JavaScript Performance Best Practices

This article gives valuable guidance for building faster web applications. The article is a collection of guidelines from various sources.

10 Jan 2010

Case study: Tips and tricks for improving JavaScript performance

Tips & tricks in this article were collected from the developer experiences when creating an example application.

Overview

- Source: Efficient JavaScript - ECMAScript

First understand the big picture and the major component of the stack affecting the performance. It is of utmost importance to understand what can and cannot be optimized in JavaScript without touching the browser codebase. A good starting point for this study is to first take a look at the JavaScript Performance Stack (depicted in the figure).

Pick your battles. It is generally a good approach to first optimize those parts which give you the biggest improvements.

There are many interrelated components that play an instrumental role in the real-life web application performance such as those responsible for layout, rendering, parsing HTML, marshaling, DOM, CSS formatting, JavaScript -- as you see, JavaScript is only one part of the performance equation.

The most expensive operations tend to be reflowing the layout and repainting. Although you as a JavaScript developer cannot optimize browser layout or painting algorithms you can still implicitly affect the performance of these expensive operations by trying to avoid triggering there expensive operations unnecessarily. A real-life example of IE8 tells us that layout and rendering tasks takes most time on IE8 (see webcast at -20:00 mins)

Below are some examples of common reasons for slow JavaScript performance that you as a JavaScript developer can easily fix and make your web application to perform better instantly:

DOM access

Interaction with the DOM is usually slower than normal JavaScript code. Interaction with the DOM is usually inevitable, but try to minimize it. For instance, dynamically creating HTML with strings and setting the **innerHTML** is usually faster than creating HTML with DOM methods.

eval

Whenever possible, avoid the eval method because significant overhead is involved in script evaluation.

with

Using with statements creates additional scope objects that slow variable access and create ambiguities.

for-in loops

Traverse arrays use the traditional `{{{1}}}`instead of for-in loops. Unfortunately, most JavaScript environments have a slow implementation of for-in loops.

Core JavaScript pitfalls

Avoid using eval or the Function constructor

- Using eval or Function constructor are expensive operations as each time they are called script engine must convert source code to executable code.
- Additionally, using eval the context of the string has to be interpreted at runtime.

Slow:

```
function addMethod(object, property, code) {
    object[property] = new Function(code);
}
addMethod(myObj, 'methodName', 'this.localVar=foo');
```

Faster:

```
function addMethod(object, property, func) {
    object[property] = func;
}
addMethod(myObj, 'methodName', function () { 'this.localVar=foo'; });
```

Avoid using with

Although seen as a convenience, with construct introduces an extra scope to search each time variable is referenced and the contents of that scope is not known at compile time.

(C) Copyright Nokia 2013. All rights reserved.

Slow:

```
with (test.object) {
    foo = 'Value of foo property of object';
    bar = 'Value of bar property of object';
}
```

Faster:

```
var myObj = test.object;
myObj.foo = 'Value of foo property of object';
myObj.bar = 'Value of bar property of object';
```

Don't use try-catch-finally inside performance-critical functions

- The try-catch-finally construct creates a new variable in the current scope at runtime each time the catch clause is executed where the caught exception object is assigned to a variable.
- Exception handling should be done at as high level in the script where it does not occur frequently, for example outside a loop.
- Or if possible, avoid try-catch-finally completely

```
Slow:
var object = ['foo', 'bar'], i;
for (i = 0; i < object.length; i++) {
   try {
      // do something that throws an exception
   } catch (e) {
      // handle exception
   }
}
```

Faster:

```
var object = ['foo', 'bar'], i;
try {
    for (i = 0; i < object.length; i++) {
        // do something
    }
} catch (e) {
    // handle exception
}
```

Avoid using global variables

- If you reference global variables from within function or another scope, scripting engine has to look through the scope to find them.
- Variable in the global scope persist though the life time of the script, whereas in local scope they are destroyed when the local scope is lost.

Slow:

```
var i,
    str = '';
function globalScope() {
    for (i=0; i < 100; i++) {
        str += i; // here we reference i and str in global scope which is slow
    }
}
globalScope();
```

Faster:

```
function localScope() {
    var i,
        str = '';
    for (i=0; i < 100; i++) {
        str += i; // i and str in local scope which is faster
    }
}
localScope();
```

Avoid for-in in performance-critical functions

- The for-in loop requires the script engine to build a list of all the enumerable properties and check for duplicates prior the start.
- If your code inside for loop does not modify the array it iterates pre-compute the length of the array into a variable **len** inside for loop scope.

Slow:

```
var sum = 0;
for (var i in arr) {
   sum += arr[i];
}
```

Faster:

```
var sum = 0;
for (var i = 0, len = arr.length; i < len; i++) {
   sum += arr[i];
}
```


(C) Copyright Nokia 2013. All rights reserved.

Use strings accumulator-style

- Using + operator a new string is created in memory and the concatenated value is assigned to it. Only after this the result is assigned to a variable.
- To avoid the intermediate variable for concatenation result, you can directly assign the result using += operator.

Slow:

```
a += 'x' + 'y';
```

Faster:

```
a += 'x';
a += 'y';
```

Primitive operations can be faster than function calls

- Consider using alternative primitive operation over function calls in performance critical loops and functions.

Slow:

```
var min = Math.min(a, b);
arr.push(val);
```

Faster:

```
var min = a < b ? a : b;
arr[arr.length] = val;
```

Pass functions, not strings, to setTimeout() and setInterval()

- If you pass a string into setTimeout() or setInterval() the string will be evaluated the same way as with eval which is slow.
- Wrap your code into an anonymous function instead so that it can be interpreted and optimized during compilation.

