Berlin Brown – Technical Blog Book

Software Engineering Cookbook

Java Technology

Java Performance

Overview and JVM Languages

One of the exciting trends to recently emerge from the Java community is the concept of the JVM language. These technologies are all that you would expect them to be. They are implementations of languages that run on the Java Virtual Machine. Some are newly created and some are based on existing, more mature languages. JRuby, Jython are two JVM languages based on CRuby and CPython. Groovy, Scala, Clojure are three completely new JVM languages that were created to add new language features that weren't supported by the core Java language. Some must or can be compiled. Some run without compilation. You can easily compile Scala code to Java bytecode. Clojure also allows this feature (ahead of time compilation). Clojure and JRuby code can also run without having be explicitly compiled. You can interact with the language. In most cases and with most JVM languages, you have full access to existing libraries that were written in pure Java. And normally you can access existing JVM language code from Java (known as Java interoperability). In most cases, it is easier to access Java calls from the JVM language than it is to call the language code from Java. It really depends on the language. In the snippet below, there is a call to the System static method, 'nanoTime'. Simply invoke the system like you would from pure Java.

```
require 'java'
def helloWorldBaseline
  # Run the application
  start1 = java.lang.System.nanoTime()
  arr = Array.new
  (1..1000000).each {
    res = 2.2 * 4.1 * 5.4
  }
  end1 = java.lang.System.nanoTime()
  diff = (end1 - start1) * 1e-6
  puts "Elapsed Time: #{diff} ms"
end
```

Dynamic Compilation - "Clojure is a compiled language, so one might wonder when you have to run the compiler. You don't. Anything you enter into the REPL or load using load-file is automatically compiled to JVM bytecode on the fly. Compiling ahead-of-time is also possible, but not required" -- Clojure Docs on being Dynamic.

On Performance

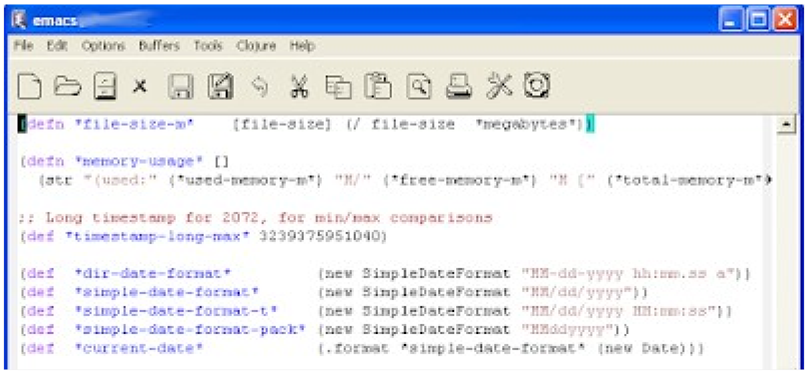
"If you want to measure something, then don't measure other shit." -- Zed Shaw [3]

For the more popular JVM languages, like JRuby and Jython, there isn't much of a difference between running code in their respective C implementations. JRuby is especially well known for being very portable. With JRuby release 1.3.1, JRuby is compatible with CRuby 1.8.6. Jython 2.5.0 was released last month and brings the level of compatibility to CPython versions 2.5. Django and other popular CPython based frameworks are able to work with Jython. You may be wondering, if the Java Virtual Machine language is compatible with the C language, native implementation, is there a loss in performance when running on the Java Virtual Machine? Is there a major loss in performance? That is this purpose of this document, how much time does it take for a particular piece of code to run in JVM language? How long does it take to run similar routines using pure Java code? I want to make it clear, you will not find a large scientific benchmark run under clean room like conditions. I want to present a simple set of routines and how long it took to run. How long did the Clojure code run? How long did the Scala code run? Basically, I want to present the code and how long each test ran, but I don't want to claim that anyone language or piece of code is faster or slower based on these tests. You could say that most of the pure Java code ran faster. Most of the time, that is what happened after running these tests. But there is too much confounding in my tests. Like Zed Shaw said, "If you want to measure something, then don't measure other shit." [3] There is a lot of stuff in my tests to not make these an official comparison. There is a lot of confounding. But, here is the code, here is how long it took to run? It be relevant in more common tests like a Project Euler problem. Project Euler is a website that contains math problems intended to be solved with computer programs. In Project Euler problem number one, I write a program in Clojure and then in Java. They both run on the JVM and the same value is returned. What was the execution time for each program? Simple tests, simple results.

