Writing Efficient JavaScript
What makes JavaScript slow and what to do about it

Nicholas C. Zakas
Principal Front End Engineer, Yahoo!
Velocity – June 2009
Who's this guy?

- Principal Front End Engineer, Yahoo! Homepage
- YUI Contributor
- Author
is getting tired of javascript. All it does is slow down page navigation and add complicated layouts and consume zillion resources

12:03 PM May 12th from XMPP Gateway

 ultraleetj
 Juan Bello

© 2009 Twitter About Us Contact Blog Status Apps API Search Help Jobs Terms Privacy
Why slow?
We grew up
Browsers didn't
In the Beginning

```javascript
var form = document.forms[0];
if (form.txtName.length == 0){
    alert("You forgot your name!");
    return false;
}
```
Now
What's the problem?
No compilation!*

* Humor me for now. It'll make this easier.
**Compiler optimization** is the process of tuning the output of a compiler to minimize or maximize some attribute of an executable computer program. The most common requirement is to minimize the time taken to execute a program; a less common one is to minimize the amount of memory occupied. The growth of portable computers has created a market for minimizing the power consumed by a program.

It has been shown that some code optimization problems are NP-complete, or even undecidable. In practice, factors such as the programmer's willingness to wait for the compiler to complete its task place upper limits on the optimizations that a compiler implementor might provide. (Optimization is generally a very CPU- and memory-intensive process.) In the past, computer memory limitations were also a major factor in limiting which optimizations could be performed.

**Contents**

1. Types of optimizations
2. Factors affecting optimization
3. Optimization techniques
   3.1 Common themes
   3.2 Optimization techniques
      3.2.1 Loop optimizations
      3.2.2 Data-flow optimizations
      3.2.3 SSA-based optimizations
      3.2.4 Code generator optimizations
      3.2.5 Functional language optimizations
      3.2.6 Other optimizations
   3.3 Interprocedural optimizations
4. Problems with optimization
5. List of compiler optimizations
6. List of static code analyses
7. See also
8. References
9. External links
Browsers won't help your code!!!!
Didn't Matter Then

```javascript
var form = document.forms[0];
if (form.txtName.length == 0) {
  alert("You forgot your name!");
  return false;
}
```
Didn't Matter Then

- JavaScript used for simple form validation or image hovers
- Slow Internet connections
  - People expected to wait
- Click-and-go model
- Each page contained very little code
Matters Now

- Ajax and Web 2.0
- More JavaScript code than ever before
- Fast Internet connections
  - People have come to expect speed
- Applications that stay open for a long time
  - Gmail
  - Facebook
- Download and execute more code as you interact
Who will help your code?
ONLY YOU
Disclaimer

What follows are graphic depictions of the parts of JavaScript that are slow. Where appropriate, the names of the offenders have been changed to protect their identities. All of the data, unless otherwise noted, is for the browsers that are being used by the majority of web users right now, in 2009. The techniques presented herein will remain valid at least for the next 2-3 years. None of the techniques will have to be reversed once browsers with super powers are the norm and handle all optimizations for us. You should not take the techniques addressed in this presentation as things you should do all of the time. Measure your performance first, find the bottlenecks, then apply the appropriate techniques to help your specific bottlenecks. Premature optimization is fruitless and should be avoided at all costs.
JavaScript Performance Issues

- Scope management
- Data access
- Loops
- DOM
- Browser limits
function setup(items) {

    var divs = document.getElementsByTagName("div");
    var images = document.getElementsByTagName("img");
    var button = document.getElementById("save-btn");

    for (var i=0; i < items.length; i++) {
        process(items[i], divs[i]);
    }

    button.addEventListener("click", function(event) {
        alert("Saved!");
    }, false);
}
Scope Chains

setup() → Scope Chain → Global

- document: object
- window: object
- navigator: object
When a Function Executes

- An execution context is created
- The context's scope chain is initialized with the members of the function's [[Scope]] collection
- An activation object is created containing all local variables
- The activation object is pushed to the front of the context's scope chain
Execution Context

Identifier Resolution
- Start at scope chain position 0
- If not found go to position 1
- Rinse, repeat
Identifier Resolution

- Local variables = fast!
- The further into the chain, the slower the resolution
Scope Chain Augmentation