Slow:

```
setInterval('doSomethingPeriodically()', 1000);
setTimeOut('doSomethingAfterFiveSeconds()', 5000);
```

Faster:

```
setInterval(doSomethingPeriodically, 1000);
setTimeOut(doSomethingAfterFiveSeconds, 5000);
```

Avoid unnecessary DOM references in objects

- Dont do this :

```
var car = new Object();
car.color = "red";
car.type = "sedan"
```

- Better way would be:

```
var car = {
 color : "red";
 type : "sedan"
}
```

Maximize object resolution speed and minimize scope chain

- Inefficient way:

```
var url = location.href;
```

- Efficient one:

```
var url = window.location.href;
```

Try to keep script comments to a minimum/ Avoid long variable names

(C) Copyright Nokia 2013. All rights reserved.

Keep script comments to a minimun or avoid them altogether, especially inside functions, loops and arrays. Comments unnecessarily slow down script execution and increase file size. For example,

- Bad Idea :

```
function someFunction()
{
var person_full_name="somename"; /* stores the full name*/

}
```

- Better way:

```
function someFunction()
{
var name="somename";
}
```

Store local references to out-of-scope variables

- When a function is executed an execution context is created and an activation object containing all local variables is pushed to the front of the context's scope chain.
- Further in the chain, the slower the identifier resolution is, which means local variables are fastest.
- By storing local references to frequently used out-of-scope variables reading and writing to variables is significantly faster. This is visible especially with global variables and other deep searches for identifier resolution.
- Also in-scope variables (`var myVar`) are faster than object property access (`this.myVar`).

Slow:

```
function doSomething(text) {
    var divs = document.getElementsByTagName('div'),
        text = ['foo', /* ... n ... */, 'bar'];
    for (var i = 0, l = divs.length; i < l; i++) {
        divs[i].innerHTML = text[i];
    }
}
```

Faster:

```
function doSomethingFaster(text) {
    var doc = document,
        divs = doc.getElementsByTagName('div'),
        text = ['foo', /* ... n ... */, 'bar'];
    for (var i = 0, l = divs.length; i < l; i++) {
        divs[i].innerHTML = text[i];
    }
}
```

If you need to access an element (e.g. the head) inside a big loop using a localized DOM access (get in the example) is faster.

Faster:

```
function doSomethingElseFaster() {
    var get = document.getElementsByTagName;
    for (var i = 0, i < 100000; i++) {
       get('head');
    }
}
```

Caching values to variables

- Caching values to local variables where ever needed prevents interpreter from doing the repetative job.
- Couple of examples below should clarify the caching/storing values to variable in broader sense.

- Example 1) Caching math functions in variables before executing calculations within a loop

Wrong Way:

```
var d=35;
for (var i=0; i<1000; i++) {
  y += Math.sin(d)*10;
}
```

Better Approach:

```
var d = 55;
var math_sind = Math.sin(d)*10;
for (var i=0; i<1000; i++) {
  y += math_sind;
}
```

- Example 2) Caching/Storing array length when used in loops

Bad Approach:

(C) Copyright Nokia 2013. All rights reserved.

The length of the array arr is recalculated every time the loop iterates.

```
for (var i = 0; i < arr.length; i++) {
    // do something
}
```

Better Approach:

Better way is to to cache the length of the array:

```
for (var i = 0, len = arr.length; i < len; i++) {
    // do something
}
```

- In General we should avoid sending the interpreter out to do unnecessary work once it has already done it once, eg: figuring out the scope chain or the function evaulation values of an expression used more than once.**storing/caching values to variable only makes sense if those values are used repetatively or more than once, otherwise we are creating overhead again for declaring a variable, assigning values and then just using only once, so keep in mind**

Sources: Variable Performance, slides High Performance Kick Ass Web Apps Video at JSCONF 2009

Tips for better loading performance

Load scripts without blocking for faster startup and to show a splash screen

- When <script> tags are found in the HTML document the referenced script resources are downloaded and executed before the rendering engine can continue to download other resources which effectively blocks the rendering of the page below the tag. To avoid this blocking behaviour a script tag can be created via a mechanism known as a dynamic script tag injection.
- This mechanism allows the rendering engine to immediately render and display the initial view (aka the splash screen) defined in HTML while the JavaScript resources are still being loaded and executed which leads to better user experience.

Slow:

```
<div id="splash"/>
<script src="my-script-file.js" type="text/javascript"></script>
```

Faster:

```
<div id="splash"/>
```

```
// JavaScript
function loadScript(src, callback) {
    var head = document.getElementsByTagName('head')[0],
        script = document.createElement('script');
    done = false;
    script.setAttribute('src', src);
    script.setAttribute('type', 'text/javascript');
    script.setAttribute('charset', 'utf-8');
    script.onload = script.onreadstatechange = function() {
        if (!done && (!this.readyState || this.readyState == 'loaded' || this.readyState == 'complete')) {
            done = true;
            script.onload = script.onreadystatechange = null;
                if (callback) {
                    callback();
                }
            }
    }
    head.insertBefore(script, head.firstChild);
}

// load the my-script-file.js and display an alert dialog once the script has been loaded
loadScript('my-script-file.js', function() { alert('my-script-file.js loaded.'); });
```

Sources:

- The best way to load external JavaScript
- Dojo require

Add an Expires or a Cache-Control HTTP header

Using Apache the Expires HTTP header and the max-age directive of the Cache-Control HTTP header in server responses can be configures in .htaccess. The syntax is as follows:

`ExpiresDefault "<base> [plus] {<num> <type>}*"` `ExpiresByType type/encoding "<base> [plus] {<num> <type>}*"`

Example: `ExpiresActive On ExpiresByType image/png "access plus 1 year"`

Source: Apache mod_expires

Gzip JavaScript and CSS resources

Below is a simple configuration to gzip not only JavaScript and CSS but also HTML, XML and JSON. To accomplish this, the following must be set in the

(C) Copyright Nokia 2013. All rights reserved.

