
Finding and
debugging
memory leaks in
JavaScript with
Chrome DevTools

$ whoami

Gonzalo Ruiz de Villa
@gruizdevilla

$ aboutthis

I co-founded & work @adesis

This presentation was make for my workshop at #spainjs 2013


What is a memory leak?

Gradual loss of available computer memory

when a program repeatedly fails to return memory that it has obtained for temporary use.

My users have laptops with 16GB of RAM.

So, why should I care?

Common belief

More memory === Better performance

Reality

Memory footprint

is strongly correlated with

increased latencies and variance

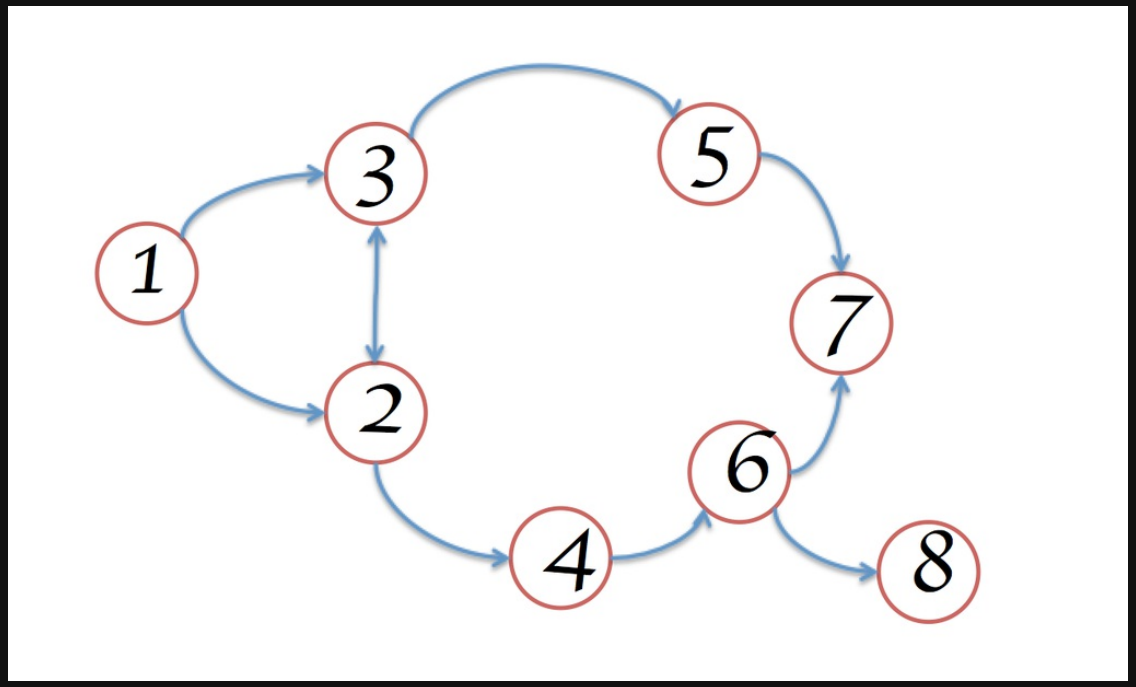
Nothing is free:

(cheap or expensive)

you will always pay a price
for the resources you use

So, let's talk about memory

Think of memory as a graph



Three primitive types:

Numbers (e.g, 3.14159...)
Booleans (true or false)
Strings (e.g, "Werner Heisenberg")

They cannot reference other values.
They are always leafs or terminating nodes.

Everything else is an "Object"

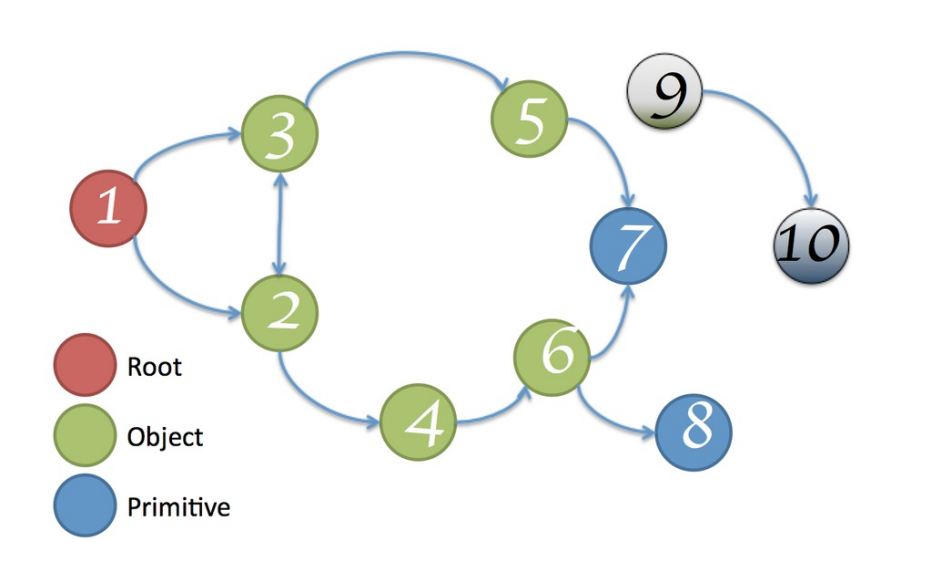
Objects are associative arrays (maps or dictionaries)

So, the object is composed of a collection of (key, value) pairs

And what about *Arrays*?

An Array is an Object
with numeric keys.

The memory graph starts with a root

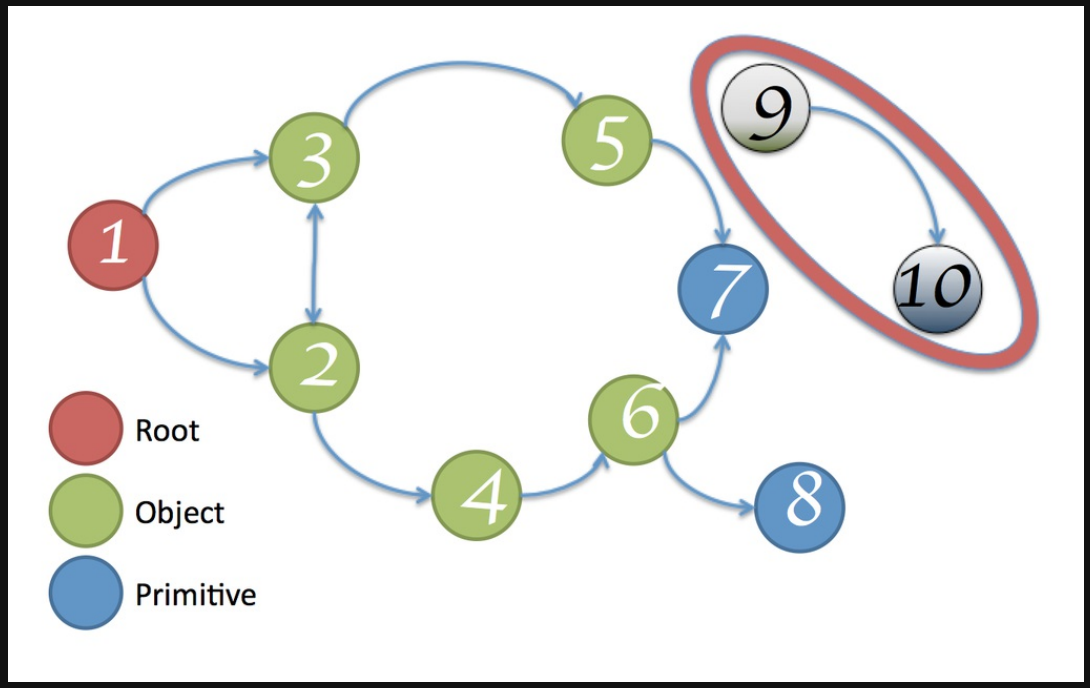


It may be the window object of the browser, or the Global object of a Node.js module.