When working with JVM languages and possible performance bottlenecks, you want to consider execution time, but you also want to look at the garbage collector and heap memory usage. Garbage collection is an expensive operation. It won't take a minute to run a garbage collect, but it will take cpu cycles away from your application. JVM code runs in a protected environment, the garbage collector provides automatic memory management and normally protects you from improper memory use. And the garbage collector attempts to free up memory that is no longer needed. You can't normally control when the JVM runs garbage collection and certainly don't want force it. But if you monitor your application, you can identify memory leaks or other problems that might cause performance bottlenecks. It will normally be evident where there is a problem. If you see too many garbage collects within a very short period of time and your total available memory is maxed out, you might eventually encounter an out of memory error. In a long running server environment, most of your performance issues might be alleviated if you look at proper heap memory use. Is your code forcing too many garbage collections within a short period of time? Are you creating too many large objects and holding on to them for too long? In performance tuning your application, there are many things to consider. It may not just be improving a particular algorithm. Consider heap memory and object allocation as well. For most of the tests, there are performance stats, memory stats and other garbage collection statistics.

On Clojure

Clojure is a Lisp dialect created in 2007 by Rich Hickey. It recently reached a 1.0 release and has a large following of Java programmers and Common Lisp developers. Since the Dec 2008 release, the Clojure library has received 18,000 downloads (based on Google Code Stats). Clojure is a dynamically (dynamic/strong) typed language, supports lazy sequences, immutable data structures, macros, and functions as first class objects [4]. Clojure is a functional language just like Common Lisp is a functional language. It is a Lisp dialect so that includes the fully parenthesized syntax. Most syntax will include a function or macro call with arguments or no arguments enclosed by a left and right parenthesis.



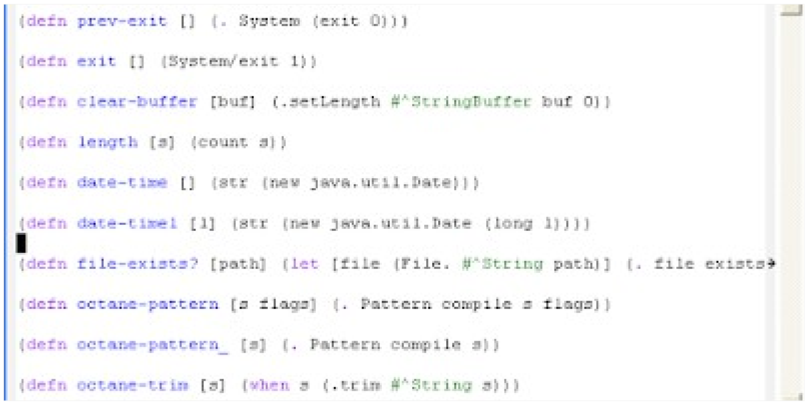
```
(defn *file-size-m*     [file-size] (/ file-size  *megabytes*))

(defn *memory-usage* []
  (str "(used:" (*used-memory-m*) "M/" (*free-memory-m*) "M [" (*total-memory-m*)

;; Long timestamp for 2072, for min/max comparisons
(def *timestamp-long-max* 3239375951040)

(def  *dir-date-format*           (new SimpleDateFormat "MM-dd-yyyy hh:mm.ss a"))
(def  *simple-date-format*        (new SimpleDateFormat "MM/dd/yyyy"))
(def  *simple-date-format-t*      (new SimpleDateFormat "MM/dd/yyyy HH:mm:ss"))
(def  *simple-date-format-pack* (new SimpleDateFormat "MMddyyyy"))
(def  *current-date*              (.format *simple-date-format* (new Date)))
```

Syntax

Most newcomers to a Lisp dialect may get distracted by the parentheses, the symbolic expressions. They can seem daunting if you are more familiar to a language like C++ or Java. But, this actually is one of the major benefits of Lisp. The simple syntax, functional call, args very much resemble how the compiler or parser will interpret the code. Simple is good. Simple is fast, especially to the machine. It can also benefit the developer because you aren't overburdened with a bunch of syntax to memorize.

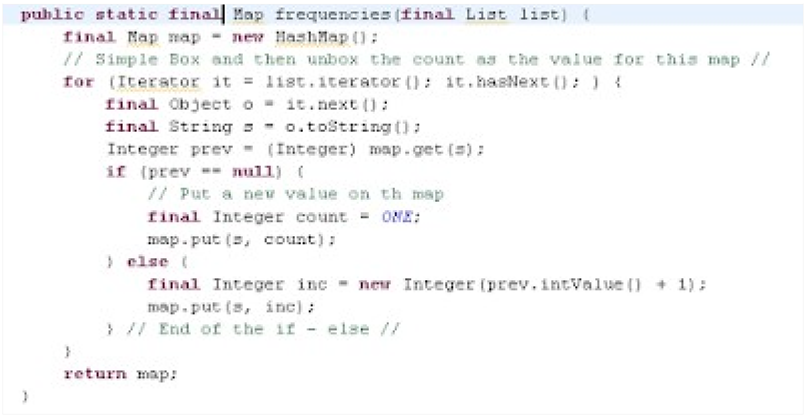
It also really helps to have a great editor like Emacs. Emacs is built with its own Lisp dialect, Emacs Lisp. So, Clojure syntax is not too foreign to Emacs. You will need to download the Clojure Emacs Mode and you want to add Slime integration.