Apache .htaccess:

```
AddOutputFilterByType DEFLATE text/html text/css text/plain text/xml application/x-javascript application/json
```

Source:

- Apache mod_deflate

Use YUI Compressor or JSMin to compress the code

- The Best: YUI Compressor provides the best overall performance when you consider it as: Total_Speed = Time_to_Download + Time_to_Evaluate. Depends on Rhino and is not applicable for real-time compression.
- Simple: JSMin has implementations in nearly all the languages and is applicable for real-time compression. After gzipping the size comes very close to that of YUI Compressor.
- JSMin or YUI Compressor + gzipping is better performer than Dean Edwards' Packer + gzipping. Although Packer provides the smallest (byte-size) code it will evaluate much slower as the scripts are unpacked on the client-side using one pass of a RegExp? this introduces an overhead especially significant on slow mobile devices.

Sources

- Library Loading Speed
- Gzip Your Minified JavaScript Files

Minimize the number and size of resources

- Concatenate multiple scripts into one. However, consider that some mobile browsers have limits in how big resources they keep in cache.
- Always aim for the smallest code size prior to minification and re-factor to increase re-usable code, consider all resources: HTML, JavaScript, CSS, images, JSON loaded using XHR.
- Minimize the number of resources per host to enable more efficient parallel loading of resources
- Serve your content from different hosts to overcode HTTP/1.1 limitation of parallel loading of two resources per host.

Load scripts without blocking parallel downloads

- Multiple scripts can be loaded in parallel without blocking using techniques such as via normal script src, XHR eval, XHR injection, script in iframe, script DOM element, script defer and document.write script tag.
- Depending on your specific constraints (resources in the same/different domain, need to preserve script loading order, need to show browser loading indicator) you can do an informed decision by checking the decision tree in the following presentation (slide 26).

Even Faster Websites - Steve Souders at SXSW '09

Couple asynchronous scripts

- If you have inline script which depend on the script you have multiple options: hardcoded callback, `window.onload`, timer, degrading script tags and script onload.
- See slide 35 onwards for details:

Even Faster Websites- Steve Souders at SXSW '09

Move inline scripts above stylesheets

- Browsers download stylesheets in parallel with other resources that follow unless the stylesheet is followed by an inline script.
- Workaround is to either avoid inline scripts or move them above stylesheets or below other resources (e.g. images, scripts) and use <link>, not @import.
- Live demo

Use iframes sparingly

- Parent's onload doesn't fire until iframe and all its components are downloaded.
- Workaround for Safari and Chrome is to set iframe src in JavaScript.
- Scripts and stylesheets also block iframe from loading.
  - In IE and FF stylesheets in the parent block the iframe or its resources
- iframe shares connection pool with parent, typically 2 connections per host.

Document Object Model (DOM) obscurities

Source: Efficient JavaScript - DOM

Slow DOM performance can be traced back into the following three main causes:

Extensive DOM manipulation	Extensive use of DOM API is a well-known cause of slowness.
Script triggers too many reflows or repaints	As a consequence of DOM manipulation, reflowing the layout and repainting are very expensive.
Slow approach to locating nodes in the DOM	Locating a desired node(s) in the DOM is potential bottleneck if the DOM is sizable and/or complex.

Minimize the size of the DOM

- The size of DOM slows down all the operation related to it such as reflowing, traversal and DOM manipulation.

(C) Copyright Nokia 2013. All rights reserved.

- The most effective way to make programs faster is to make n smaller means that the DOM should be as small as possible at all times.
- Minimize n, you can track the number of elements in a page by:
  - {{{1}}}
  - {{{1}}}

Use document fragment templates for re-usability

- Dynamically inserting and updating elements into the DOM is expensive. An efficient way to tackle this is to use HTML templates which can be cloned and re-used for re-usable parts of the DOM such as dialogs and other UI widgets.
- In practice the approach is to modify, clone and append all nodes in JavaScript without touching the live DOM and append the completed document fragment to the DOM at once. One can do this with DOM API or alternatively construct a string representation of the HTML fragment to append based on a string template and push that into the DOM with a one innerHTML assignment. On both cases the rendering engine does not have to reflow and repaint the layout multiple times. Next we introduce some techniques to achieve this behavior.

Minimize the number of reflows and repaints

Repaint Repainting happens when something is made visible or hidden without altering the layout of the document. For example if an outline is added to an element, its background color is changed of its visibility is changed. Repainting is an expensive operation (paint events demo).

Reflow Reflow happens whenever the DOM is manipulated in a way it affects the layout. For example, style is changed to affect the layout, className property is changed or browser window size is changed. Once an element needs to be reflown, its children will also be reflown and any elements appearing after the element in the DOM. Finally, everything is repainted. Reflows are even more expensive operations, than repainting. In many cases reflowing is comparable to layout out the entire page again (reflow demo videos).

- Operation that trigger reflows should be used sparsely.
- Reflowing a table element is more expensive that reflowing equivalent element with block display.
- Elements that are positioned absolutely or fixed do not affect the main document layout, so their reflowing is cheaper as they do not trigger main document reflowing. This is recommended approach for element that need to be animated.
- DOM modifications trigger reflow. This means that operations such as adding new elements, changing the value of text nodes or adding element attributes and their properties cause reflow.
- Good strategies to overcome this limitation are elaborated next.
- Further reading
  - Repaint and reflow at Opera Developer Network
  - Notes on HTML Reflow - more detailed information on the reflow process (archived)
  - Reflows & Repaints: CSS Performance making your JavaScript slow
  - Go With The Reflow
  - Rendering: repaint, reflow/relayout, restyle
- Tools
  - XUL Profiler
  - Mac OS X Quartz Debug (in developer tools part of every OS X) Autoflush drawing setting illustrates how the page is reflown in step-by-step detail. (more)

Use createDocumentFragment()

- Make multiple changes in a DOMDocumentFragment and add the fragment into the DOM in a single operation. This triggers only one reflow.