You don't control how this root object is GC

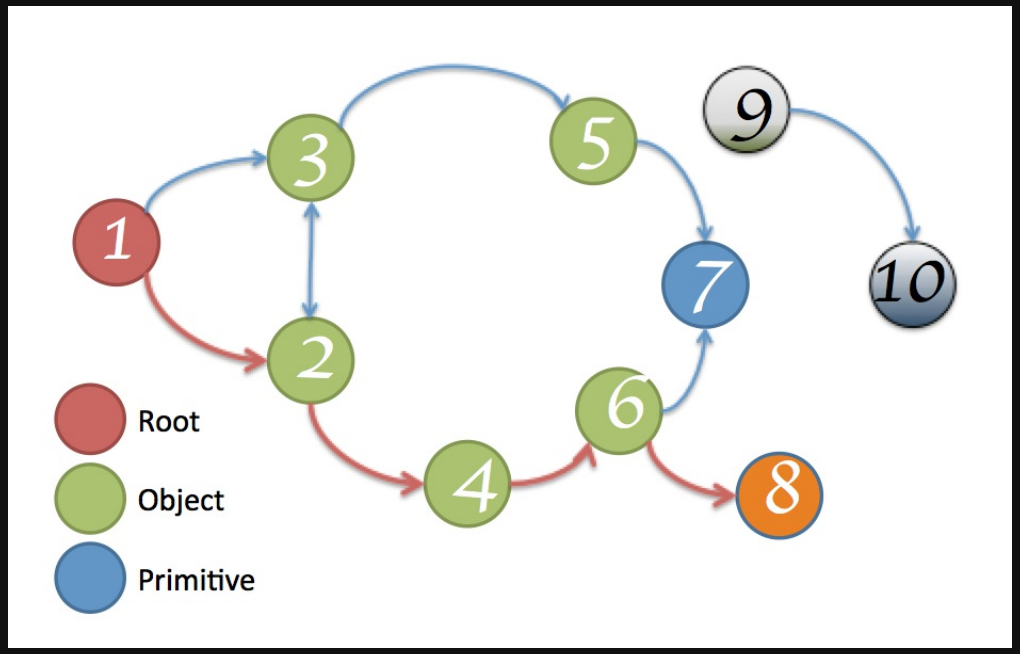
What does get GC?

Whatever is not reachable from the root.

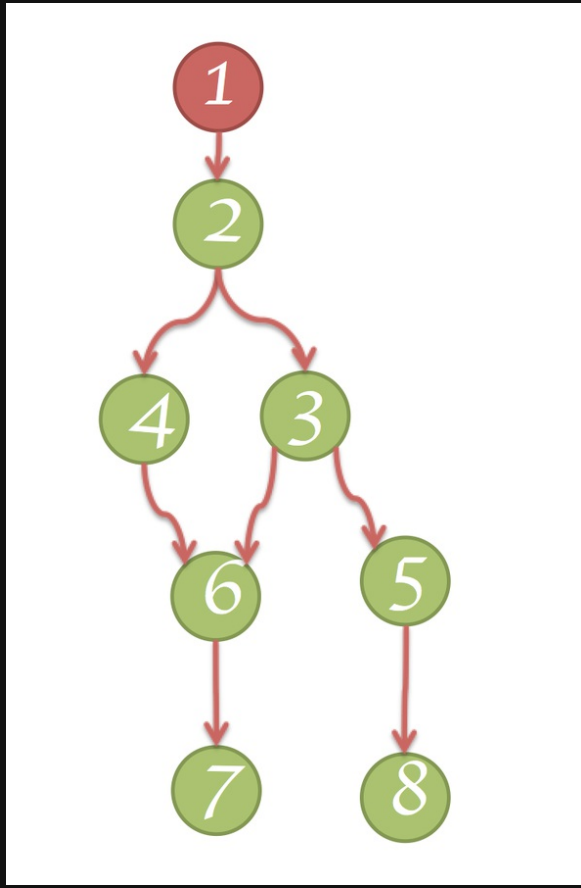


Retaining path

We call a retaining path any path from GC roots to a particular object



Dominators



Node 1 dominates node 2
Node 2 dominates nodes 3, 4 and 6
Node 3 dominates node 5
Node 5 dominates node 8
Node 6 dominates node 7

Some facts about the V8 Garbage Collector

Generational Collector

Age of a value

The age of a value: number of bytes allocated since it was allocated.

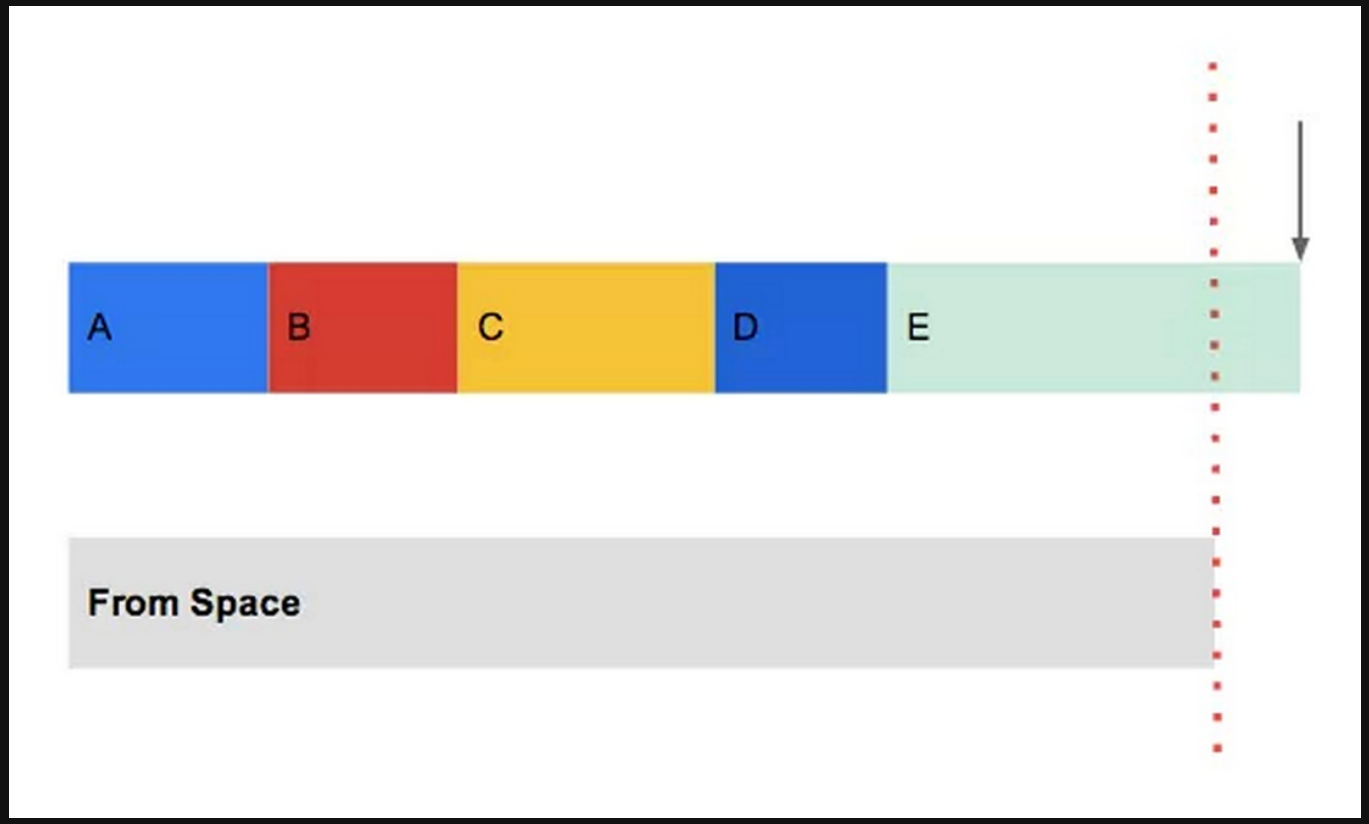
Young Generation

- Splited in two spaces: named "to" and "from"
- "to space": very fast allocation
- filling the "to space" triggers a collection:
  - "to" and "from" swap
  - maybe promotion to old generation
  - ~10ms (remember 60fps -> ~16ms)

Old Generation

- Old generation collection is slow.