Here is a snippet of Clojure code. Just focus on the left parenthesis and the token adjacent to the character. The token, function or macro call and the left parenthesis.



```clojure
(defn prev-exit [] (. System (exit 0)))

(defn exit [] (System/exit 1))

(defn clear-buffer [buf] (.setLength #^StringBuffer buf 0))

(defn length [s] (count s))

(defn date-time [] (str (new java.util.Date)))

(defn date-time1 [l] (str (new java.util.Date (long l))))

(defn file-exists? [path] (let [file (File. #^String path)] (. file exists
(defn octane-pattern [s flags] (. Pattern compile s flags))

(defn octane-pattern_ [s] (. Pattern compile s))

(defn octane-trim [s] (when s (.trim #^String s)))
```

Lisp does not normally get more complicated than the parenthesis tokens and defining function bodies. A function/macro call and arguments. The arguments are normally separated by a variable number of spaces and may include calling another routine. That is the essence of functional programming. You have functions that return some value and can call other functions. You don't have to worry about Object creation syntax, for loop syntax, anonymous inner classes or things that you might encounter with Java. Here is some sample Java code. Look at all the tokens that part of the language.



```java
public static final Map frequencies(final List list) {
    final Map map = new HashMap();
    // Simple Box and then unbox the count as the value for this map //
    for (Iterator it = list.iterator(); it.hasNext(); ) {
        final Object o = it.next();
        final String s = o.toString();
        Integer prev = (Integer) map.get(s);
        if (prev == null) {
            // Put a new value on th map
            final Integer count = ONE;
            map.put(s, count);
        } else {
            final Integer inc = new Integer(prev.intValue() + 1);
            map.put(s, inc);
        } // End of the if - else //
    }
    return map;
}
```

Note: With this document, I tried not to position one language as better or worse than the other. Each technology that I mention has advantages and disadvantages for writing software. Each tool may give the developer productivity gains and some developers may never get used to changing to a new syntax, never truly realizing some of the intended benefits that the language has to offer. You will have to evaluate these languages (or not) on your own and make. I merely try to point out some of the similarities and some of the differences.

Lisp certainly suffers from that old adage, "easy to learn, may take a lifetime to master". It is easy to write Lisp code, it may take time to write readable, solid, idiomatic Lisp code. I am sure I will get many comments on how to write more idiomatic Clojure code even for these rudimentary examples. Luckily, Clojure has many of the functions that you will encounter with most other Lisp dialects. In Java or C++, you may be accustomed to the 'for loop' syntax. Here is the Clojure macro for building a list of elements.

```clojure
;; Build a collection of random numbers
(for [_ (range 100) ] (.nextInt random))
```

For Clojure API Documentation: "List comprehension. Takes a vector of one or more binding-form/collection-expr pairs, each followed by zero or more modifiers, and yields a lazy sequence of evaluations of expr. Collections are iterated in a nested fashion, rightmost fastest, and nested coll-exprs can refer to bindings created in prior binding-forms."

```
;; Macro definition for 'for'
(defmacro for
  [seq-exprs body-expr]
  (assert-args for
     (vector? seq-exprs) "a vector for its binding"
     (even? (count seq-exprs)) "an even number of forms in binding vector")
  (let [to-groups (fn [seq-exprs]
                        (reduce (fn [groups [k v]]
                                 (if (keyword? k)
                                   (conj (pop groups) (conj (peek groups) [k v]))
                                   (conj groups [k v])))
                                 [] (partition 2 seq-exprs)))
         err (fn [& msg] (throw (IllegalArgumentException. (apply str msg))))
         emit-bind (fn emit-bind [[[bind expr & mod-pairs]
                                    & [[_ next-expr] :as next-groups]]]
                        (let [giter (gensym "iter__")
                              gxs (gensym "s__")
                              do-mod (fn do-mod [[[k v :as pair] & etc]]
                                        (cond
                                         (= k :let) `(let ~v ~(do-mod etc))
                                         (= k :while) `(when ~v ~(do-mod etc))
                                         (= k :when) `(if ~v
                                                        ~(do-mod etc)
                                                        (recur (rest ~gxs)))
                                         (keyword? k) (err "Invalid 'for' keyword "
k)

                                         next-groups
                                         `(let [iterys# ~(emit-bind next-groups)
                                                fs# (seq (iterys# ~next-expr))]
                                            (if fs#
                                              (concat fs# (~giter (rest ~gxs)))
                                              (recur (rest ~gxs))))
                                      :else `(cons ~body-expr
                                                   (~giter (rest ~gxs)))))]
                          `(fn ~giter [~gxs]
                           (lazy-seq
                            (loop [~gxs ~gxs]
                              (when-first [~bind ~gxs]
                                         ~(do-mod mod-pairs)))))))]
    `(let [iter# ~(emit-bind (to-groups seq-exprs))]
       (iter# ~(second seq-exprs)))))
;; END OF MACRO (code from core.clj, clojure source);;
```

OK, that was a little daunting, but it is good to know the code for all of the Clojure source is freely available. This a yet another added benefit that you may not capitalize on, but all of the Clojure source is highly readable and lightweight. There are only 24k lines of Java code in the clojure.lang package. And only 7k lines of Clojure source in the core libray. I routinely make changes to the Clojure Java code to add trace logic. It took only 10 minutes to figure out where I needed to place my logic and then compile with my own Ant script. Imagine making similar changes to the Java API?

