complete “port” is implemented. The next sections describes which features already are implemented.

The code itself needs to be refactored a little bit, because a lot of things I discovered only in the middle of the process, and because my first goal was getting a working prototype soon to evaluate the viability of the “port”.

4.1. External Codes

Beyond the old `firebug.js` code, all the rest of the source was written from scratch, with the exception of the queued Ajax feature (reimplemented from open source JavaScript framework I wrote), and the following codes:

- John Resig's Sizzle CSS selector engine
- Douglas Crockford's JavaScript parser (not used yet, see the “And if...?” section)
- PPK's cookies (not used yet, to save interface settings, commands list, and other things)

4.2. Code structure

I've tried to follow the same pattern of structure of the code of the original Firebug, including the directory structure.

The only thing I changed is that I've included some sub-directories for organization purposes, but the files could be all located at the same directory, as the original Firebug, if someone think this could cause some confusion.

- `chrome/` – chrome modules
- `object/` – modules related to DOM objects (reps, inspector, html, selector)
- `panel/` – where the panels modules will be located (not used yet)
- `script/` – experimental script parsing and debugging (not used yet)

4.3. Namespaces

4.3.1. Public Namespaces

console

We all know the global `console` namespace.

FBL

In Firebug, `FBL` is a private variable that links to the Lib module. In the “pseudo-port” version, the FBL is being exposed to the global scope for extensibility purposes, so extension developer can use the same syntax to create their extensions.

4.3.2. Internal Namespaces

Firebug

The `Firebug` namespace actually lives in the `FBL` namespace. Inside a module you can refer to any `FBL` property without using `FBL[dot]` notation. You just use the name of the property directly as if it were a local variable/function.

This happens because all modules are wrapped with a declaration like this:

```
FBL.ns(function() { with (FBL) {
```

Important note: It seems that non-FF browsers has problem with the scope changing – the `with(FBL)` trick – in event handlers functions.

FB

The FB namespace is a shortcut of `FBL.ChromeAPI`, the API of the user interface. It is defined only in the chrome context (an iframe or a popup).

4.4. Scope

We all know the global scope, the Firebug introduces some interesting features related to scope handling. These features will be referred as “module scope” and “shared scope” in this document.

4.4.1. Module scope

The basic structure of a module is:

```
FBL.ns(function() { with (FBL) {
// ********************************************************************

// Module code here

// ********************************************************************
}});
```

All variables declared inside the module won't be accessible to other modules, therefore, we can say they are in the “module scope”.

4.4.2. Shared scope

Sometimes you need to share some information (variable/function) with all modules. This is achieved with the "shared scope", powered by the `FBL.ns` trick:

```
FBL.ns(function() { with (FBL) {
// ********************************************************************


// ********************************************************************
// ModuleName API

Firebug.ModuleName =
{
};

// ********************************************************************
// Module Internals

// Shared variables
FBL.sharedVariable = null;
FBL.anotherSharedVariable = null;

// Private variables
var privateVariable = null;
var anotherPrivateVariable = null;

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

function privateFunction()
{
};


// ********************************************************************
}});
```

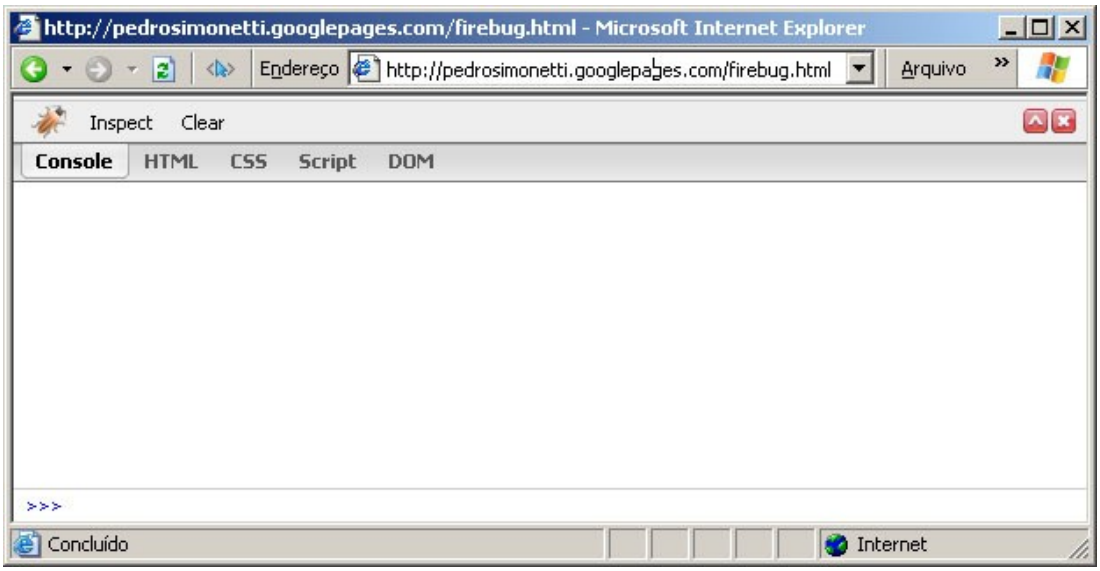
Unfortunately, I truly understood the way the scopes are handled in the Firebug somewhat later in the process, so a lot of variables are in the shared scope, and could be placed in the module scope, a more robust approach.