"To" and "From" spaces



Remember: triggering a collection pauses your application.

Some de-reference common errors

Be careful wit the delete keyword.

"o" becomes an SLOW object.

```
var o = {x:"y"};
delete o.x;
o.x;   //undefined
```

It is better to set "null".

```
var o = {x:"y"};
o = null;
o.x;   //TypeError
```

Only when the last reference to an object is removed, is that object eligible for collection.

A word on "slow" objects

- V8 optimizing compiler makes assumptions on your code to make optimizations.
- It transparently creates hidden classes that represent your objects.
- Using this hidden classes, V8 works much faster. If you "delete" properties, these assumptions are no longer valid, and the code is de-optimized, slowing your code.

Fast Object

```
function FastPurchase(units, price) {
   this.units = units;
   this.price = price;
   this.total = 0;
   this.x = 1;
}
var fast = new FastPurchase(3, 25);
```

"fast" objects are faster

Slow Object

```
function SlowPurchase(units, price) {
   this.units = units;
   this.price = price;
   this.total = 0;
   this.x = 1;
}
var slow = new SlowPurchase(3, 25);
//x property is useless
//so I delete it
delete slow.x;
```

"slow" should be using a smaller memory footprint than "fast" (1 less property), shouldn't it?

REALITY: "SLOW" is using 15 times more memory

Constructor	Distance	Objects Count		Shallow Size		Retained Size	
▶ SlowPurchase	3	300 001	31 %	3 600 012	3 %	127 200 104	89 %
▶ FastPurchase	3	300 001	31 %	8 400 012	6 %	8 400 104	6 %

Timers

Timers are a very common source of memory leaks.

Look at the following code:

```
var buggyObject = {
 callAgain: function () {
  var ref = this;
  var val = setTimeout(function () {
   console.log('Called again: '
   + new Date().toTimeString());
   ref.callAgain();
  }, 1000);
 }
};
```

If we run:

```
buggyObject.callAgain();
buggyObject = null;
```

With this we have a memory leak:

Closures

Closures can be another source of memory leaks. Understand what references are retained in the closure.

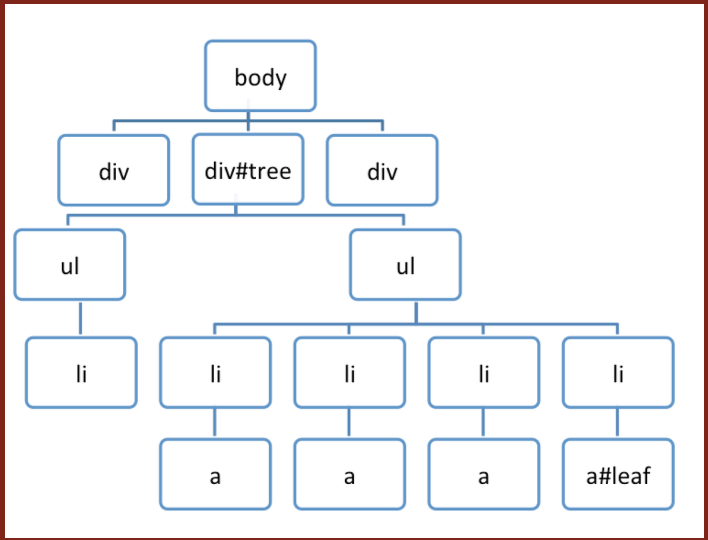
And remember: eval is evil

```
var a = function () {
  var largeStr =
        new Array(1000000).join('x');
  return function () {
    return largeStr;
  };
}();

var a = function () {
    var smallStr = 'x',
          largeStr =
              new Array(1000000).join('x');
   return function (n) {
      return smallStr;
   };
}();

var a = function () {
   var smallStr = 'x',
         largeStr =
             new Array(1000000).join('x');
    return function (n) {
        eval(''); //maintains reference to largeStr
        return smallStr;
     };
}();
```

DOM leaks are bigger than you think



When is the #tree GC?

```
var select = document.querySelector;
var treeRef = select("#tree");
var leafRef = select("#leaf");
var body = select("body");
body.removeChild(treeRef);
//#tree can't be GC yet due to treeRef
treeRef = null;
//#tree can't be GC yet, due to
//indirect reference from leafRef
leafRef = null;
//NOW can be #tree GC
```

#leaf maintains a reference to it's parent (parentNode), and recursively up to #tree, so only when leafRef is nullified is the WHOLE tree under #tree candidate to be GC

Rules of thumb

Use appropiate scope	Better than de-referencing, use local scopes.
Unbind event listeners	Unbind events that are no longer needed, specially if the related DOM objects are going to be removed.
Manage local cache	Be careful with storing large chunks of data that you are not going to use.

Object Pools

Young generation GC takes about 10ms.

Maybe it is too much time for you:

Instead of allocating and deallocating objects, reuse them with object pools.

Note: object pools have their own drawbacks (for example, cleaning used objects)

Three key questions

1. Are you using too much memory?
2. Do you have memory leaks?
3. Is your app GCing too often?

Knowing your arsenal

Browser Info

You can measure how your users are using memory.

```
> performance.memory
MemoryInfo {
jsHeapSizeLimit: 793000000,
usedJSHeapSize: 27600000,
totalJSHeapSize: 42100000
}
```

You can monitor their activity to detect unexpected use of memory
(only in Chrome)

jsHeapSizeLimit	the amount of memory that JavaScript heap is limited to
usedJSHeapSize	the amount of memory that JavaScript has allocated (including free space)
totalJSHeapSize	the amount of memory currently being used

If usedJSHeapSize grows close to jsHeapSizeLimit there is a risk of:

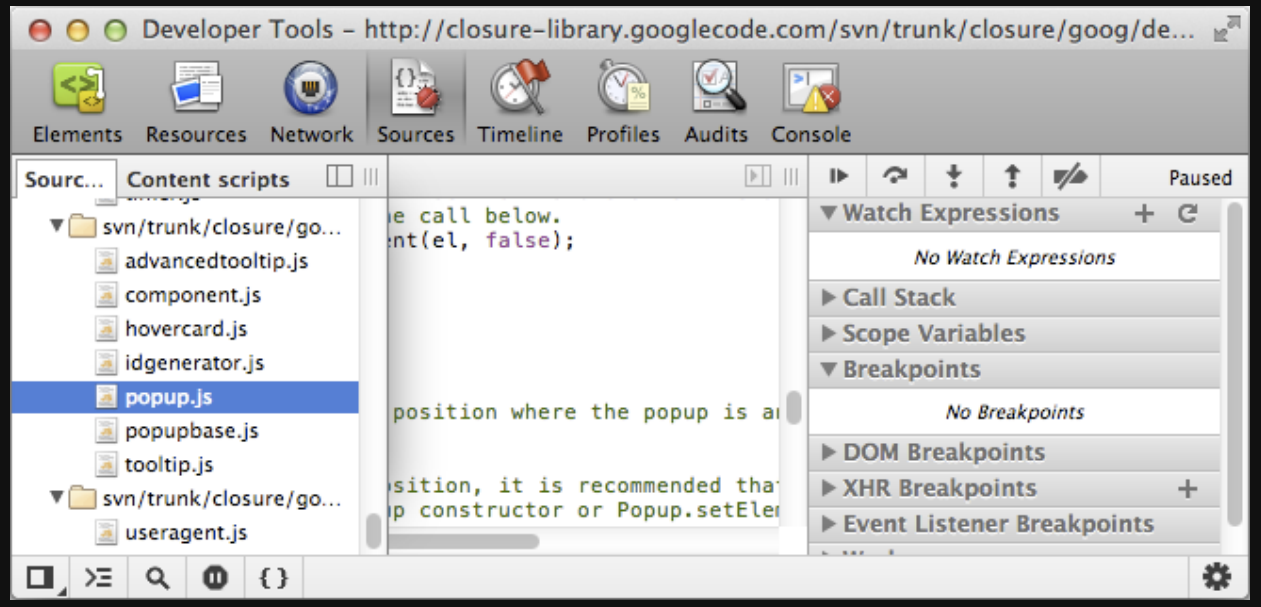
I mean...