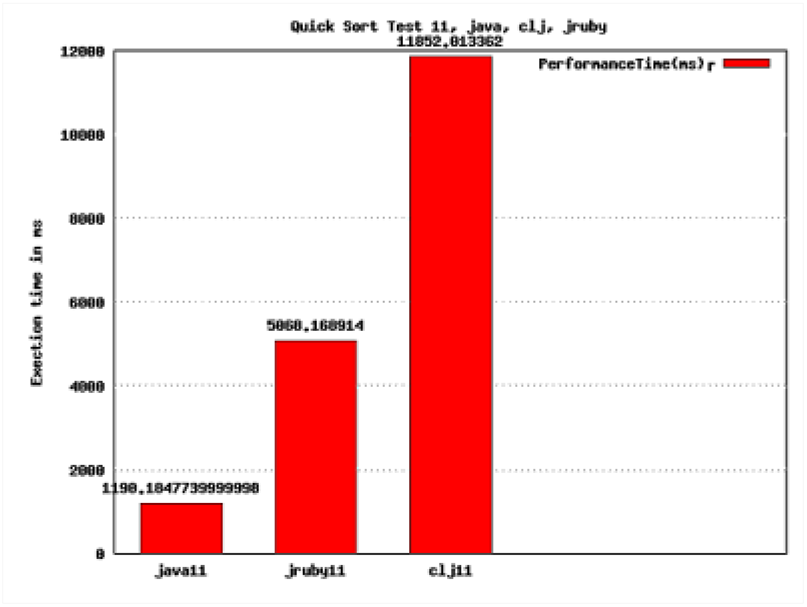
More on the Clojure, Java source for the performance tests

I used a simple format for the performance tests that I will present later in the document. I wrote or used existing code from a third-party and then ran the code. There may be many differences between the style of code. And the Clojure code may not be idiomatic. But it should be close. Here is the source for the Quick Sort, Clojure and Java:

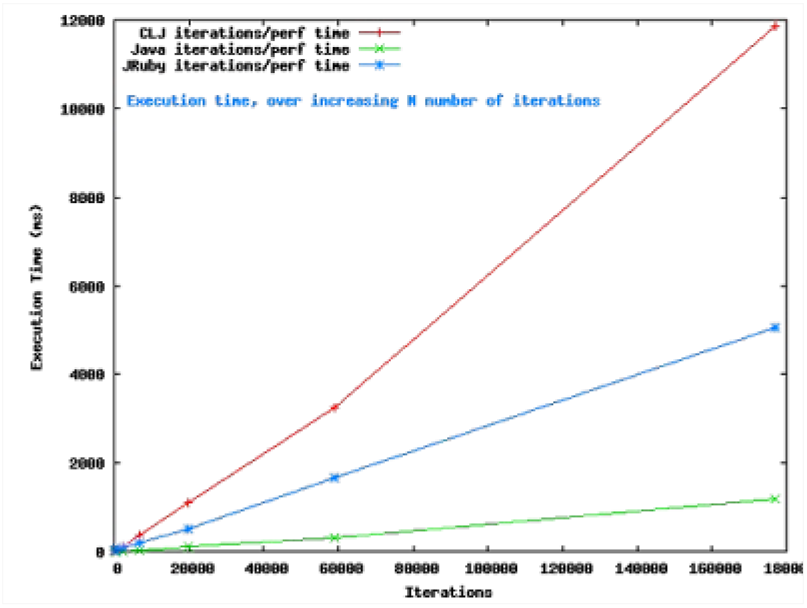
```
;; Two versions of the Clojure Quick Sort:
(defn qsort
  "Quick sort from rosetta code:
 http://rosettacode.org/wiki/Quicksort#Clojure"
  [[pvt & rs]]
  (if pvt
    `(~@(qsort (filter #(<  % pvt) rs))
      ~pvt
      ~@(qsort (filter #(>= % pvt) rs)))))
(defn qsort-2 [[pivot & xs]]
  (when pivot
    (let [smaller #(< % pivot)]
      (lazy-cat (qsort-2 (filter smaller xs))
                [pivot]
                (qsort-2 (remove smaller xs))))))
(dotimes [x 4]
      (println "i: " (int (Math/pow 10.0 x)))
    (time (count (qsort (for [_ (range (int (Math/pow 10.0 x))) ]
                           (.nextInt random))))))



// JAVA VERSION OF QUICK SORT
    public static final List quicksort(final List arr) {
        if (arr.size() <= 1) {
            return arr;
        }
        Integer pivot = (Integer) arr.get(0); //This pivot can change to get faster
results

        List less = new LinkedList();
        List pivotList = new LinkedList();
        List more = new LinkedList();

        // Partition
        for (Iterator it = arr.iterator(); it.hasNext();) {
            Integer i = (Integer) it.next();
            if (i.compareTo(pivot) < 0) {
                less.add(i);
            } else if (i.compareTo(pivot) > 0) {
                more.add(i);
            } else {
                pivotList.add(i);
            } // End of the if - else //

        } // End of the for //

        // Recursively sort sublists
        less = quicksort(less);
        more = quicksort(more);

        // Concatenate results
        less.addAll(pivotList);
        less.addAll(more);
        return less;
    }
// END OF EXAMPLE
```