- The `with` statement
- The `catch` clause of `try-catch`
- Both add an object to the front of the scope chain
Inside of Global Function

**Execution Context**

```
Scope
```

**Scope Chain**

```
<table>
<thead>
<tr>
<th>Scope</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
</table>
```

**Activation Object**

```
<table>
<thead>
<tr>
<th>this</th>
<th>window</th>
</tr>
</thead>
<tbody>
<tr>
<td>arguments</td>
<td>[items]</td>
</tr>
<tr>
<td>items</td>
<td>(array)</td>
</tr>
<tr>
<td>divs</td>
<td>undefined</td>
</tr>
<tr>
<td>images</td>
<td>undefined</td>
</tr>
<tr>
<td>button</td>
<td>undefined</td>
</tr>
<tr>
<td>i</td>
<td>undefined</td>
</tr>
</tbody>
</table>
```

**Global**

```
| document | (object) |
| window | (object) |
| navigator | (object) |
```
Inside of **with/catch** Statement

- Local variables now in second slot
- **with/catch** variables now in first slot
“with statement considered harmful”
-Douglas Crockford
Closures

• The `[[Scope]]` property of closures begins with at least two objects
• Calling the closure means three objects in the scope chain (minimum)
function setup(items) {

    var divs = document.getElementsByTagName("div");
    var images = document.getElementsByTagName("img");
    var button = document.getElementById("save-btn");

    for (var i = 0; i < items.length; i++) {
        process(items[i], divs[i]);
    }

    button.addEventListener("click", function(event) {
        alert("Saved!");
    }, false);
}

Inside of Closure

Execution Context:
- Scope

Scope Chain:
- 0
- 1
- 2

Activation Object:
- `this`: button
- `arguments`: [event]
- `event`: (Event)

Activation Object:
- `this`: window
- `arguments`: [items]
- `items`: (array)

Global:
- `document`: (object)
- `window`: (object)
- `navigator`: (object)
Recommendations

• Store out-of-scope variables in local variables
  – Especially global variables

• Avoid the `with` statement
  – Adds another object to the scope chain, so local function variables are now one step away
  – Use local variables instead

• Be careful with `try-catch`
  – The `catch` clause also augments the scope chain

• Use closures sparingly

• Don't forget `var` when declaring variables
function setup(items) {

    var doc = document;
    var divs = doc.getElementsByTagName("div");
    var images = doc.getElementsByTagName("img");
    var button = doc.getElementById("save-btn");

    for (var i=0; i < items.length; i++) {
        process(items[i], divs[i]);
    }

    button.addEventListener("click", function(event) {
        alert("Saved!");
    }, false);
}

JavaScript Performance Issues

- Scope management
- Data access
- Loops
- DOM
- Browser limits
Places to Access Data

- Literal value
- Variable
- Object property
- Array item

```javascript
//literal
var name = "Nicholas";

//variable
var name2 = name;

//object property
var name3 = object.name;

//array item
var name4 = items[0];
```
Data Access Performance

• Accessing data from a literal or a local variable is fastest
  – The difference between literal and local variable is negligible in most cases

• Accessing data from an object property or array item is more expensive
  – Which is more expensive depends on the browser
Property Depth

- `object.name < object.name.name`
- The deeper the property, the longer it takes to retrieve
Property Notation

- Difference between `object.name` and `object["name"]`?
  - Generally no
  - Exception: Dot notation is faster in Safari
Recommendations

• Store these in a local variable:
  – Any object property accessed more than once
  – Any array item accessed more than once

• Minimize deep object property/array item lookup
function process(data)
{
    if (data.count > 0)
    {
        for (var i=0; i < data.count; i++)
        {
            processData(data.item[i]);
        }
    }
}
function process(data) {
    var count = data.count,
        item = data.item;
    if (count > 0) {
        for (var i=0; i < count; i++) {
            processData(item[i]);
        }
    }
}
JavaScript Performance Issues

- Scope management
- Data Access
- Loops
- DOM
- Browser limits
Loops

• ECMA-262, 3rd Edition:
  - for
  - for-in
  - do-while
  - while

• ECMA-357, 2nd Edition:
  - for each
// for loop
for (var i=0; i < values.length; i++){
    process(values[i]);
}

// do-while loop
var j=0;
do {
    process(values[j++]);
} while (j < values.length);

// while loop
var k=0;
while (k < values.length){
    process(values[k++]);
}
Which loop?
It doesn't matter!
What Does Matter?