He's Dead, Jim!
Something caused this webpage to be killed, either because the operating system ran out of memory, or for some other reason. To continue, press Reload or go to another page.
Learn more

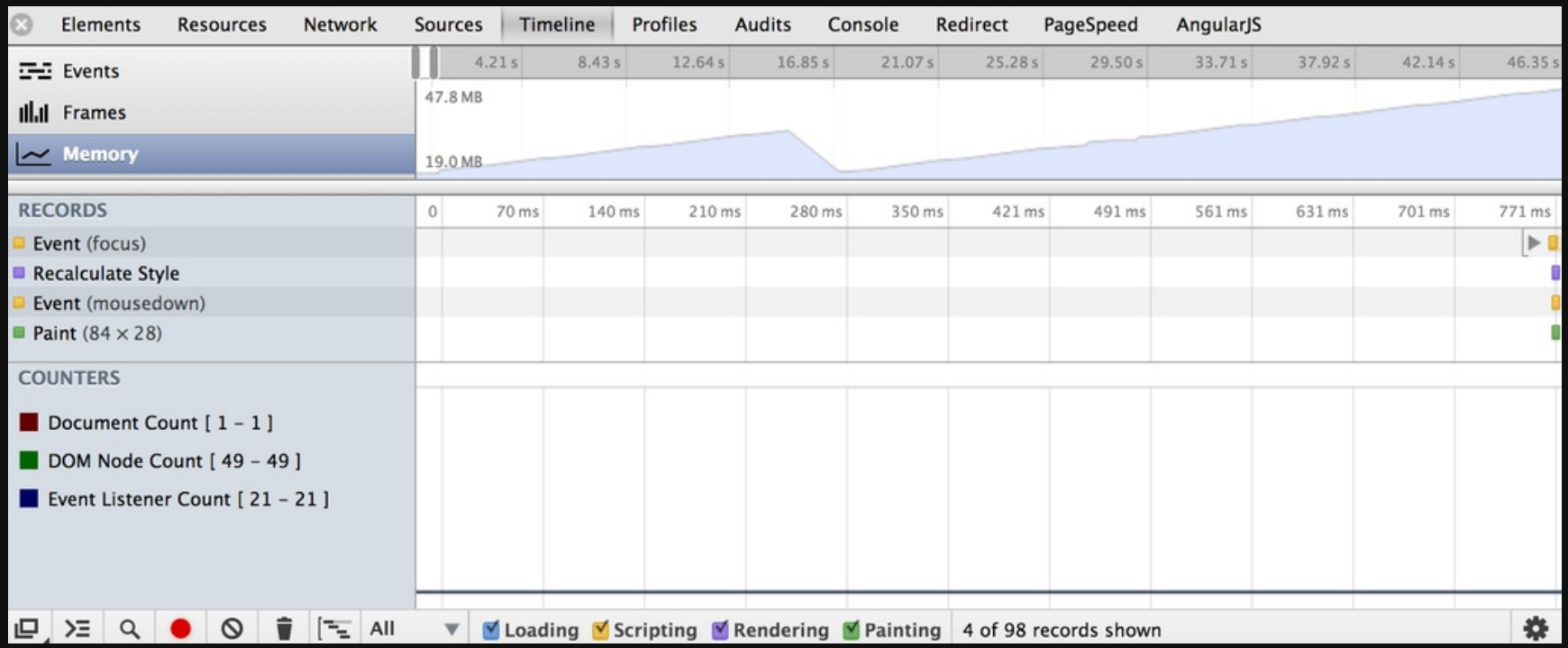
Chrome DevTools

Ctrl+Shift+I
⌥⌘I

https://developers.google.com/chrome-developer-tools/

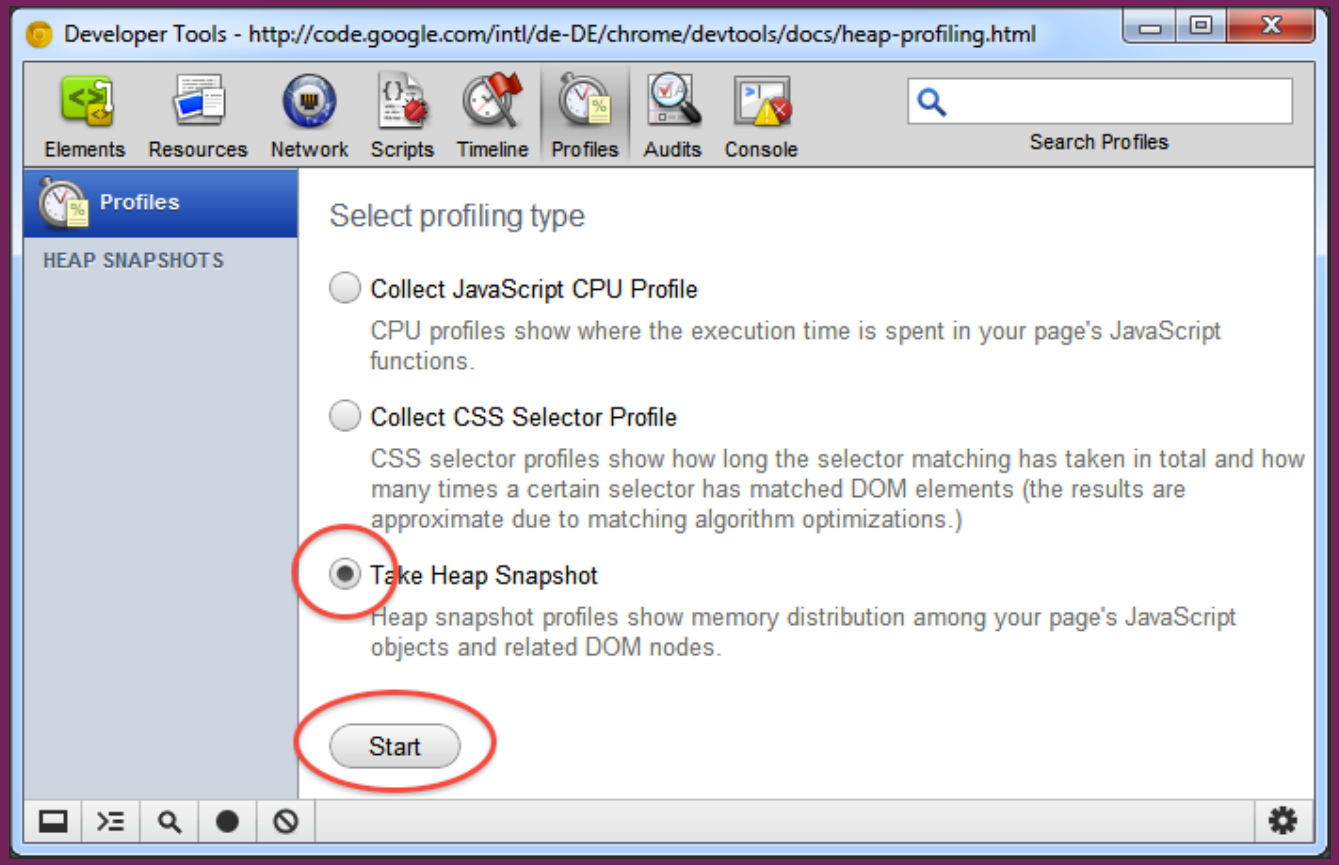


Memory timeline



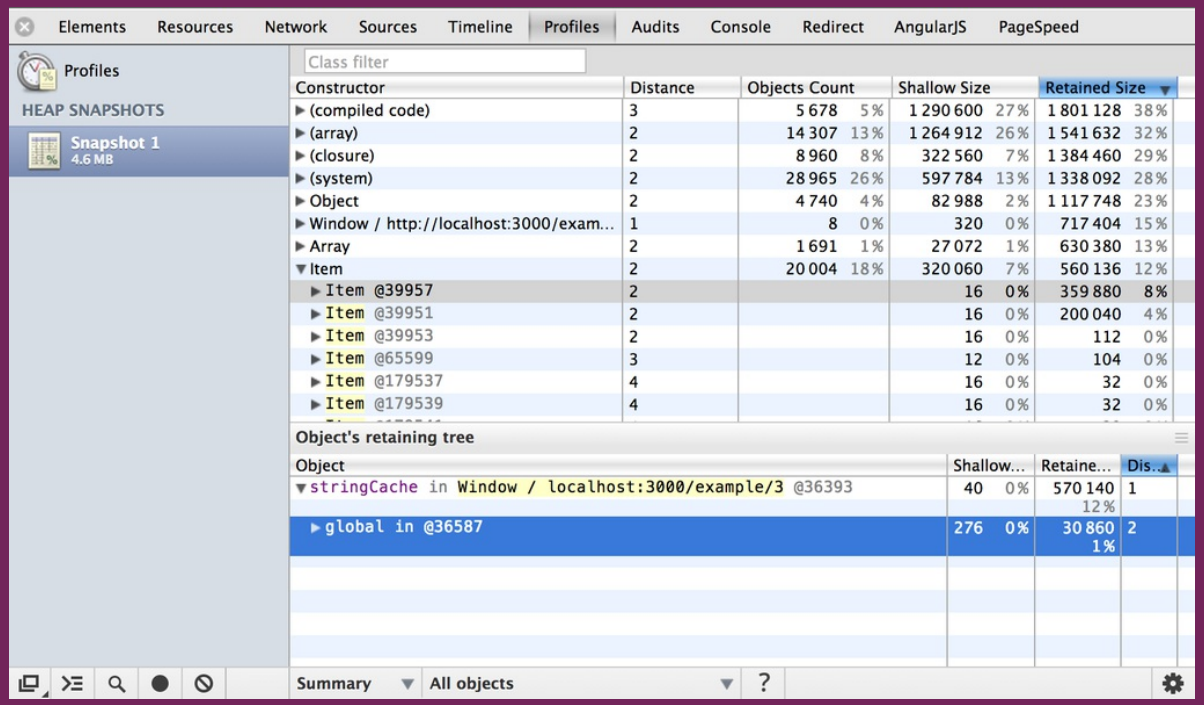
Memory Profiling

Taking snapshots



Reading your results

Summary



EYE-CATCHING THINGS IN THE SUMMARY

Distance:
distance from the GC root.
If almost all the objects of the same type
are at the same distance,
and a few are at a bigger distance,
that's something worth investigating.
Are you leaking the latter ones?

MORE EYE-CATCHING THINGS IN THE SUMMARY

Retaining memory:
the memory used by the objects
AND
the objects they are referencing.