Performance of Test 11, Sorting

How did the various languages perform?



```
JavaTest    PerformanceTime(ms)
#####       ###########
java11      1190.1847739999998
jruby11     5068.168914
clj11       11852.013362
```



JRuby and Scala Source, Scala Performance for the Simple Quick Sort

On Scala

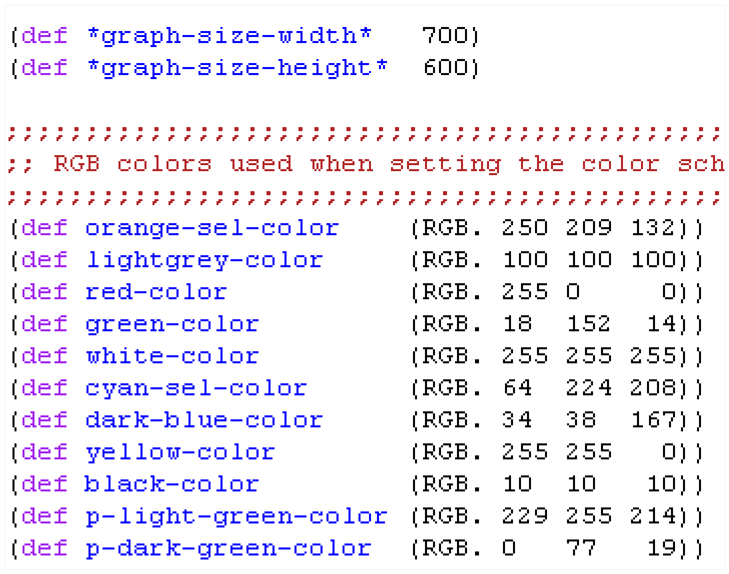
Scala was created in 2001 and is currently at version 2.7.5 as of 6/3/2009. Scala is a stark contrast to Clojure. David Pollak had this to say about Scala's type system, "Put another way, Scala is statically typed with optional duck typing... the opposite of Objective-C et. al. with their duck typing with optional static typing." Clojure on the other hand is dynamic in the sense that you don't have to explicitly define a type whenever you need to use a particular object, but you can define the type specification using type hints.

```
;; Notice, second version uses the String type hint
(defn str-split-refl [s regex]
      (vec (. s split regex)))
(defn str-split-fast [#^String s regex]
      (vec (. s split regex)))
```

The above example presents two versions of a string split method, in Clojure. The second version use the String type hint to help the compiler determine the type for future calls in that function. On Scala's type system, "Scala is equipped with an expressive type system that enforces statically that abstractions are used in a safe and coherent manner" [6]

When would you use a JVM language? (Scala, Clojure, or just Java)

Why even investigate a JVM language? What is the point? A programming language is like any other tool or library used to create and interact with other software or hardware components? If you work in a web or J2EE environment, you might write SQL code, define CSS scripts, write Javascript code, write HTML/XHTML. It isn't uncommon for web application developers to write Java, sql, css, javascript or HTML code. Sometimes all within the same day, sometimes during different phases of a project. That doesn't take into account the number libraries that you must normally learn, understand and work with. So, NOT learning a new JVM programming language just on the premise that it is something different, isn't a valid reason for not using it. People have asked me, would you use Clojure, Java, or Scala? I have used all three for small GUI projects. I have used Scala for the the backend API where I don't need to make small changes. I used Clojure because of the dynamic nature of the language. I can make many, quick incremental changes without having any major impact on the Scala backend API. Take the Java api for example. Most of the core library is set in stone. The java.lang.String class hasn't changed much in over a decade. I see Scala being used for those type of rigid API requirements. This doesn't mean that Clojure couldn't be used for this purpose, it just means that is how I have used Scala and it just seemed to fit because of how easy it is to call Scala from Java (Java interoperability), also because of the nature of Scala's types. Here is just one example on how I used Clojure. The code snippet below contains valid Clojure code used to develop a small GUI application. If you just look at it without understanding the syntax, the code below almost looks like a general purposed configuration file. Here I can easily modify the layout of my GUI window, buttons without ever really getting any complex language details. I am just looking at the data required to change my layout.



```
(def *graph-size-width*   700)
(def *graph-size-height*  600)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; RGB colors used when setting the color sch
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(def orange-sel-color     (RGB. 250 209 132))
(def lightgrey-color      (RGB. 100 100 100))
(def red-color            (RGB. 255 0     0))
(def green-color          (RGB. 18  152  14))
(def white-color          (RGB. 255 255 255))
(def cyan-sel-color       (RGB. 64  224 208))
(def dark-blue-color      (RGB. 34  38  167))
(def yellow-color         (RGB. 255 255   0))
(def black-color          (RGB. 10  10   10))
(def p-light-green-color  (RGB. 229 255 214))
(def p-dark-green-color   (RGB. 0   77   19))
```