• Amount of work done per iteration
  – Includes terminal condition evaluation and incrementing/decrementing
• Number of iterations
• These don't vary by loop type
Fixing Loops

• Decrease amount of work per iteration
• Decrease number of iterations
//for loop
for (var i=0; i < values.length; i++) {
    process(values[i]);
}

//do-while loop
var j=0;
do {
    process(values[j++]);
} while (j < values.length);

//while loop
var k=0;
while (k < values.length) {
    process(values[k++]);
}
// for loop
for (var i=0; i < values.length; i++){
    process(values[i]);
}

// do-while loop
var j=0;
do {
    process(values[j++]);
} while (j < values.length);

// while loop
var k=0;
while (k < values.length){
    process(values[k++]);
}
// for loop
for (var i=0; i < values.length; i++) {
  process(values[i]);
}

// do-while loop
var j=0;
do {
  process(values[j++]);
} while (j < values.length);

// while loop
var k=0;
while (k < values.length) {
  process(values[k++]);
}
Easy Fixes

• Eliminate object property/array item lookups
```javascript
var len = values.length;

// for loop
for (var i=0; i < len; i++){
    process(values[i]);
}

// do-while loop
var j=0;
do {
    process(values[j++]);
} while (j < len);

// while loop
var k=0;
while (k < len)
    process(values[k++]);
```
Easy Fixes

• Eliminate object property/array item lookups
• Combine control condition and control variable change
  – Work avoidance!
var len = values.length;

//for loop
for (var i=0; i < len; i++){
    process(values[i]);
}

//do-while loop
var j=0;
do {
    process(values[j++]);
} while (j < len)

//while loop
var k=0;
while (k < len){
    process(values[k++]);
}
var len = values.length;

//for loop
for (var i=len; i--; ) {
    process(values[i]);
}

//do-while loop
var j = len - 1
do {
    process(values[j--]);
} while (j--);

//while loop
var k = len;
while (k--) {
    process(values[k]);
}
Easy Fixes

• Eliminate object property/array item lookups
• Combine control condition and control variable change
  – Work avoidance!
Things to Avoid for Speed

- **ECMA-262, 3rd Edition:**
  - `for-in`
- **ECMA-357, 2nd Edition:**
  - `for each`
- **ECMA-262, 5th Edition:**
  - `array.forEach()`
- **Function-based iteration:**
  - `jQuery.each()`
  - `Y.each()`
  - `$each`
  - `Enumerable.each()`
values.forEach(function(value, index, array) {
    process(value);
});

- Introduces additional function
- Function requires execution (execution context created, destroyed)
- Function also creates additional object in scope chain
JavaScript Performance Issues

- Scope management
- Data Access
- Loops
- DOM
- Browser limits
HTMLCollection
HTMLCollection Objects

- `document.images`, `document.forms`, etc.
- `getElementsByName()`
- `getElementsByClassName()`
2.3. Miscellaneous Object Definitions

Interface HTMLCollection

An HTMLCollection is a list of nodes. An individual node may be accessed by either ordinal index or the node’s name or id attributes.

Note: Collections in the HTML DOM are assumed to be live meaning that they are automatically updated when the underlying document is changed.

IDL Definition

Note: Collections in the HTML DOM are assumed to be live meaning that they are automatically updated when the underlying document is changed.

length
This attribute specifies the length or size of the list.

Methods

item
This method retrieves a node specified by ordinal index. Nodes are numbered in tree order (depth-first traversal order).

Parameters

index  The index of the node to be fetched. The index origin is 0.

Return Value

The Node at the corresponding position upon success. A value of null is returned if the index is out of range.

This method raises no exceptions.
	namedItem
This method retrieves a Node using a name. It first searches for a Node with a matching id attribute. If it doesn't find one, it then searches for a Node with a matching name attribute, but only on those elements that are allowed a name attribute.