4.5. User Interface

The interface was redesigned to be a separated HTML and CSS files.

The main part of interface the interface is written in a normal HTML file, loaded in a `iframe`. In the old `firebug.js` version, Joe used an iframe to load the interface. This is an interesting approach because you "pollute" the HTML with a single element, the `iframe`, and all the HTML and CSS of the interface will be only parsed inside the context of the `iframe`. This avoids the interface elements being accidentally captured by some external query. This also avoids conflicts with the CSS, since the rules from the document won't be evaluated inside the `iframe`. Another interest result of this approach is that it makes easier to adapt the code the run in a popup, once the code is ready to capture elements from other context.

As we all know, IE6 sucks. The JavaScript interpreter is very buggy, but I think the CSS errors are even worse: problems with margin, borders, positioning, transparency, and others. In modern desktop computer languages, we have 3 ways of positioning elements: “fixed”, “relative”, and “fit” sizes. It's very common to have a toolbar with fixed height on the top, maybe some control bar on the bottom, but the main part of the screen is positioned with the “fit” method, making it fill all remaining space. In some cases like this, when you have some fixed sized element, and other that should fit in it, the best to do is use the good ol' `tables`, to make only the wireframe of the interface.



The whole interface is a single HTML file

4.6. Source code

The old firebug.js was splitted in the following files: `console.js`, `reps.js`, `commandLine.js`, `frame.js`, and I've added some others as described in the following sections.

4.6.1. `lib.js`

This is the core of the Firebug, a general purpose library that drives the Firebug internal code, that includes namespaces management, event handling, browser checking, and so on. I've included basically a Ajax feature with a queue of requests, and some cookie management functions (from PPK).

The lib module is referred in the source code as the private variable `FBL`.

For extensibility purposes, the `FBL` variable is exposed to the global scope as a variable with the same name, so extension developers can use the `FBL` library the same way each module of Firebug is implemented.

4.6.2. `firebug.js`

This module contains some general definitions used in the rest of the application. This module also implements some functions to manage the loading and initialization of the application and its modules, find the URL location where the current Firebug is running, and cache the document.

4.6.3. `devmode.js`

This module doesn't exists in Firebug. It includes some functions to make the development of the code easier. The main purpose is to dynamically load all modules files, in the proper order according to the dependencies.

While in development, you can easily turn on/off some modules, using the `modules` variable:

```
var modules =
[
    'lib.js',

    'firebug.js',

    'firebug/object/reps.js',
    'firebug/object/selector.js',
    'firebug/console.js',
    'firebug/commandLine.js',

    'firebug/chrome.js',
    'firebug/chrome/frame.js',
    'firebug/chrome/popup.js',
    'firebug/chrome/injected.js',

    'firebug/object/inspector.js',
    'firebug/object/html.js',

    /*
    'firebug/script/tokens.js',
    'firebug/script/parse.js',
    'firebug/script/json2.js',
    'firebug/script/debugger.js',/**/

    'firebug/boot.js'
];
```

As you can see above, the ultra-experimental debugger module is commented, and therefore, it won't be loaded.

This module also implements a build function, and HTML and CSS compression functions (used in the injected interface when in bookmarlet mode).

4.6.4. `console.js`

This file is basically the old `firebug.js`.

4.6.5. `commandLine.js`

This module implements all main features of the original Firebug. It allows searching the commands history with the UP and DOWN keys, auto-completion with TAB and Shift+TAB (reverse), and shortcuts of some functions:

- `$()` – a shortcut for `document.getElementById()`
- `$$()` – a shortcut for `getElementsBySelector()` - powered by Sizzle
- `dir()` – a shortcut for `console.dir()`
- `dirxml()` – a shortcut for `console.dirxml()`

As in the original implementation, these shortcuts won't be allowed if some global function/variable is defined with that same name, preventing conflicts with JavaScript frameworks.