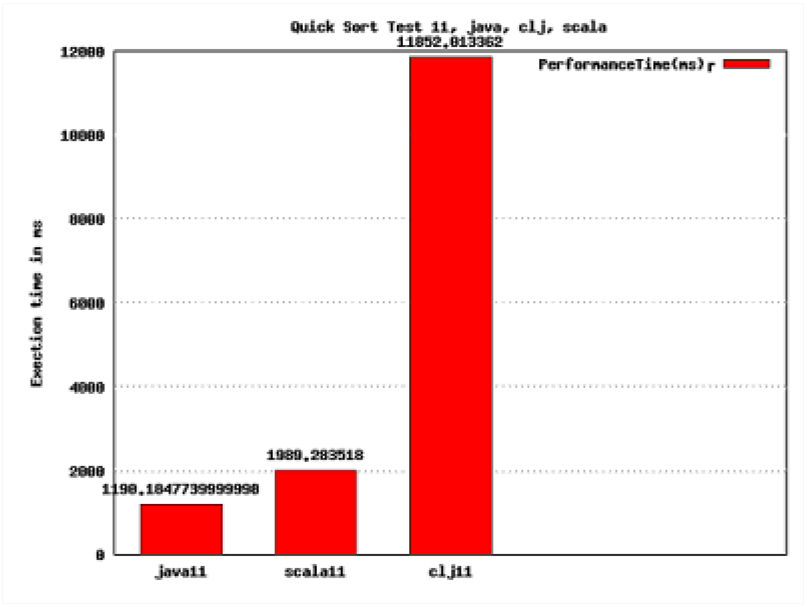
A blogger from creativekarma.com provided a list of some of Scala's features:

```
    * Conventional class types,
    * Value class types,
    * Nonnullable types,
    * Monad types,
    * Trait types,
    * Singleton object types (procedural modules, utility classes, etc.),
    * Compound types,
    * Functional types,
    * Case classes,
    * Path-dependent types,
    * Anonymous types,
    * Self types,
    * Type aliases,
    * Generic types,
    * Covariant generic types,
    * Contravariant generic types,
    * Bounded generic types,
    * Abstract types,
    * Existential types,
    * Implicit types,
    * Augmented types,
    * View bounded types, and
    * Structural types which allow a form of duck typing when all else fails." [8]
```

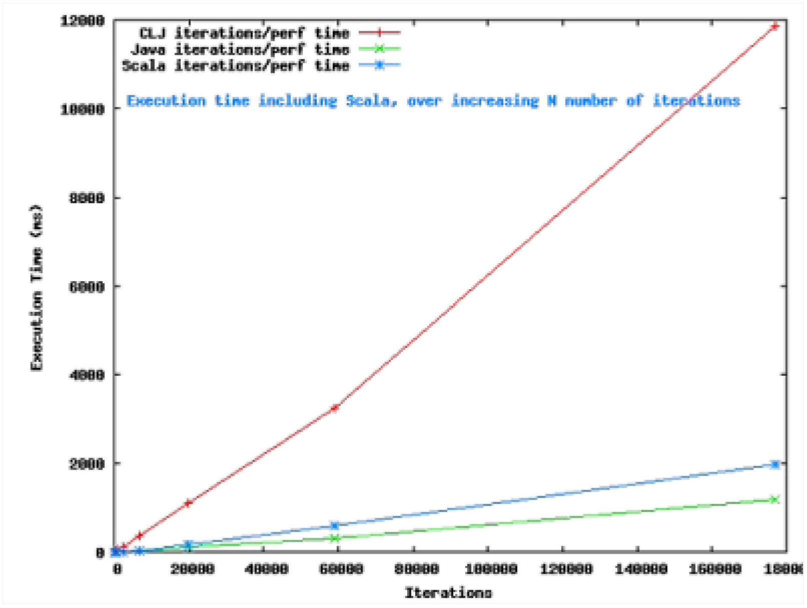
```
// Here is an Example Scala Quick Sort:
 // http://en.literateprograms.org/Quicksort_(Scala)
 // From scala home
  def sort[A <% Ordered[A]](xs: List[A]):List[A] = {
    if (xs.isEmpty || xs.tail.isEmpty) xs
    else {
      val pivot = xs( xs.length / 2)
      // list partition
      // initialize boxes
      var lows: List[A] = Nil
      var mids: List[A] = Nil
      var highs: List[A] = Nil
      for( val item <- xs) {
        // classify item
        if ( item == pivot) mids = item :: mids
        else if (item < pivot) lows = item :: lows
        else highs = item :: highs
      }
      // return sorted list appending chunks
      sort(lows) ::: mids ::: sort(highs)
    }
  }
  // Running the sort and time
  def run(n:Int) = {
    val start1 = java.lang.System.nanoTime()
    //val l = repeat(n)(random.nextInt)
    val l = initialize(n)
    val l2 = sort(l)
    val d = l2.length
    val z = d + 1
    val end1 = java.lang.System.nanoTime()
    val diff = (end1 - start1) * 1e-6
    println("Elapsed Time: " + diff + " ms " + z)
  }
 // End of scala example
```