Use it to know where are you
using most of the memory.

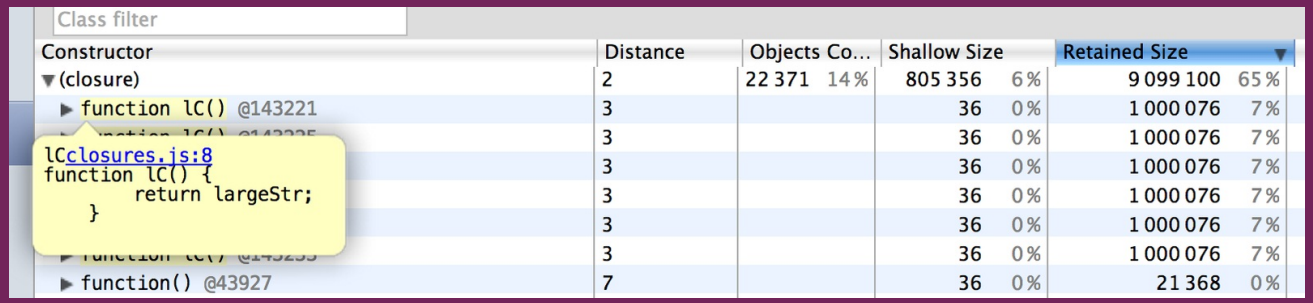
A TIP ABOUT CLOSURES

It helps a lot to name the functions, so you easily distinguish between closures in the snapshot.

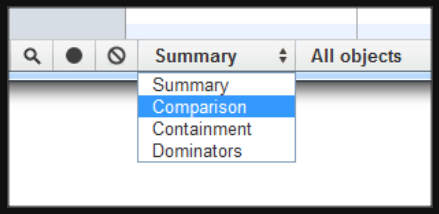
```
function createLargeClosure() {
   var largeStr = new Array(1000000).join('x');

   var lC =  function() { //this IS NOT a named function
      return largeStr;
   };
   return lC;
}

function createLargeClosure() {
   var largeStr = new Array(1000000).join('x');
   var lC = function lC() { //this IS a named function
      return largeStr;
   };
   return lC;
}
```

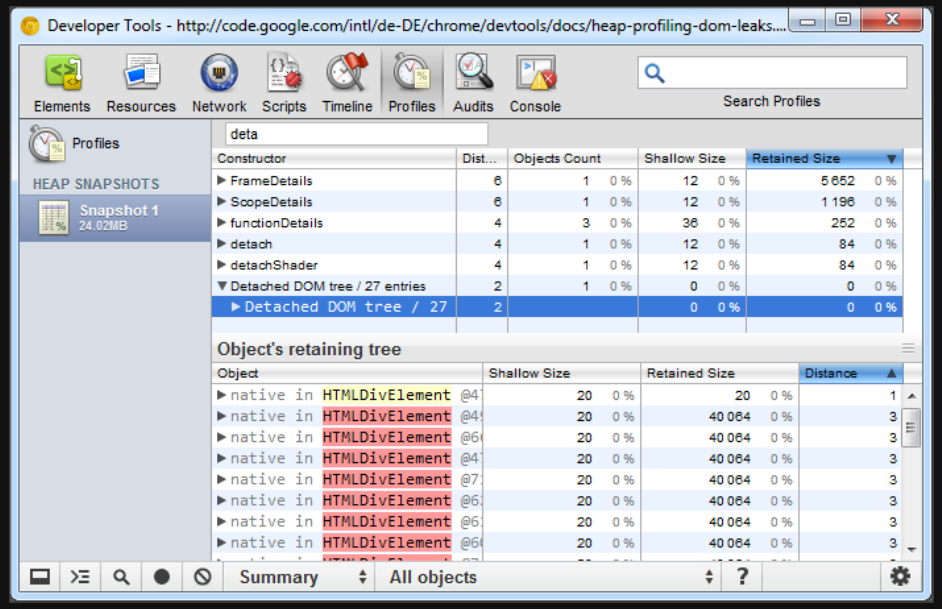


Switching between snapshots views



Summary: groups by constructor name
Comparison: compares two snapshots
Containment: bird's eye view of the object structure
Dominators: useful to find accumulation points