4.6.6. `chrome.js`

This is the main part of the Firebug “pseudo-port” version. This module is responsible to manage all user interface aspects of the code, from rendering to event handling/delegation.

Most of my efforts were concentrated in this module. It isn't stable yet, and the rendering and event aspects should be entirely reimplemented, but in general, the current implementation of this module already is considerably robust.

4.6.7. `chrome/frame.js`

This is the chrome module for handling the interface as an iframe

4.6.8. `chrome/popup.js`

This is the chrome module for handling the interface as a popup

4.6.9. `chrome/injected.js`

This is the chrome module for handling injected interfaces (frame and popup), necessary when in bookmarlet mode to prevent cross-domain problems.

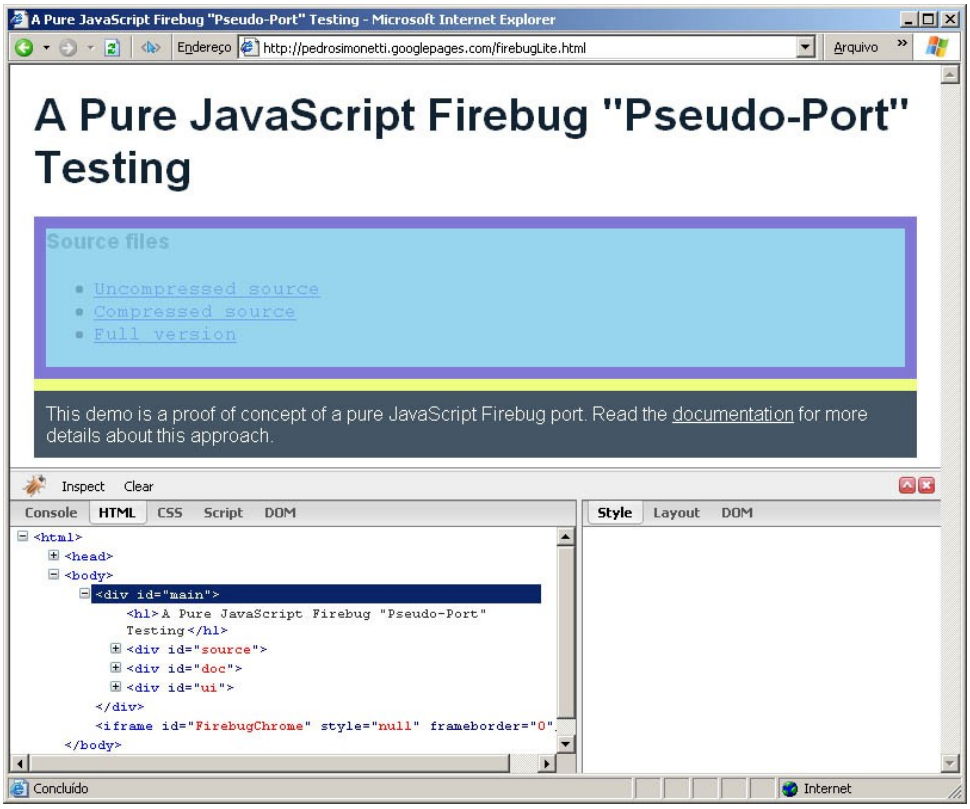
4.6.10. `object/reps.js`

This is basically a piece of the old `firebug.js`, responsible for generating the representations of basic types of objects.

4.6.11. `object/inspector.js`

This is other module I gave special attention. It includes several methods for converting units (pt, px, ex, em, and percentage), and other methods for getting element measurements, and also some hacky function to find auto margin value in pixels (kinda buggy yet).

I really like the Box Model inspector in Firebug, so I tried to recreate it with accuracy. The percentage conversion function is not yet implemented, and some relative elements is not properly handled yet (form elements), but the primary results are quite good.



Box model inspection working on IE6

4.6.12. `object/html.js`

This module is responsible for generating the HTML tree in the HTML tab. It's very similar to the `dirxml()` function, but it is expandable via tree controls (the "+" icon).