JRuby Quick Sort

```
# JRuby Sort Example:
class Array
  def quick_sort
    return self if length <= 1
    pivot = self[length / 2]
    return (find_all { |i| i <  pivot }).quick_sort +
           (find_all { |i| i == pivot }) +
           (find_all { |i| i >  pivot }).quick_sort
  end
end
def runTest1a(n)
  # Run the application
  start1 = java.lang.System.nanoTime()
  arr = Array.new
  (1..n).each {
    arr << rand(100000)
  }
  res = arr.quick_sort
  end1 = java.lang.System.nanoTime()
  diff = (end1 - start1) * 1e-6
  puts "Elapsed Time: #{diff} ms"
end
# End of Example
```



```
Test      PerformanceTime(ms)
#####     ###########
java11    1190.1847739999998
scala11   1989.283518
clj11     11852.013362
```

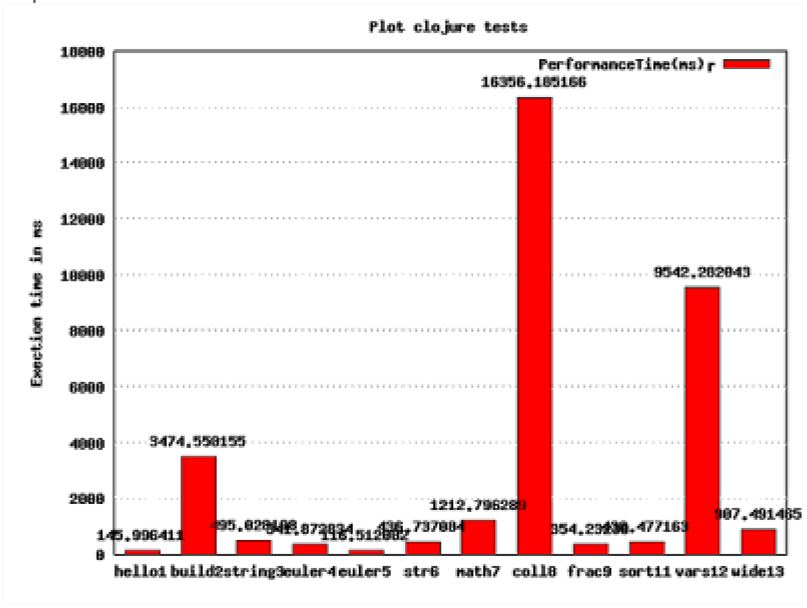


The line graph is a combination of Java, Scala, and Clojure execution times with an increasing number of random elements. Here are the results for the Clojure quick sort:

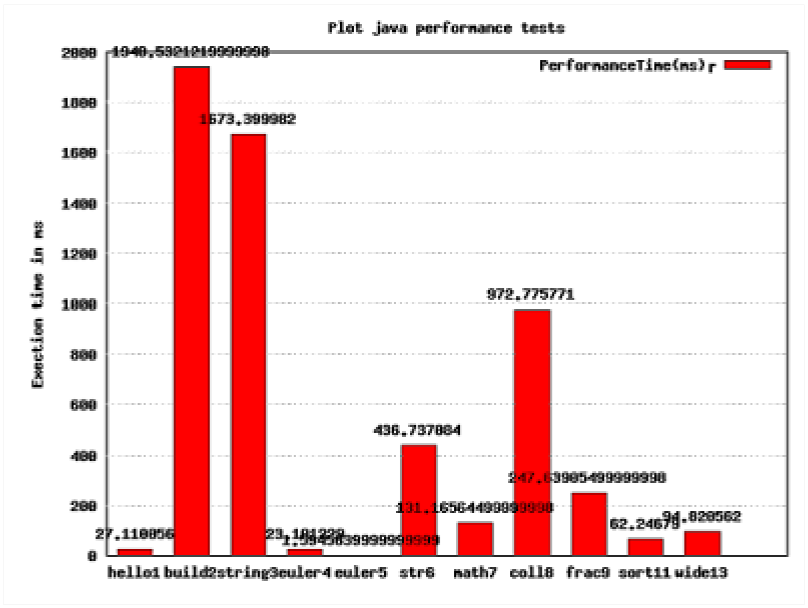
```
iterat    exectime(ms)
#####     #########
81        8.019583
243       16.82068
729       69.767482
2187      121.894408
```

```
6561      370.054832
19683     1096.553958
59049     3245.377605
177147    11852.013362
```