Slow:

```
var list = ['foo', 'bar', 'baz'],
    elem,
    contents;
for (var i = 0; i < list.length; i++) {
    elem = document.createElement('div');
    content = document.createTextNode(list[i]);
    elem.appendChild(content);
    document.body.appendChild(elem);
}
```

Faster:

```
var fragment = document.createDocumentFragment(),
    list = ['foo', 'bar', 'baz'],
    elem,
    contents;
for (var i = 0; i < list.length; i++) {
    elem = document.createElement('div');
    content = document.createTextNode(list[i]);
    fragment.appendChild(content);
}
document.body.appendChild(fragment);
```

Use cloneNode()

- If you're not working on elements that do not contain form elements or event handlers, you can clone the element to modify and swap it in place after all the changes have been done resulting in a one reflow only.
- A faster alternative to above slow approach is presented below.

Faster:

```
var orig = document.getElementById('container'),
    clone = orig.cloneNode(true),
    list = ['foo', 'bar', 'baz'],
    elem,
    contents;
clone.setAttribute('width', '50%');
for (var i = 0; i < list.length; i++) {
    elem = document.createElement('div');
    content = document.createTextNode(list[i]);
    elem.appendChild(content);
    clone.appendChild(elem);
}
original.parentNode.replaceChild(clone, original);
```

Use HTML templates and innerHTML

(C) Copyright Nokia 2013. All rights reserved.

- One way to implement a templating system is to populate template content based on a light-weight JavaScript object acting as a date model. The data model may be persisted as JSON serialization into e.g. `setPreferenceForKey()` store in S60 WRT, in a HTTP cookie or XHR'd to the server-side.

```
<nowiki>
var model = { title: 'My Test Page'; },
    template = [];
template.push('<h1>' + model.title + '<h1>');
template.push('<div>Another Test Element<div>');
document.getElementById(containerId).innerHTML = template.join('');
// alternatively you can use concat() -- see string concatenation test results
</nowiki>
```

Modify an invisible element

- If the display property of an element is set to none it will not be repainted.
- By setting display to none, do the modifications and then set it to block causes only two reflows

Slow:

```
var subElem = document.createElement('div'),
    elem = document.getElementById('animated');
elem.appendChild(subElem);
elem.style.width = '320px';
```

Faster:

```
var subElem = document.createElement('div'),
    elem = document.getElementById('animated');
elem.style.display = 'none';
elem.appendChild(subElem);
elem.style.width = '320px';
elem.style.display = 'block';
```

Minimize the use of operations determining the dimensions or location of elements

- Determining dimensions or location of elements via getComputedStyle, offsetWidth, scrollWidth and clientWidth properties will force reflow.
- If you take the measurements repeatedly, consider taking them only once.
- This issue is the main cause of slowness in WebKit according to Dave Hyatt

Slow:

```
var elem = document.getElementById('animated');
elem.style.fontSize = (elem.offsetWidth / 10) + 'px';
elem.firstChild.style.marginleft = (elem.offsetWidth / 20) + 'px';
```

Faster:

```
var elem = document.getElementById('animated'),
    elemWidth = elem.offsetWidth;
elem.style.fontSize = (elemWidth / 10) + 'px';
elem.firstChild.style.marginleft = (elemWidth / 20) + 'px';
```

Make multiple predefined style changes at once using className

- As with DOM manipulation, several style changes can be done at the same time.
- Instead if you set the styles one by one, multiple reflows and repaints can be triggered.

Slow:

```
var elem = document.getElementById('styled');
elem.style.background = 'blue';
elem.style.color = 'white';

Faster:
<code html4strict>
<style type="text/css">
div { background: white; color: black; }
div.active { background: blue; color: white; }
</style>
...
```

```
var elem = document.getElementById('styled').className = 'active';
```

Make multiple dynamic style changes at once using setAttribute

- For dynamic animation, using predefined styles does not work. In this case setAttribute object can be used (for IE, use style.cssText property)

Faster:

```
var elem = document.getElementById('styled');
elemStyle = 'background: blue; color: white;';
elem.setAttribute('style', elemStyle);
```

CSS class name vs. style property changing

Changing the class name of an element is a nice way to use JavaScript to dynamically change elements. Performance varies from browser to browser, but generally it is faster to change an element's visual appearance directly via the Javascript style attribute, rather than to change a class name on that element.

Slow: `div.active { border: 1px solid red; }`

Faster (for a one element):

(C) Copyright Nokia 2013. All rights reserved.

```
var container = document.getElementById('container');
container.style.border = '1px solid red';
```

The above method appears to be more efficient when changing a specific number of items. Sometimes a single class name change is effective however. If you need to change all elements under a given container for example, it is more efficient to change the class name of a parent container which holds the affected elements and let CSS do what it does best.

Faster (if multiple child elements of a container need to be changed): `// by changing the class name of the container, all of its child div elements will be updated #container.active div { border: 1px solid red; }`

Depending on the specific case at hand you should use the method which gives you the best performance (without sacrificing too much of the separation of concerns benefits of externally defined CSS).

Source:

- A Snapshot of The Yahoo! Photos Beta - Web Development Perspective

Avoid traversing large number of nodes

- Always try to use inbuilt methods and collections of the DOM to narrow down the search to smallest number of nodes possible.
- Try to avoid manually recursively stepping through the DOM as much as possible.