Understanding node colors

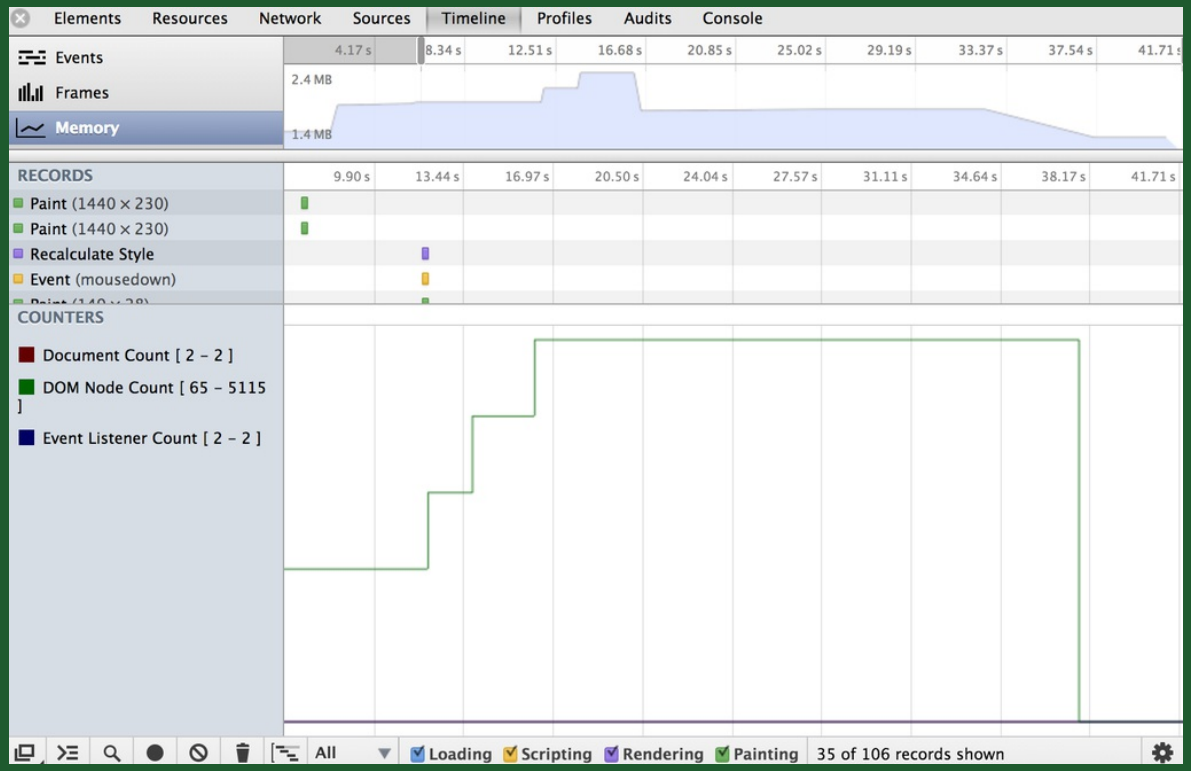


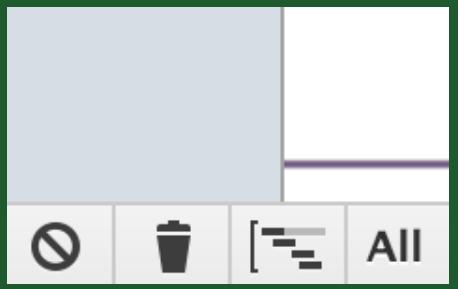
Yellow: object has a JavaScript reference on it

Red: detached node. Referenced from one with yellow background

You can force GC from Chrome DevTools

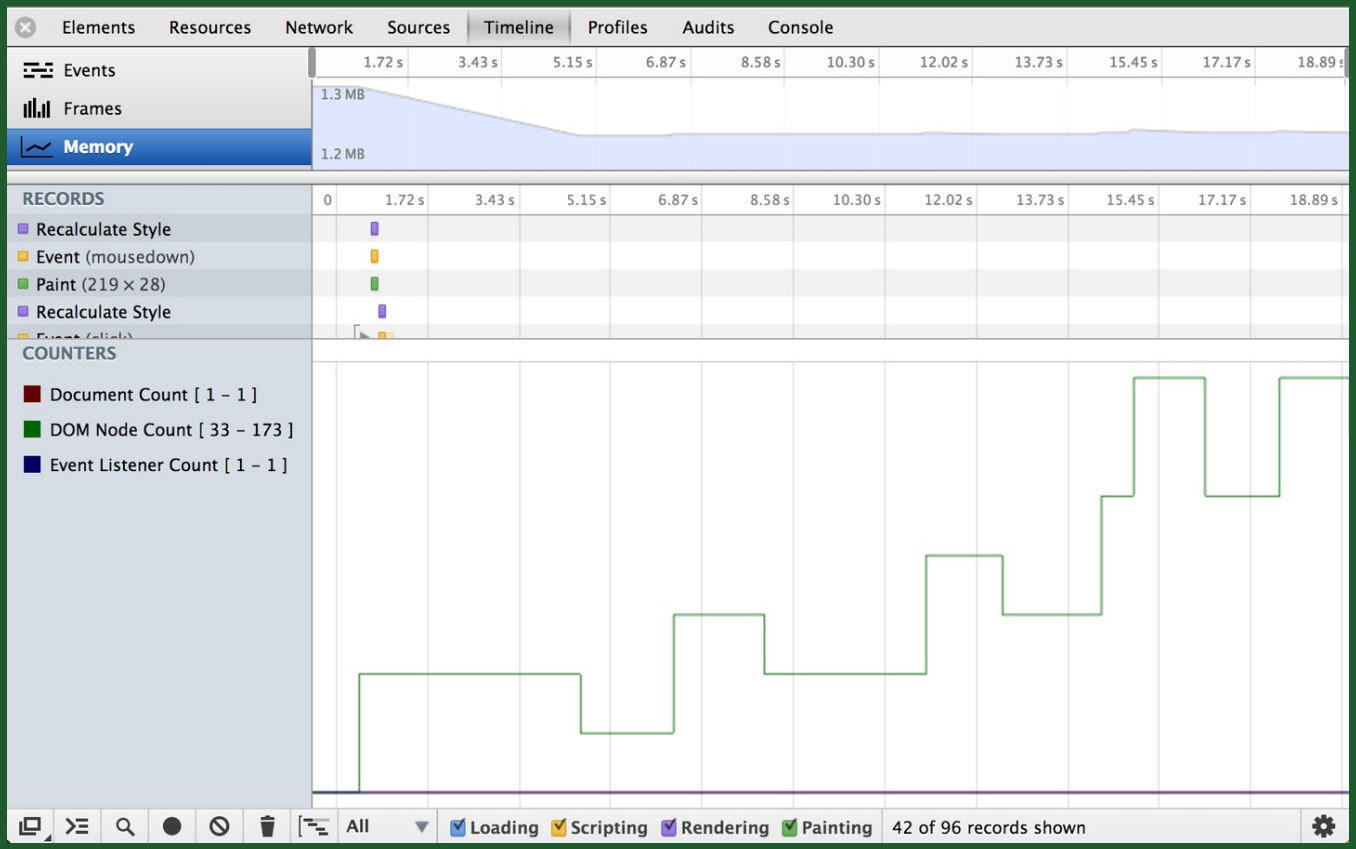
When taking a Heap Snapshot, it is automatically forced.
In Timeline, it can be very convenient to force a GC.