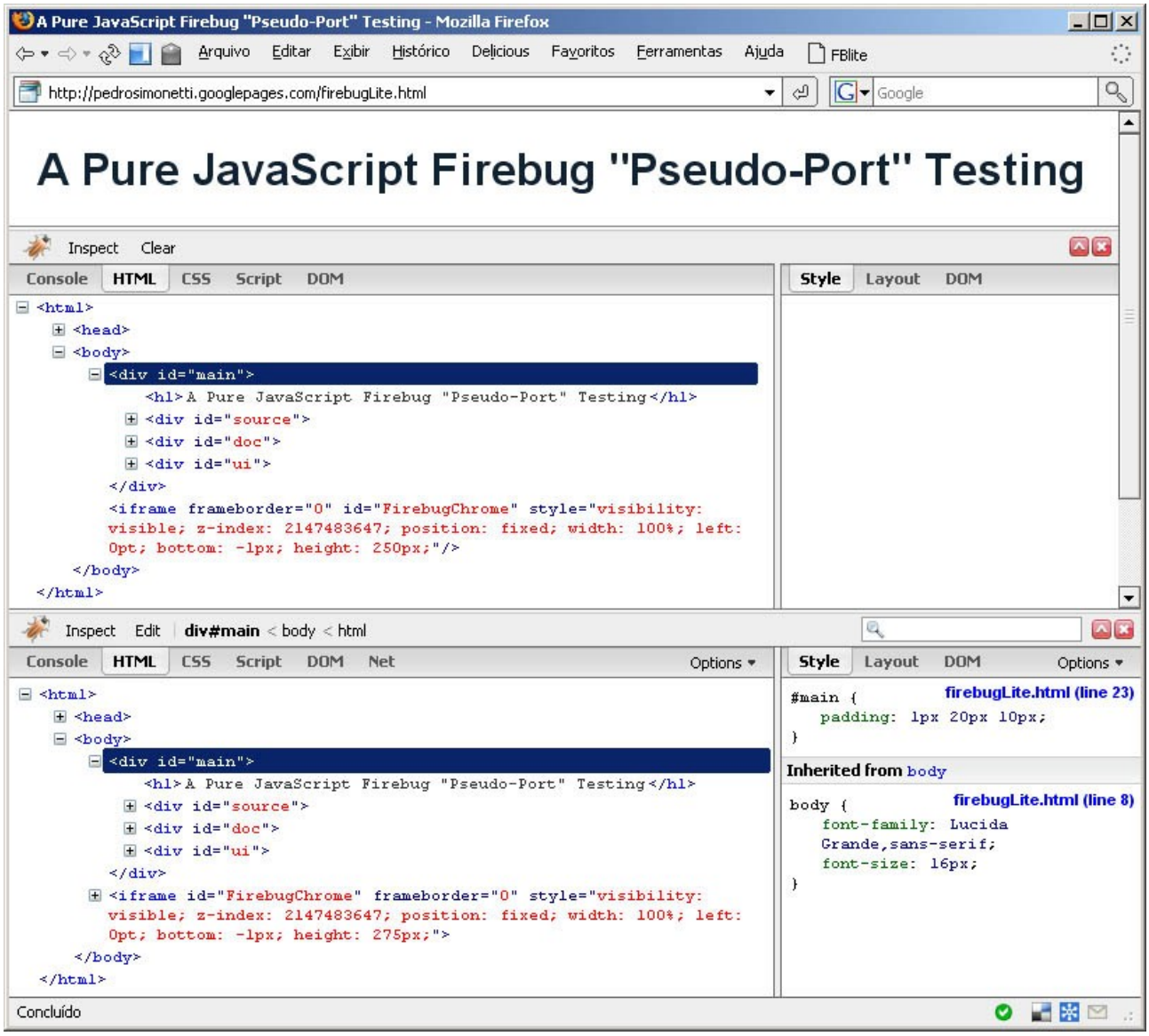
To make it look pretty in the screen, we must use a lot of DOM elements (a lot of `<div>` and `<span>`), so in large documents this could be a problem, not only to parse the DOM tree entirely (like `dirxml` does), but also to simply render it on the screen. When you reach the number of a couple of thousands of DOM elements, IE6 will start to run very slowly.

After a lot of experiments, I realize that the best solution is to generate only the DOM nodes you need at the time. Only when you press the "+" button, the child nodes of the related element will be parsed and generated (but not their children). This saves a lot of processing, make things faster. But we still have the problem in IE6 with large number of elements, so when you press the button again (now with the "-" icon), the elements will be actually removed from the DOM tree.

5. Demo

The proof of concept demo can be found here:
http://pedrosimonetti.googlepages.com/firebugLite.html

If you're gonna look the code, I suggest downloading the full zipped version (that contains all modules in separated .js files):
http://pedrosimonetti.googlepages.com/firebug.zip



The original Firebug and the "pseudo-port" running together

6. Next steps

Some thoughts about what could be next.