Slow:

```
var elements = document.getElementsByTagName('*'); // searches every element, slow
for (i = 0; i < elements.length; i++) {
    if (element[i].hasAttribute('selected')) { // continues even through element was found
        ...
    }
}
```

Faster:

```
var elements = document.getElementById('parent').childNodes; // we know the element is a child of parent
for (i = 0; i < elements.length; i++) {
    if (element[i].nodeType == 1 && element[i].hasAttribute('selected') { // first test for valid node type
        ...
        break; // break out of the loop if we found what we were looking for
    }
}
```

Avoid modifications while traversing

- childNodes and NodeList returned by getElementsByTagName() are live. This means that these collections may change without waiting for the execution to finish first.
- If new elements are added to the collections while they are traversed, an infinite loop may occur.
- If new elements are added even outside of collection itself, the collection must look for potential new entries. Due to this it cannot remember its last position or length which need to be recalculated.

Slow:

```
var elems = document.getElementsByTagName('div');
for (var i = 0; i < elems.length; i++) {
    elems[i].appendChild(document.createTextNode(i));
}
```

Faster:

```
var elems = document.getElementsByTagName('div'),
    temp = [];
for (var i = 0; i < elems.length; i++) {
    temp[i] = elems[i]; // first a build static list of elements to modify
}
for (var i = 0; i < temp.length; i++) {
    temp[i].appendChild(document.createTextNode(i)); // perform modifications on static list instead of live NodeList
}
temp = null;
```

Cache DOM values into variables

- Cache frequently accessed DOM values into variables.

Slow:

```
document.getElementById('elem').propertyOne = 'value of first property';
document.getElementById('elem').propertyTwo = 'value of second property';
document.getElementById('elem').propertyThree = 'value of third property';
```

Faster:

```
var elem = document.getElementById('elem').propertyOne = 'value of first property';
elem.propertyTwo = 'value of second property';
elem.propertyThree = 'value of third property'
```


(C) Copyright Nokia 2013. All rights reserved.

Remove references to documents that have been closed

- If a reference to frame, iframe or object is stored in a global variable or a property, clear it by setting it to null or deleting it.
- For example, a popup window that is closed and has a reference to global variable, will be kept in memory although the document itself is no longer loaded, if it is not deleted manually.

Slow:

```
var frame = parent.frames['frameId'].document,
    container = frame.getElementById('contentId'),
    content = frame.createElement('div');
content.appendChild(frame.createTextNode('Some content'));
container.appendChild(content);
```

Faster:

```
var frame = parent.frames['frameId'].document,
    container = frame.getElementById('contentId'),
    content = frame.createElement('div');
content.appendChild(frame.createTextNode('Some content'));
container.appendChild(content);
// nullify references to frame
frame = null;
container = null;
content = null;
```

Object-Oriented JavaScript

Consider alternative inheritance mechanisms

- There exists many ways to mimic Object Orientated style inheritance in JavaScript. However, not all the mechanism to emulate inheritance perform similarly.
- Especially if your code calls overridden methods frequently you should take a look at the results of JavaScript Inheritance Performance study.

Client-server communication

Set timeouts to XMLHttpRequests

- Browsers will timeout XHR requests after some time, but sometimes it is beneficial to abort the connection under script control. This can be done by adding timeouts to XHR calls using setTimeout().

```
var xhr = new XMLHttpRequest ();
xhr.open('GET', url, false);
xhr.onreadystatechange = function () {
    if (this.readyState == 4) {
        clearTimeout(timeout);
        // do something with response data
    }
}
var timeout = setTimeout(
    function () {
        xhr.abort();
        // call error callback
    },
    60*1000 // timeout after a minute
);
xhr.send();
```

Consider using a custom data exchange format for large datasets, as an alternative to XML and JSON

- Depending on the browser, its version and OS, the performance difference between XML or JSON parsing and traversing may be very significant. For example, on Nokia 5800 XpressMusic with a very large dataset JSON is approx. 5x faster.
- Using a custom string-based data format (think CSV) over XML and JSON is efficient in terms of transferred bytes and parse time with large datasets.
- Using JavaScript's String and RegExp methods one can match the speed of JSON executed natively with as little overhead to the file size as possible.

JSON: {{{1}}}

Equivalent using custom data format and String and RegExp methods:

```
that.contacts = o.responseText.split("\\c");

for (var n = 0, len = that.contacts.length, contactSplit; n < len; n++) {

   contactSplit = that.contacts[n].split("\\a");

   that.contacts[n] = {};
   that.contacts[n].n = contactSplit[0];
   that.contacts[n].e = contactSplit[1];
   that.contacts[n].u = contactSplit[2];
   that.contacts[n].r = contactSplit[3];
   that.contacts[n].s = contactSplit[4];
   that.contacts[n].f = contactSplit[5];
   that.contacts[n].a = contactSplit[6];
   that.contacts[n].d = contactSplit[7];
   that.contacts[n].y = contactSplit[8];
}
```

Sources:

- Building Fast Client-side Searches

(C) Copyright Nokia 2013. All rights reserved.

Animations

Use animations modestly

- Animation without hardware support is slow. Try to avoid excessive use of animations which do not bring any real usability value. At least give users an opportunity to disable animations.

Use scrollTo() to animate scrolling

Using native scrolling via scrollTo() performs significantly better as it does not trigger reflow.

Absolutely or fixed position animated elements

- By default elements have style property position: static and animating such elements causes reflowing of the layout and is expensive.
- Elements to be animated should be set as position: absolute or position: fixed if reflowing is not mandatory for smoother animation and lesser CPU load. Such elements do not affect other elements layout, so they will only cause a repaint rather than a full reflow with a nice performance boost.
- CSS positioning schemes for position property:

Position value	Description
static	Default. An element with position: static always has the position the normal flow of the page gives it (a static element ignores any top, bottom, left, or right declarations)
relative	An element with position: relative moves an element relative to its normal position, so left:20 adds 20 pixels to the element's left position
absolute	An element with position: absolute is positioned at the specified coordinates relative to its containing block. The element's position is specified with the left, top, right, and bottom properties.
fixed	An element with position: fixed is positioned at the specified coordinates relative to the browser window. The element's position is specified with the left, top, right, and bottom properties. The element remains at that position regardless of scrolling.