Memory leak pattern

Some nodes are not being collected:



The 3 snapshot technique

Rationale

Your long running application is in an stationary state.

$$\frac{\Delta\, Memory}{\Delta\, Time} = 0$$

Memory oscillates around a constant value.

(or has a constant, controlled, expected and justified growth).

What do we expect?

New objects to be constantly and consistently collected.

Let's say we start from a steady state:

Checkpoint #1

We do some stuff

Checkpoint #2

We repeat the same stuff

Checkpoint #3

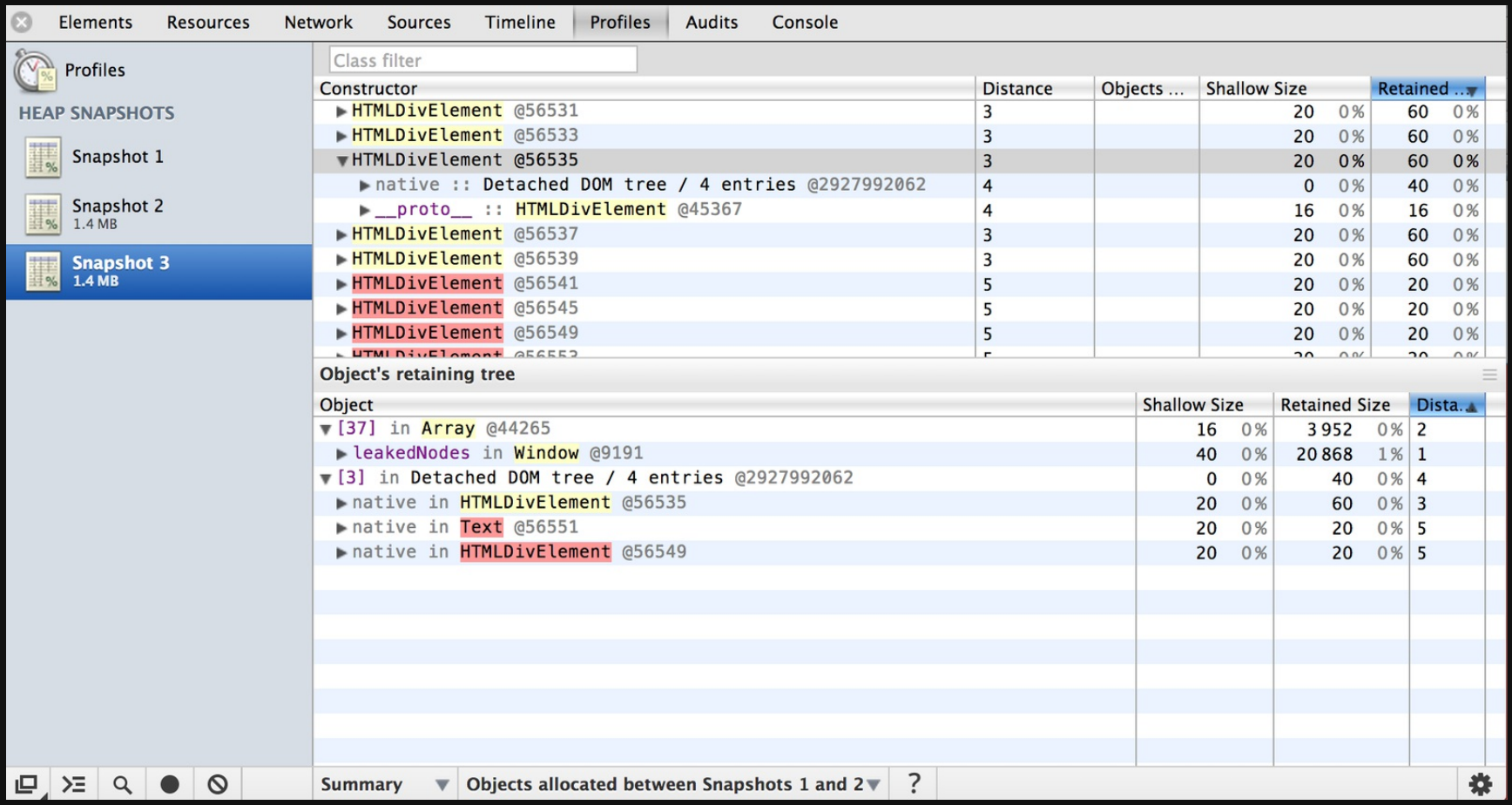
Again, what should we expect?

All new memory used between Checkpoint #1 and Checkpoint #2 has been collected.

New memory used between Checkpoint #2 and Checkpoint #3 may still be in use in Checkpoint #3

The steps

- Open DevTools
- Take a heap snapshot #1
- Perform suspicious actions
- Take a heap snapshot #2
- Perform same actions again
- Take a third heap snapshot #3
- Select this snapshot, and select "Objects allocated between Snapshots 1 and 2"


Elements
Resources
Network
Sources
Timeline
Profiles
Audits
Console
Profiles
HEAP SNAPSHOTS
Snapshot 1
Snapshot 2
1.4 MB
Snapshot 3
1.4 MB
Class filter
Constructor
Distance
Objects ...
Shallow Size
Retained ...
HTMLDivElement @56531 3 20 0% 60 0%
HTMLDivElement @56533 3 20 0% 60 0%
HTMLDivElement @56535 3 20 0% 60 0%
native :: Detached DOM tree / 4 entries @2927992062 4 0 0% 40 0%
__proto__ :: HTMLDivElement @45367 4 16 0% 16 0%
HTMLDivElement @56537 3 20 0% 60 0%
HTMLDivElement @56539 3 20 0% 60 0%
HTMLDivElement @56541 5 20 0% 20 0%
HTMLDivElement @56545 5 20 0% 20 0%
HTMLDivElement @56549 5 20 0% 20 0%
Object's retaining tree
Object
Shallow Size
Retained Size
Dista...
[37] in Array @44265 16 0% 3 952 0% 2
leakedNodes in Window @9191 40 0% 20 868 1% 1
[3] in Detached DOM tree / 4 entries @2927992062 0 0% 40 0% 4
native in HTMLDivElement @56535 20 0% 60 0% 3
native in Text @56551 20 0% 20 0% 5
native in HTMLDivElement @56549 20 0% 20 0% 5
Summary
Objects allocated between Snapshots 1 and 2
?