Parameters
Infinite Loop!

```javascript
var divs = document.getElementsByTagName("div");

for (var i=0; i < divs.length; i++) {
    var div = document.createElement("div");
    document.body.appendChild(div);
}
```
**HTMLCollection Objects**

- Look like arrays, but aren't
  - Bracket notation
  - `length` property
- Represent the results of a specific query
- The query is re-run each time the object is accessed
  - Include accessing `length` and specific items
  - Much slower than accessing the same on arrays
- Exceptions: Opera, Safari
```javascript
var items = [{}, {}, {}, {}, {}, {}, {}, {}, {}, {}];
for (var i=0; i < items.length; i++) {
}

var divs = document.getElementsByTagName("div");
for (var i=0; i < divs.length; i++) {
}
```
```javascript
var items = [{}], {}, {}, {}, {}, {}, {}, {}, {}, {};
for (var i=0, len=items.length; i < len; i++) {
}

var divs = document.getElementsByTagName("div");
for (var i=0, len=divs.length; i < len; i++) {
}
```
**HTMLCollection Objects**

- Minimize property access
  - Store length, items in local variables if used frequently
- If you need to access items in order frequently, copy into a regular array
function array(items) {
    try {
        return Array.prototype.concat.call(items);
    } catch (ex) {

        var i = 0,
            len = items.length,
            result = Array(len);

        while (i < len) {
            result[i] = items[i];
            i++;
        }

        return result;
    }
}
Repaint & Reflow
Repaint...is what happens whenever something is made visible when it was not previously visible, or vice versa, without altering the layout of the document.

- Mark 'Tarquin' Wilton-Jones, Opera
When Repaint?

- Change to *visibility*
- Formatting styles changed
  - Backgrounds
  - Borders
  - Colors
  - Anything that doesn't change the size, shape, or position of the element but does change its appearance
- When a reflow occurs
Reflow is the process by which the geometry of the layout engine's formatting objects are computed.

- Chris Waterson, Mozilla
When Reflow?

• Initial page load
• Browser window resize
• DOM nodes added or removed
• Layout styles applied
• Layout information retrieved
Addressing Repaint & Reflow

- Perform DOM changes off-document
- Groups style changes
- Cache retrieved layout information
```javascript
var list = document.getElementById("list");

for (var i=0; i < 10; i++){
    var item = document.createElement("li");
    item.innerHTML = "Option #" + (i+1);
    list.appendChild(item);
}
```
Off-Document Operations

• Fast because there's no repaint/reflow

• Techniques:
  – Remove element from the document, make changes, insert back into document
  – Set element's display to “none”, make changes, set display back to default
  – Build up DOM changes on a DocumentFragment then apply all at once
DocumentFragment

- A document-like object
- Not visually represented
- Considered to be owned by the document from which it was created
- When passed to `addChild()`, appends all of its children rather than itself
```javascript
var list = document.getElementById("list");
var fragment = document.createDocumentFragment();

for (var i=0; i < 10; i++){
    var item = document.createElement("li");
    item.innerHTML = "Option #" + (i+1);
    fragment.appendChild(item);
}
list.appendChild(fragment);
```

**Reflow!**

**No reflow!**
Addressing Repaint & Reflow

• Perform DOM changes off-document
• Group style changes
• Cache retrieved layout information
element.style.color = "red";
element.style.height = "100px";
element.style.fontSize = "25px";
element.style.backgroundColor = "white";
What to do?

- Minimize changes on `style` property
- Define CSS class with all changes and just change `className` property
- Set `cssText` on the element directly
.active {
    color: red;
    height: 100px;
    width: 25px;
    background-color: white;
}

element.className = "active";
element.style.cssText = "color: red; " +
"height: 100px;" +
"width: 25px; " +
"background-color: white";

Reflow!
Addressing Repaint & Reflow

- Perform DOM changes off-document
- Group style changes
- Cache retrieved layout information
  - Reflow may be cached
```javascript
var width = element.offsetWidth;
var scrollTop = element.scrollTop;
var display = window.getComputedStyle(div, '').getPropertyValue("display");
```
What to do?