Source:

- w3schools - CSS position Property

Use one timer to animate multiple elements at the same time

- `setTimeout()` and `setInterval()` timers are two basic methods used to implement animation (i.e. trigger changes to element size, position and/or appearance over time).
- If multiple elements are animated at the same time, the best frame rate is achieved by iterating across all animated elements inside a single loop. Using multiple timers makes animation less efficient and consistent, presumably due to timer invocation overhead.

Slow:

```
setInterval(function() {
    animateFirst(arg);
}, 50);
setInterval(function() {
    animateSecond(arg);
}, 50);
function animateFirst(arg) {};
function animateSecond(arg) {};
```

Faster:

```
setInterval(function() {
    animateFirst(arg);
    animateSecond(arg);
}, 50);
function animateFirst(arg) {};
function animateSecond(arg) {};
```

Sources, further reading:

- Javascript Animation: Tutorial, Part 1
- Javascript Animation: Tutorial, Part 2

Trade animation smoothness for speed

To trade smoothness for speed means that while you may want to move an animation 1 pixel at a time, the animation and subsequent reflows may in that case use 100% of the CPU and the animation will seem jumpy as the browser is forced to drop frames to update the flow. Moving the animated element by e.g. 5 pixels at a time may seem slightly less smooth on faster machines, but won't cause CPU thrashing that easily on mobile devices.

Events

Use event delegation

- Assigning event handlers to individual objects can add up quickly and is expensive if you create a lots of new elements dynamically to which event handlers need to be bound.
- This becomes especially interesting if you assign multiple event listeners (e.g. click and blur to a one element. In case of 100 elements that would mean 200 event handlers.
- Using DOM Level 2 event model all events propagate toward the document object which is highest up in the hierarchy. This means that one can bind

(C) Copyright Nokia 2013. All rights reserved.

event listeners to document which invokes a controller and passes the event object to it. The controller is responsible for inspecting the event and dispatching to appropriate logic.

Slow:

```
var elems = [first, ..., last]; // an array which holds say 1000 references to element to which assign the event handlers to
for (var i, l = elems.length; i++; i < l) {
    elems[i].onclick = function() {};
    elems[i].onblur = function() {};
}
```

Faster:

```
//HTML
<button id="doSomething">Click me to do something</button> // you can add more of elements without the need to worry about binding e

// JS
document.addEventListener('click', function(event) { eventController(event); }, false);
document.addEventListener('blue', function(event) { eventController(event); }, false);

function eventController(event) {
   // inspect the event object internals and do something wise
   if (event.target.id === 'doSomething') {
      doSomething();
   }
}

function doSomething() {}
```

Throttle event handlers which fire excessively

- If a handler is called many times, the responsiveness of the UI degrade and tops the CPU. This is especially an issue if the event handler triggers reflow as is the case with resize.
- On S60 5.0 devices input element's blur handler is called each time the content of the input field changes, e.g. user types in a single letter.
- You may want your function to run once after the last event has fired to prevent excessive calls to potentially expensive functions. You must implement throttling mechanism to achieve that kind of a rate limited.

Slow:

```
window.onresize = resizeHandler; // fires excessively during resize
```

Faster:

```
function SomeObject() {
     var self = this;
     this.lastExecThrottle = 500; // limit to one call every "n" msec
     this.lastExec = new Date();
     this.timer = null;
     this.resizeHandler = function() {
         var d = new Date();
         if (d-self.lastExec < self.lastExecThrottle) {
             // This function has been called "too soon," before the allowed "rate" of twice per second
             // Set (or reset) timer so the throttled handler execution happens "n" msec from now instead
             if (self.timer) {
                 window.clearTimeout(self.timer);
             }
             self.timer = window.setTimeout(self.resizeHandler, self.lastExecThrottle);
             return false; // exit
         }
         self.lastExec = d; // update "last exec" time
         // At this point, actual handler code can be called (update positions, resize elements etc.)
         // self.callResizeHandlerFunctions();
     }
}

var someObject = new SomeObject();
window.onresize = someObject.resizeHandler;
```

Source:

- A Snapshot of The Yahoo! Photos Beta - Web Development Perspective

Escaping the JavaScript call stack with `setTimeout`

- Setting up handlers that run after an event has fired needs some trickery. Because the event doesn't take effect until the event-handling call stack opens and closes completely, there's no way to work in a post-event environment via conventional event handlers.
- When something is called using setTimeout with a delay of 0 the JavaScript engine notices it is busy (with a task which invoked `setTimeout`) and queues the `setTimeout` code for execution immediately after the current call stack closes.
- This technique can be used to prioritize certain functionality such as showing and hiding loading indicator prior to executing a computationally heavy operation such as modifying the DOM.

In a typical scenario this will not turn the loading indicator visible:

```
showLoadingIndicator();
doSomethingExpensive();
```

Workaround is to use setTimeout as follows (please take extra care when using this anti-pattern as what it does is actually delays the execution in order to display the loading indicator in the UI):

```
function switchViews() {
    setTimeout(function() {
        showLoadingIndicator();
```


(C) Copyright Nokia 2013. All rights reserved.

```
    }, 0);

    setTimeout(function() {
        doSomethingExpensive();
    }, 50);
}
```

Source:

- Escaping the JavaScript call stack with setTimeout

Styling

Optimize CSS

- Create a component library
- Use consistent semantic styles
- Design modules to be transparent on the inside
- Be flexible
- Learn to love grids
- Minimize selectors
- Separate structure and skin
- Separate container and content
- Extend objects by applying multiple classes to an element
- Use reset and fonts from YUI
- Source: Object Oriented CSS

Change CSS classes near the edges of the DOM tree

To limit the scope of the reflow to as few nodes as possible you should avoid changing a class on wrapper (or body) element(s) which affects the display of many child nodes. Additionally, that may result in re-resolving style on the entire document and for a large DOM that could lock up the browser for a while.