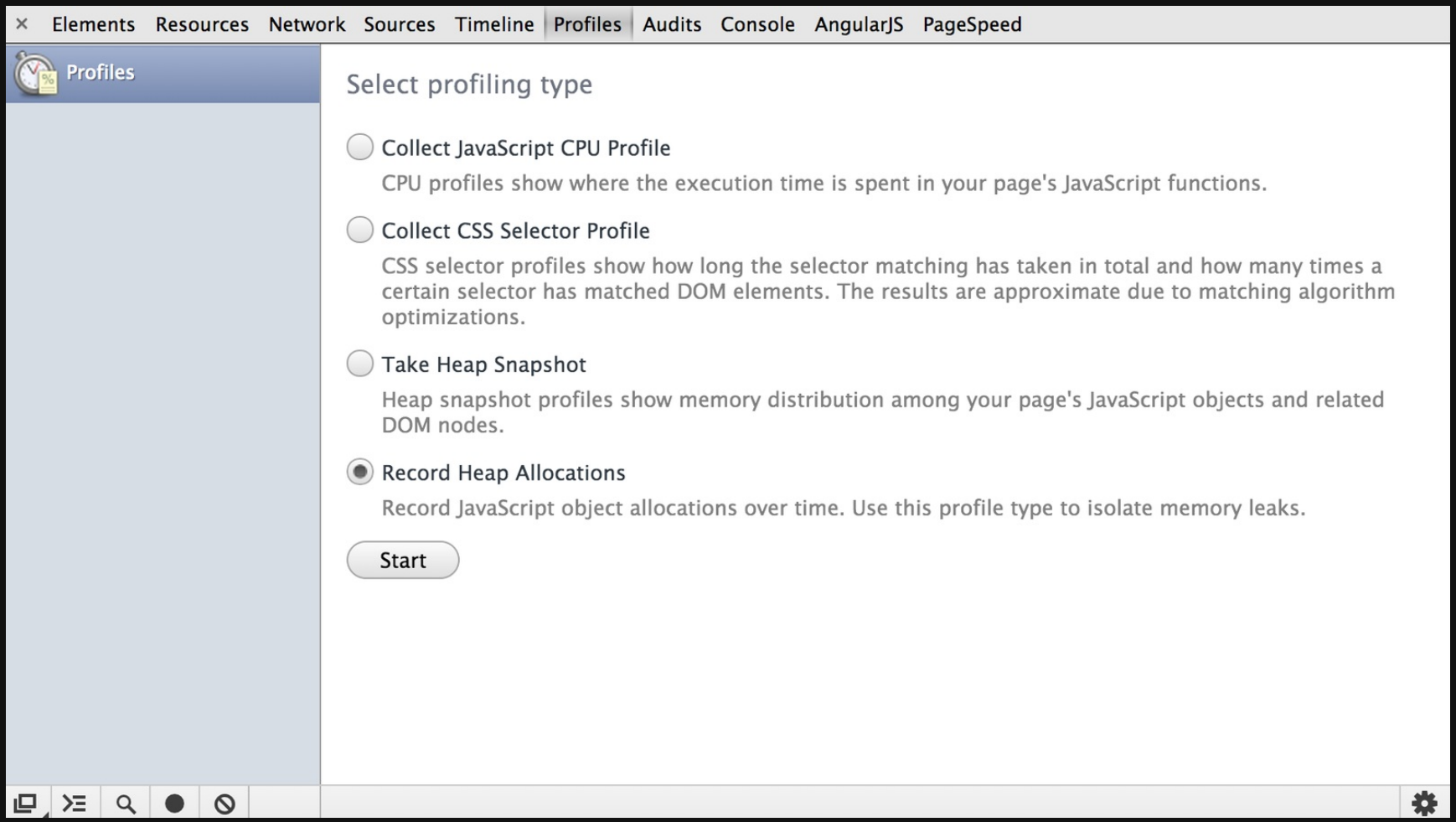
The 3 snapshot technique evolved

Simpler & more powerful but...

Do you have Chrome Canary installed?

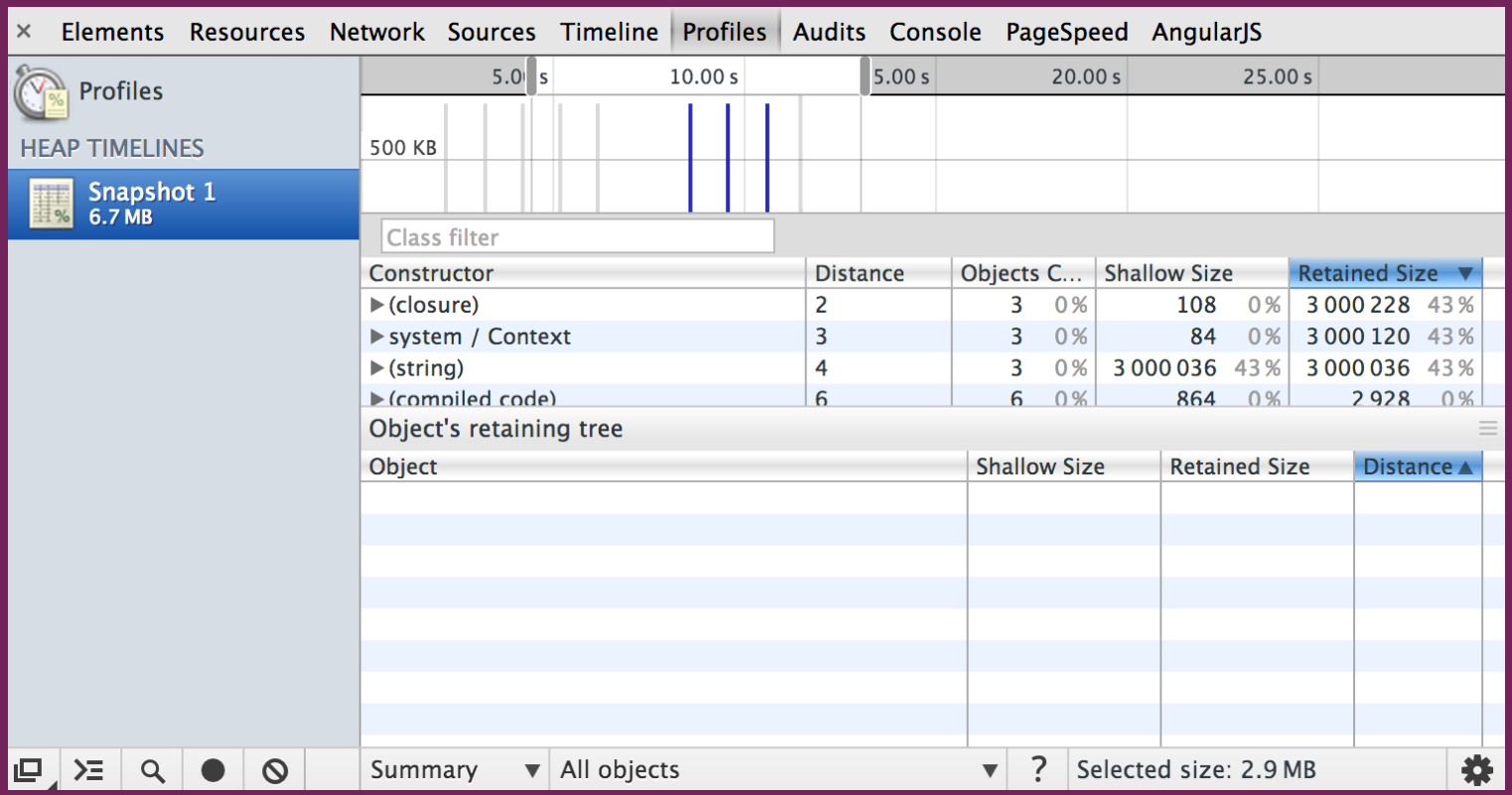
Brand new feature:

Record Heap Allocations


Elements
Resources
Network
Sources
Timeline
Profiles
Audits
Console
AngularJS
PageSpeed
Profiles
Select profiling type
Collect JavaScript CPU Profile
CPU profiles show where the execution time is spent in your page's JavaScript functions.
Collect CSS Selector Profile
CSS selector profiles show how long the selector matching has taken in total and how many times a certain selector has matched DOM elements. The results are approximate due to matching algorithm optimizations.
Take Heap Snapshot
Heap snapshot profiles show memory distribution among your page's JavaScript objects and related DOM nodes.
Record Heap Allocations
Record JavaScript object allocations over time. Use this profile type to isolate memory leaks.
Start

Blue bars : memory allocations.
Taller equals more memory.

Grey bars : deallocated



Let's play!

You can get the code from:

https://github.com/gonzaloruizdevilla/debuggingmemory.git

Or you can use:

http://goo.gl/4SK53

Thank you!

gonzalo.ruizdevilla@adesis.com
@gruizdevilla

(btw, we are hiring!)