**Improve Loading Time of Web Pages**

<http://sixrevisions.com/web-development/10-ways-to-improve-your-web-page-performance/>

<http://developer.yahoo.com/blogs/ydn/posts/2007/07/high_performanc_5/>

<http://devmoose.com/coding/10-ways-to-instantly-speed-up-your-website>

<http://www.abhishekbharadwaj.com/2010/12/speed-up-your-website-avoid-duplicate-scripts/>

**Improve Javascript Performance**

In todays world websites make extensive use of javascript compared to the website created a number of years back. Javascript has not only helped in validations but also to make the UI more flashy and attractive. The extensive use of javascript would require the browser engine to execute the scripts meaning more load on the browsers which could lead to slower response times. A number of changes can be done to improve the performance of javascript. Following, are some tips that can be used to better the UI experience using javascript.

**Avoid using eval**

When using the eval function the script engine has to convert the string into executable code and then execute it. This would result in a performance overhead, leading to slower execution time.

Consider the code snippet

function testEval(index) {

var result;

eval('result = test[' + index + ']');

return result;

}

The above code can be replace with

function testWOEval(index) {

var result;

result = test[index];

return result;

}

**Avoid using try – catch block**

The try catch block is not like other statements. Each time the catch block is executed a new variable is created, basically the exception caught and assigned to a variable. Once the catch block is executed the variable is destroyed. This behavior is not handled efficiently in some browsers and causes a performance issue.

Example:

var oProperties = ['first','second','third',...,'nth'], i;

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

try {

test[oProperties[i]].someproperty = somevalue;

} catch(e) {

...

}

}

This could be replaced by the following code where we check if the property exists before assigning any value.

var oProperties = ['first','second','third',...,'nth'], i;

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

**if**( test[oProperties[i]] ) {

test[oProperties[i]].someproperty = somevalue;

}

}

**Implicit Literal Conversion**

Literals such as strings, numbers and booleans have two representations in ECMAScript. Each of these literals can be created as a value or an object. If you consider the string literal, a string value is created as var vString = ‘abcd’. The equivalent object representation would be var oString = new String (‘abcd’). Similarly, for number it is var num = new Number(value); and boolean as var myBoolean=new Boolean();.

The problem lies in the literal value. Whenever you access a literal value the ECMAScript engine would create an object representation of the literal and then perform the action.

For example,

var **s** = '0123456789';

**for**( var i = 0; i < **s**.**length**; i++ ) {

**s**.charAt(i);

}

In the above code example each time s.length is executed it is converted to an object and then the length is calculated. Also, the similar thing happens when s.charAt(i) is invoked. Totally, 21 string objects are created and destroyed to run the above code. This can be replaced with

var **s** = new String('0123456789');

**for**( var i = 0; i < **s**.**length**; i++ ) {

**s**.charAt(i);

}

Here, only one object gets created, thus an improve in performance should be expected.

One needs to keep in mind that when code calls functions of literals it is better to make them as objects before using them.

**Avoid for-in in performance-critical functions**

The for-in loop is was introduced to make things simple but is often misused. At times a normal for loop would be more appropriate than the for-in loop. Use of for-in loop requires the script engine to build an enumeration before it can iterate through it.

The following would be an incorrect use of the for-in loop

**var** oSum = 0;

**for**( **var** i **in** oArray ) {

oSum += oArray[i];

}

The above could be replaced with the normal for loop to be more efficient

**var** oSum = 0;

**var** oLength = oArray.length;

**for**( **var** i = 0; i < oLength; i++ ) {

oSum += oArray[i];

}

A proper use of a for-in loop would be to iterate through an list of properties in an object like shown below.

var o = {'name':'Batman', 'age':33, 'city':'Gotham City'};

for (var p in o) {

alert(p+': '+o[p]);

}

Here we are iterating through each object in the JSON, we get the key and access the value.

**String Concatentation**

String concatenation can be an expensive operation. When using the ‘+’ operator a temporary variable is created in memory to store the result before it is assigned.

For example, consider the line a += 'x' + 'y'; Here a temporary variable is created in memory and assigned the value ‘xy’. This is then concatenated with the value of variable a and reassigned to a.

We can avoid the temporary variable being created by splitting up the above line into two.

a += 'x';

a += 'y';

Here ‘x’ is appended to variable a and then ‘y’. The temporary variable to store ‘xy’ is not created here.

**Avoid inspecting large numbers of nodes**

When trying to locate a node, try to limit your search. Searching across all elements can be a performance hit.

**var** allElements = document.getElementsByTagName('\*');

**for**( **var** i = 0; i < allElements.length; i++ ) {

**if**( allElements[i].hasAttribute('someattr') ) {

...

}

}

The above code gets all elements and tries to search if elements have the attribute ‘someattr’. Rather than using the above code, if you know the id of the element or the parent node/element that contains the node you are searching for, you can use that information to retrieve the node faster. Say for example, that the unknown element is known to be inside a div with the id inhere, this code could perform far better:

**var** allElements = document.getElementById('inhere').getElementsByTagName('\*');

**for**( **var** i = 0; i < allElements.length; i++ ) {

**if**( allElements[i].hasAttribute('someattr') ) {

...

**break**;

}

**}**

**Use var to define variables**

When variables are being used, they are searched based on a scope chain starting from the most to least specific scope. In the worst case scenario if the variable does not yet exist, it is first searched in all scopes in the chain and then created. To avoid this if the variable is declared using var the engine would know that it is being created at that point and avoid searching in further scopes.

function testLocalVar() {

local\_status = "Let’s see if this works."

status = "It works!!"