Source:

- Dave Hyatt on CSS Selectors

Avoid tables for layout or use table-layout: fixed

Avoid tables for layout. As if you needed another reason to avoid them, tables often require multiple passes before the layout is completely established because they are one of the rare cases where elements can affect the display of other elements that came before them on the DOM. Imagine a cell at the end of the table with very wide content that causes the column to be completely resized. This is why tables are not rendered progressively in all browsers and yet another reason why they are a bad idea for layout.

It is recommended to use a fixed layout (table-layout: fixed) for data tables to allow a more efficient layout algorithm. This will allow the table to render row by row according to the CSS 2.1 specification.

Source:

- Reflows & Repaints: CSS Performance making your JavaScript slow

Optimize images

- Combine similar colors
- Avoid whitespace in sprites, use CSS instead
- Horizontal is better than vertical
- Limit the number of colors
- First optimize individual images, next sprites in it
- Reduce anti-aliased pixels via size and alignment
- Avoid diagonal gradients due to increased image size
- Avoid alpha transparency (IE does not support)
- Change gradient color every 2-3 pixels

Source:

- Design Fast Websites slides and presentation

S60 specifics

Styling

Remove borders around focusable elements

In WRT 1.x you can remove default blue borders in tab navigation mode with the following CSS. `a:hover, a:focus { outline: none; }`

Fix focus problems in tabbed navigation mode

In tabbed navigation mode the focus may lost while switching views. A fix to this issue is to place a read-only `<input>` text element (can be absolutely positioned, zero-sized and transparent) under the element to be focused after switching the view and `focus()`:ing on it after the view changes.

(C) Copyright Nokia 2013. All rights reserved.

```
/* CSS */
#top {
    position: absolute;
    top: 0px;
    width: 0px;
    height: 0px;
    border: 0px;
    background: transparent;
}

<!-- HTML -->
<input id="top" type="text" readonly="readonly"/>
```

```
// JavaScript
widget.setNavigationEnabled(false); // tabbed navigation mode

function toggleViews() {
    hideFirstView();
    showSecondView();
    document.getElementById('top').focus();
}
```

You may need to alter the positioning of the element by modifying the `top` property to position it underneath the first focusable element depending on your layout.

Prevent elements losing focus in tabbed navigation mode

There is no exhaustive fix to this bug but it seems to help if you set the styles via onblur and onfocus event handlers instead of using CSS pseudo selector :hover.

So instead of `:hover` in CSS:

```
<style type="text/css">
#focusable:hover {
    background-color: blue;
}
</style>
<button id="focusable">Foobar</button>
```

You utilize `onfocus` and `onblur` event handlers and set the styles using JavaScript:

```
document.getElementById('focusable').onfocus = function () {
    this.style.backgroundColor = 'blue';
}

document.getElementById('focusable').onblur = function () {
    this.style.backgroundColor = 'inherit';
}
```

```
<button id="focusable">Foobar</button>
```

Allow animated gif images to animate after being hidden

S60 3.x and 5.0 devices do not animated gif images if they have been set as display: none at any point during their lifespan regardless of subsequent display value. A workaround to this bug is to avoid display: none and use position: absolute and alter left property to move the animated gif outside of the visible viewport to hide it (left: -100%) and bring it back (left: 0).

Alternatively you can fold the animated gif into another element whose style you set in a similar fashion.

```
<!-- animation hidden -->
<img style="position: absolute; left: -100%;" src="animated.gif" />
```

```
<!-- animation visible -->
<img style="position: absolute; left: 0;" src="animated.gif" />
```

Home Screen Widgets

- Remove unneeded images in mini view
  - If images are used exclusively in full view, you can get rid of all of them (or selectively one by one) and save memory.
  - Any other unneeded images (e.g. images specific to screen orientation, portrait or landscape) can be also removed using similar technique.
  - To remove all images use:

    ```
    for (var i in document.images) {
        document.images[i].src = "#";
    }
    ```

- Remove unneeded nodes from the DOM when in mini view
  - {{{1}}}
- Check and remove unused JavaScript functions and <embed> tags when in mini view
  - Removing unused JavaScript functions and <embed> tags will save memory.
- Consider using background color in mini view instead of images
- Respect onshow and onhide
  - When you get an `onhide` event, call `clearInterval()` and `clearTimeout()` for any pending timers and abort any pending XHR calls to keep the CPU usage low.
- Use setInterval wisely
- Animations on Home Screen
  - Avoid animations on home screen unless they are really needed for the sake of good user experience.

(C) Copyright Nokia 2013. All rights reserved.

Misc

Avoid background-repeat if the background-image is small relative to its containing element

Interaction such as scrolling is extremely slow if background-repeat property is set to repeat-x or repeat-y combined with a background-image of a size which needs to be repeated multiple times to fill in the element's background. A workaround is to avoid using repeating backgrounds i.e. use a big enough background-image with `background-repeat: no-repeat`. For example, a commonly used CSS design pattern -- which degrades the performance and should be avoided -- is to create a continuous vertical gradient background using a background image of width 1px which is repeated horizontally across the whole element width.

This optimization pattern is a tradeoff between the initial load time and the subsequent interaction smoothness. In a typical scenario using a bigger background-image (longed load time) with `background-repeat: no-repeat` delivers a far superior overall UX over using a smaller repeating background.

Slow:

```
<style type="text/css">
.container {
    width: 500px;
    height: 50px;
}
.repeat {
    background-image: url(background1.png); /* background1.png width 1px, height 50px */
    background-repeat: repeat-x;
}
</style>

<div class="container repeat">Foobar</div>
```

Faster:

```
<style type="text/css">
.container {
    width: 500px;
    height: 50px;
}
.norepeat {
    background-image: url(background2.png); /* background2.png width 500px, height 50px */
    background-repeat: no-repeat;
}
</style>

<div class="container norepeat">Foobar</div>
```

---

jQuery-specifics

NB! These optimization patterns are aimed at jQuery 1.2.6. Recent versions and the Sizzle selector engine alleviate some of these bottlenecks.