- Minimize access to layout information
- If a value is used more than once, store in local variable
Speed Up Your DOM

- Be careful using `HTMLCollection` objects
- Perform DOM manipulations off the document
- Group CSS changes to minimize repaint/reflow
- Be careful when accessing layout information
JavaScript Performance Issues

- Scope management
- Data Access
- Loops
- DOM
- Browser limits
Call Stack

Runaway Timer
Call Stack

- Controls how many functions can be executed in a single process
- Varies depending on browser and JavaScript engine
- Errors occur when call stack size is exceeded
### Call Stack Size

<table>
<thead>
<tr>
<th>Browser</th>
<th>Call Stack Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firefox 3</td>
<td>3000</td>
</tr>
<tr>
<td>Firefox 3.5 Beta 4</td>
<td>3000</td>
</tr>
<tr>
<td>Chrome 1</td>
<td>21837</td>
</tr>
<tr>
<td>Chrome 2</td>
<td>21837</td>
</tr>
<tr>
<td>Internet Explorer 7</td>
<td>1789</td>
</tr>
<tr>
<td>Internet Explorer 8</td>
<td>2232</td>
</tr>
<tr>
<td>Opera 9.62</td>
<td>10000</td>
</tr>
<tr>
<td>Opera 10 Beta</td>
<td>10000</td>
</tr>
<tr>
<td>Safari 3.2</td>
<td>500</td>
</tr>
<tr>
<td>Safari 4</td>
<td>37448</td>
</tr>
</tbody>
</table>

Note: Internet Explorer changes call stack size based on available memory.
Call Stack Overflow

• Error messages
  – IE: “Stack overflow at line x”
  – Firefox: “Too much recursion”
  – Safari: “Maximum call stack size exceeded.”
  – Opera: “Abort (control stack overflow)”
  – Chrome: n/a

• Browsers throw a regular JavaScript error when this occurs
  – Exception: Opera just aborts the script
Runaway Script Timer

- Designed to prevent the browser from affecting the operating system
- Limits the amount of time a script is allowed to execute
- Two types of limits:
  - Execution time
  - Number of statements
- Always pops up a scary dialog to the user
- Exception: Opera has no runaway timer
Stop running this script?

A script on this page is causing Internet Explorer to run slowly. If it continues to run, your computer may become unresponsive.

Yes  No
Firefox

Warning: Unresponsive script

A script on this page may be busy, or it may have stopped responding. You can stop the script now, open the script in the debugger, or let the script continue.


☐ Don't ask me again

Stop script    Debug script    Continue
Safari

Slow Script
A script on the page file://///C:/Documents%20and%20Settings/Nicholas/Desktop/LongRunningScriptTest.htm is making Safari unresponsive. Do you want to continue running the script, or stop it?

Stop

Continue
Chrome

The following page(s) have become unresponsive. You can wait for them to become responsive or kill them.

LongRunningScriptTest.htm

Kill pages

Wait
Runaway Script Timer Limits

- Internet Explorer: 5 million statements
- Firefox: 10 seconds
- Safari: 5 seconds
- Chrome: Unknown, hooks into normal crash control mechanism
- Opera: none
The Browser UI Thread
• Shared between JavaScript and UI updates
• Only one can happen at a time
• Page UI frozen while JavaScript is executing
• A queue of actions is kept containing what to do next
Browser Limit Causes

• Too much DOM interaction
  – Repaint & reflow
• Too much recursion
• Long-running loops
RECURSION
It recurs.

RECURSION

RECURSION
It recurs.

RECURSION

RECURSION

RECURSION
It recurs.

RECURSION

RECURSION

RECURSION
It recurs.

RECURSION

RECURSION

RECURSION

RECURSION

RECURSION

RECURSION
It recurs.

RECURSION

RECURSION

RECURSION

RECURSION

RECURSION

RECURSION

RECURSION

RECURSION

RECURSION
It recurs.

RECURSION
Recursion Pattern #1

```javascript
function recurse()
{
    recurse();
}

recurse();
```
function doSomething() {
    doSomethingElse();
}

function doSomethingElse() {
    doSomething();
}

doSomething();
Recursion Solutions

• Iteration
function merge(left, right) {
    var result = [];

    while (left.length > 0 && right.length > 0) {
        if (left[0] < right[0]) {
            result.push(left.shift());
        } else {
            result.push(right.shift());
        }
    }

    return result.concat(left).concat(right);
}

// recursive merge sort algorithm
function mergeSort(items) {
    if (items.length == 1) {
        return items;
    }

    var middle = Math.floor(items.length / 2),
        left = items.slice(0, middle),
        right = items.slice(middle);

    return merge(mergeSort(left), mergeSort(right));
}
// iterative merge sort algorithm
function mergeSort(items){
    if (items.length == 1) {
        return items;
    }

    var work = [];
    for (var i=0, len=items.length; i < len; i++){
        work.push([items[i]]);
    }
    work.push([]);  // in case of odd number of items

    for (var lim=len; lim > 1; lim = (lim+1)/2){
        for (var j=0, k=0; k < lim; j++, k+=2){
            work[j] = merge(work[k], work[k+1]);
        }
        work[j] = [];
    }  // in case of odd number of items

    return work[0];
}
Recursion Solutions

- Iteration
- Memoization
function factorial(n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
function factorial(n){