}

In the above code you will see a variable local\_status defined within the function scrope being assigned a value. Since local\_status is not declared using var it will search for the declaration in all scopes. If the variable is not found, a global variable will be created. Since the variable is created at the global level, all other functions will now be able to access it.

It is highly recommended to define your variables with var, especially within the function scope. This would prevent look up operation by the script engine and execute your scripts much faster.

Also, consider the variable status. It was intended to be used within the function (local scope) but since it was not defined with var, the engine starts searching through the scopes. Since it was not found you would expect it to create a global variable, but DOM has a property status (window.status) and the value gets assigned to this. One needs to be careful when defining variables without var.

### Cache DOM Elements in script variables

### Every time you access a property, variable or method a look-up is performed. There are certain cases where DOM values can be cached. Consider the following code:

document.getElementById('test').property1 = 'value1';

document.getElementById('test').property2 = 'value2';

document.getElementById('test').property3 = 'value3';

document.getElementById('test').property4 = 'value4';

Here four requests are being made to the same object and properties being assigned. Rather than writing the above code you could store a reference to the object and assign all the properties, reducing the number of requests made to access the node.

**var** sample = document.getElementById('test');

sample.property1 = 'value1';

sample.property2 = 'value2';

sample.property3 = 'value3';

sample.property4 = 'value4';

Consider the next example

function BuildUI()  
{  
      var baseElement = document.getElementById(‘target’);  
      baseElement.innerHTML = ‘’;   
      baseElement.innerHTML += BuildTitle();  
      baseElement.innerHTML += BuildBody();  
      baseElement.innerHTML += BuildFooter();  
}

Here innerHTML is looked up several times to set the value. Rather than this the entire UI could have been constructed and assigned to innerHTML in a single assignment.

function BuildUI()  
{  
      var elementText = BuildTitle() + BuildBody() + BuildFooter();  
      document.getElementById(‘target’).innerHTML = elementText;  
}

**Cache Functions**

Everything in javascript is a lookup, even when a function is called. If you have a function being called multiple times it would make sense to cache it locally and then use it.

function IterateWorkOverCollection()  
{  
      var length = myCollection.getItemCount();  
   
      for(var index = 0; index<length; index++)  
      {  
            Work(myCollection[index]);  
      }  
}

You will see that function Work is called repeatedly based on items in the collection. Each time Work is called a look-up is done and then invoked. If we cache the function locally it would not need to search far and would improve performance. The above code can now be changed to

function IterateWorkOverCollection()  
{  
      var funcWork = Work;  
      var length = myCollection.getItemCount();  
   
      for(var index = 0; index<length; index++)  
      {  
            funcWork(myCollection[index]);  
      }  
}

Here, notice Work is stored ad funcWork. So when this is called it would need to search locally and not in all scopes.

**JQuery Specific**

**User For instead of Each**

Native javascript functions are always faster than any helper counterparts. If your looping through an array you can use javascript for loop or the jQuery each function. The each function will always be slower. Some cases if your returning a JSON, you could probably convert that to an array and then loop through it. If you try out the below code snippet you would see the difference. (jQuery-forVsEach.html for more details)

var array = new Array ();

for (var i=0; i<10000; i++) {

array[i] = 0;

}

console.time('native');

var l = array.length;

for (var i=0;i<l; i++) {

array[i] = i;

}

console.timeEnd('native');

console.time('jquery');

$.each (array, function (i) {

array[i] = i;

});

console.timeEnd('jquery');

**Use IDs Instead of Classes**

Using jQuery’s class selector to find an element is very flexible and handy but is much slower in performance compared to using ids. JQuery, internally uses the native getElementById to get an element when using Id, where as the class selector would have to traverse the entire DOM in search of the elements.

// Example creating a list and filling it with items

// and selecting each item once

console.time('class');

var list = $('#list');

var items = '<ul>';

for (i=0; i<1000; i++) {

items += '<li class="item' + i + '">item</li>';

}

items += '</ul>';

list.html (items);

for (i=0; i<1000; i++) {

var s = $('.item' + i);

}

console.timeEnd('class');

console.time('id');

var list = $('#list');

var items = '<ul>';

for (i=0; i<1000; i++) {

items += '<li id="item' + i + '">item</li>';

}

items += '</ul>';

list.html (items);

for (i=0; i<1000; i++) {

var s = $('#item' + i);

}

console.timeEnd('id');

If you want to use the class selector, try limiting the search to a specific element. If you are aware of the element that contains the node having the class you’re searching, you can specify that node as a context in jQuery.

You can use $(expression, context) to limit search to within a specific element. So instead of using $('.class').css ('color' '#123456'); go for

$('.class', '#class-container').css ('color', '#123456');

The second line of code will be much faster, because jQuery need not traverse the entire DOM, just the ‘#class-container’ element.

**Cache**

As with normal javascript, caching elements would be much faster than trying to access them again and again. Never select elements multiple times inside a loop EVER! It’d be a speed-killer!

$('#item').css ('color', '#123456');

$('#item').html ('hello');

$('#item').css ('background-color', '#ffffff');

// you could use this instead

$('#item').css ('color', '#123456').html ('hello').css ('background-color', '#ffffff');

// or even

var item = $('#item');

item.css ('color', '#123456');

item.html ('hello');

item.css ('background-color', '#ffffff');

// as for loops, this is a big no-no

console.time('no cache');

for (var i=0; i<1000; i++) {

$('#list').append (i);

}

console.timeEnd('no cache');

// much better this way

console.time('cache');

var item = $('#list');

for (var i=0; i<1000; i++) {

item.append (i);

}

console.timeEnd('cache');