Sources Advanced jQuery with John Resig JavaScript Function Call Profiling

Minimize the use of slow jQuery methods

`remove()`, `hmtl()` and `empty()` have order of n2 time complexity, that is T(n) = O(n2). Minimizing the use of these methods is recommended in jQuery versions prior to 1.3.3. .

`append, prepend, before, after` methods are expensive as well as taking a chunk of HTML and serializing it (`$("
Hello World!
")`).
The process behind these manipulations methods is the following: cleaning the input string, converting the string into a DOM fragment and injecting it into the DOM.

Optimize selectors

Pay attention to the selectors you use. E.g. if you want to hide paragraphs which are direct children of div elements there are multiple ways to do it.

Using the selector below will try to find all div elements, loops thought all of them and find all p relative to the div, merge and figure out unique results. Making sure that the returned set is unique is where the most time is spent, and is very slow especially with a large DOM tree.

Slow:

```
$("div p").hide();
```

A one way to optimize the selector is to match direct children of div (this may require you to re-factor the DOM structure of your app and may not be feasible in every scenario). This alternative flatter selector will try to find all div elements, loops thought all child elements and verifies if element is p.

Faster:

```
$("div > p").hide();
```

Consider alternatives to jQuery.each()

(C) Copyright Nokia 2013. All rights reserved.

Depending on the size of the array, looping through it using `{{{1}}}` instead of `jQuery.each(myArray)` may be faster.

Consider alternatives to .show(), .hide() and .toggle()

Toggling element visibility via `.css({'display':'none'})` and `.css({'display':'block'});` is faster than using convenience functions `.show()`, `.hide()` and `toggle()`, especially while working with large number of elements. Depending on the rendering engine used, `.css()` is also faster than using `.addClass()` and `.removeClass()`.

Slow:

```
$('#elementToHide').hide();
$('#elementToShow').show();
```

Faster:

```
$('#elementToHide').css({'display':'none'});
$('#elementToShow').css({'display':'block'});
```

For dealing with extremely large set of DOM elements, you may want to consider disabling stylesheets as follows:

```
<style id="special_hide">.special_hide { display: none; }</style>
<!--  ...  -->
<div class="special_hide">Special hide DIV</div>
```

```
// show all elements with a class of “special_hide”
$('#special_hide').attr('disabled', 'true');

// hide all elements with a class of “special_hide”
$('#special_hide').attr('disabled', 'false');
```

Sources

- Learning jQuery - show/hide performance]

References

Generic JavaScript

- Y! Best Practices for Speeding Up Your Web Site
- Notes on JS Compression
- Performance Roundup for 2008
- CS193H: High Performance Web Sites at Stanford
- Coupling Async Scripts
- Google I/O: Even Faster Web Sites
- Ajax Experience webcasts
- Douglas Crockford
  - Ajax Performance
  - Advanced JavaScript - Performance
- Mobile Web Application Best Practices (W3C Working Draft)
- JavaScript Performance Rocks!
- Measuring User Experience Performance
- Serving JavaScript Fast
- JavaScript Performance Tips - generic, and Prototype-specific
- JavaScript String Performance Analysis - string concatenation with +=, join() and concat()
- IMB developerWorks Ajax Performance Analysis
- Opera Developer Network
  - Efficient JavaScript - Tips for ECMAScript, DOM and document loading
  - Cross Device Development Techniques for Widgets
  - JavaScript Best Practices
- DOM DocumentFragments
- John Resig
  - JavaScript Engine Speeds
  - Benchmark
  - getElementByClassName Speed Comparison
  - Performance Improvements in Browsers
  - Browser Paint Events
  - JavaScript Performance Stack
  - Native JSON Support is Required
- A Snapshot of The Yahoo! Photos Beta - Web Development Perspective
- Building Fast Client-side Searches
- Even Faster Websites - Steve Souders at SXSW ‘09
- Reflows & Repaints: CSS Performance making your JavaScript slow?
- Nicholas Zakas
  - JavaScript Variable Performance

(C) Copyright Nokia 2013. All rights reserved.

  - slides
  - Speed Up Your JavaScript part 1 part 2, part 3 and part 4
  - JavaScript Stack Overflow Error
- Stoyan Stefanov
  - High Performance Kick Ass Web Apps Video at JSCONF 2009
- Compatibility Tests
  - Quirksmode.org Mobile Tests
- Google Page Speed
- Extreme JavaScript Performance by Thomas Fuchs

HTML5-specific

- Best Practices for a Faster Web App with HTML5

jQuery-specific

- John Resig
  - Advanced jQuery with John Resig
- Extending jQuery's Selector Capabilities]
- JavaScript Function Call Profiling - !FireUnit profiling extensions
- JavaScript (jQuery) performance measurement and best practices
- Improve Your jQuery - 25 Tips
- jQuery Performance Tips
- 43,439 reasons to use append() correctly
- jQuery and JavaScript Coding: Examples and Best Practices

Latest

- Planet Performance - News and views from the web performance blogosphere

This article is originally authored by Anssi Kostiainen and licensed under a Creative Commons Attribution, Non Commercial - Share Alike 2.5 license.

Copyright notice: Nokia may include links to sites on the Internet that are owned or operated by third parties. If you choose to use such links to such sites you agree to review and accept such site's rules of use before using such site. You access the third-party sites at your own risk. Nokia does not assume any responsibility for material created or published by such third-party sites. By providing a link to a third party site Nokia does not imply that Nokia endorses the site or the products or services referenced in such third party site.

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.