    // create a cache
    if (!factorial.cache) {
        factorial.cache = {
            "0": 1,
            "1": 1
        };
    }

    if (!factorial.cache.hasOwnProperty(n)) {
        factorial.cache[n] = n * factorial(n - 1);
    }

    return factorial.cache[n];
}
function memoizer(fundamental, cache) {
    cache = cache || {};
    var shell = function(arg) {
        if (!cache.hasOwnProperty(arg)) {
            cache[arg] = fundamental(arg);
        }
        return cache[arg];
    };
    return shell;
}

var memoized = memoizer(original, { "arg": "value" });
Browser Limit Causes

• Too much DOM interaction
  – Repaint & reflow
• Too much recursion
• Long-running loops
  – Too much per iteration
  – Too many iterations
  – Lock up the browser UI
setTimeout()
• Schedules a task to be added to the UI queue later
• Can be used to yield the UI thread
• Timer functions begin with a new call stack
• Extremely useful for avoiding browser limits
function swap(items, firstIndex, secondIndex) {
    var temp = items[firstIndex];
    items[firstIndex] = items[secondIndex];
    items[secondIndex] = temp;
}

function bubbleSort(items) {
    var len = items.length,
        i, j;

    for (i = len - 1; i >= 0; i--) {
        for (j = len - i; j >= 0; j--) {
            if (items[j] < items[j - 1]) {
                swap(items, j, j - 1);
            }
        }
    }

    return items;
}
function bubbleSort(array, onComplete)
{
    var pos = 0;

    (function()
    {
        var j;

        for (j=array.length; j > pos; j--){
            if (array[j] < array[j-1]){
                swap(array, j, j-1);
            }
        }
    })
    pos++;

    if (pos < array.length){
        setTimeout(arguments.callee, 10);
    } else {
        onComplete();
    }
})();
}
function chunk(array, process, context){
    var items = array.concat();  //clone the array
    setTimeout(function(){
        var item = items.shift();
        process.call(context, item);

        if (items.length > 0){
            setTimeout(arguments.callee, 100);
        }
    }, 100);
}

chunk(items, function(value){
    console.log(value);
});
Avoiding Browser Limits

• Mind your DOM
  – Limit repaint/reflow

• Mind your recursion
  – Consider iteration or memoization

• Mind your loops
  – Keep small, sprinkle setTimeout() liberally if needed
Will it be like this forever?
No
Browsers With Optimizing Engines

• Chrome (V8)
• Safari 4+ (Nitro)
• Firefox 3.5+ (TraceMonkey)
• Opera 10? 11? (Carakan)

All use native code generation and JIT compiling to achieve faster JavaScript execution.
Summary
ONLY YOU
PLEASE DO NOT TOUCH.

TOUCHING CAN HARM THE ART.
WARNING

Permanently wet paint
Questions?
Etcetera

- My blog: www.nczonline.net
- My email: nzakas@yahoo-inc.com
- Twitter: @slicknet
Creative Commons Images Used

• http://www.flickr.com/photos/neogabox/3367815587/
• http://www.flickr.com/photos/lydz/3355198458/
• http://www.flickr.com/photos/37287477@N00/515178157/
• http://www.flickr.com/photos/ottoman42/455242/
• http://www.flickr.com/photos/hippie/2406411610/
• http://www.flickr.com/photos/flight saber/2204113449/
• http://www.flickr.com/photos/crumbs/2702429363/
• http://www.flickr.com/photos/oberazzi/318947873/
• http://www.flickr.com/photos/vox_efx/2912195591/
• http://www.flickr.com/photos/fornal/385054886/
• http://www.flickr.com/photos/29505605@N08/3198765320/
• http://www.flickr.com/photos/torley/2361164281/
• http://www.flickr.com/photos/rwp-roger/171490824/