6.1. What else could be done?

Virtually, almost every feature of Firebug could be ported to a pure JavaScript implementation. The problem is that some of them would require so much effort that it wouldn't worth.

6.1.1. Domplate + Reps

Firebug source code is a gold of mine of ideas. The `lib.js` has some very nice ideas, but the `domplate.js` is event nicer!

Christoph Dorn is working on a cross-browser version of Domplate. With a cross-browser Domplate, we could port the `reps.js` too without much problems. The `reps.js` is another ultra-cool part of the Firebug source, responsible to generate the various kinds of representation each “thing” has in Firebug. Porting the `reps.js` file would allow the same variety in a cross-browser version.

6.1.2. CSS

You can only read the inline style or the computed style in other browsers than FF. So, it's not directly possible to show the inheritance of styles like Firebug does.

This kind of feature is so useful that may be worth simulating it.

1. Find all CSS rules
2. For each rule:
    1. Use sizzle to get the related elements
    2. Signalize the related cache representation of the element that a rule has been matched for the element, informing what rule is, it's CSS selector, the file and the line number where the rule is.

6.1.3. Layout

The layout tab could be added without much problems, because most of the measurements functions you need already is implemented in the Inspector module.

6.1.4. Live Editing

A live editing feature with keyboard handling is also possible with some reasonable effort. Maybe digging in the `editor.js` module in Firebug we can find some basics functions to be ported.

6.2. And if... ?

Some findings and brainstorming.

6.2.1. Script Debugging

The ugly way: The alert + popup trick

The `alert()` is probably one of the most popular JavaScript functions. Before Firebug, everyone used to call `alert` to inspect things in the code. The real problem of alert is that it halts all JavaScript processing. Nothing in the events stack will be executed until the `alert` box is closed, even the mouse events. This could lead to problems in asynchronous scenarios.

In the other hand, we can use this behavior to do a simple kind of debugging.

```
Firebug.Script.debugFn(object, "functionName");
```

Then we'll have a handler function similar to this:

```
function debugHandler()
{
    forcePopup();
    alert("Press ok to stop debugging.");
    originalFunction.apply(this, arguments);
};
```

Now, each time `object.functionName()` is called, the execution of the application will be halted, the debug interface will be opened in a popup, and then you will be able to inspect the context of that function call in the DOM, HTML and CSS tabs. It's very primitive but it may be handy in some situations.

The hard way: a JavaScript Function call debugger

Douglas Crockford has made a very cool JavaScript compiler written in JavaScript (the used in the JSLint tool). The parsing process is very CPU intensive, so would be hard to parse the whole JavaScript code. But would be possible to parse only a function call, allowing the “step over”, “step into”, “step out” and watch debugging features. It's hard, but it's possible.

6.2.2. Net Profiling

The ugly way: Rewrite functions source code with RegExp

The hard way: a JavaScript parser profiler

7. Conclusions

It's possible to make a “pseudo-port” of Firebug to a pure JavaScript version. There are some great benefits using this approach, but there are also big challenges.

7.1. Benefits

- Having Firebug and Firebug Lite the same basic API and code principles could benefit the development of the Lite version in some significant ways.
    - It makes easier for developers involved with Firebug to contribute with the Lite code base
    - Would be possible to port some some bugs and/or features from the original code base to the Lite one (for example, the Domplate and Reps modules can be ported to pure JavaScript).
    - Some Firebug extensions could, in theory, be ported to the Lite version as well (for example, Resig's Fireunit extension?)
    - Some documentation of the Full version could also be ported to the Lite one.
- By using the same algorithm as Firebug in the core features, and using the same CSS for styling, is possible to recreate the same look-and-feel of the original Firebug.
- The modular pattern of the Firebug is a benefit itself once each module encapsulates a specific subdomain of the application, making easier to track bugs, and easier to divide the work between developers.

7.2. Challenges

- The uncompressed code base could be very large, making the first load heavy. I estimate that a full port could reach about 500-600k of JavaScript (15000-20000 lines of code). This could be minimized by lazy loading some modules, but the Firebug modules are so interconnected that only a minimal code wouldn't be required in the first load. The best option would be compressing and zipping.
- The increased size and complexity makes the application even hard to code and manage.
- The large size makes necessary a good unit test coverage.
- Considering the differences of the Firebug code and the new features implemented by Azer, it may be difficult to refactor his code to this pattern.
- IE6 is slow, buggy, ugly. This requires using a lot of hacks, ugly techniques, and non-standards implementations.
- The performance and memory leak are critical specially when debugging large documents.