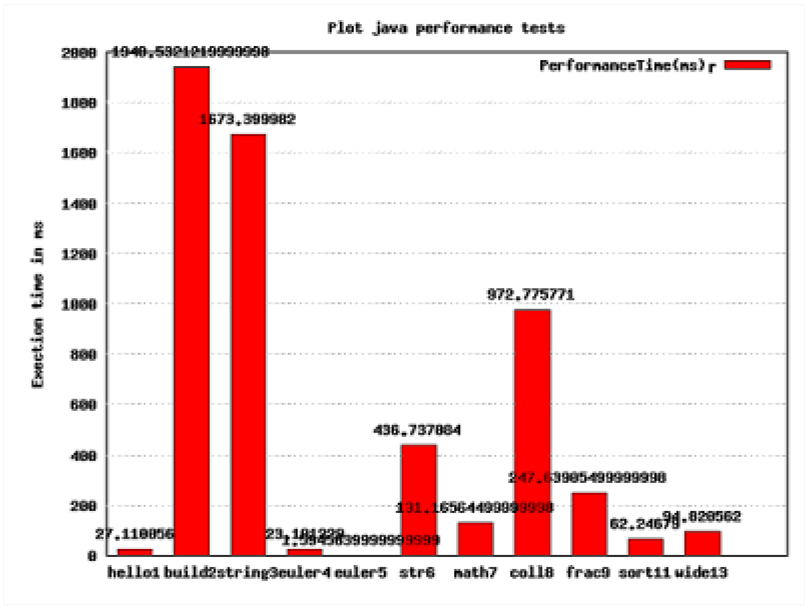
All Tests
I ran 14 different tests, ranging from a simple hello world application to the wider finder test. The figure below depicts some of the results.

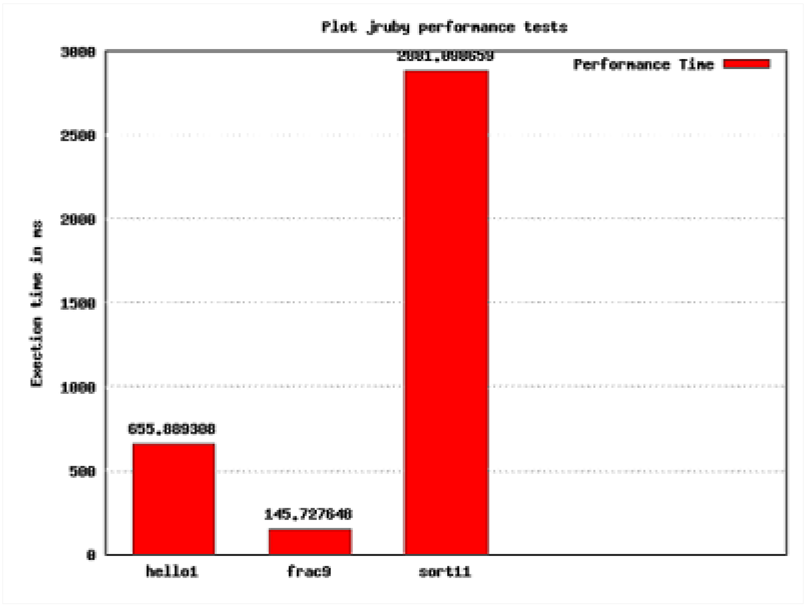


```
CljTest PerformanceTime(ms)
##### ###########
hello1  145.996411    Simple hello world test, more than 10k iterations
build2  3474.550155   Build large and small set of random strings
str3 495.028198    Various string operations
euler4  341.872834    Euler problem number 1
euler5  116.512002    Euler problem number 2
str6 213.379223    Additional string operations
math7 1212.796289   Misc Math operations, including sqrt and other calls
coll8  16356.185166  Collection operation, simple word frequency test
frac9  354.23239     Mandelbrot fractal
sort11  438.477163    Sorting routines, mostly quick sort
vars12  9542.282043   Large and small object creation
wide13  907.491465    Wide finder project, 10k and smaller number of lines
```



```
JavaTest  PerformanceTime(ms)
##### ###########
hello1  27.110056         Simple hello world test, more than 10k iterations
build2  1940.5321219      Build large and small set of random strings
string3 1673.399982       Various string operations
euler4  23.101229         Euler problem number 1
euler5 1.5945639999999   Euler problem number 2
str6 436.737084        Additional string operations
math7 131.1656449998    Misc Math operations, including sqrt and other calls
coll8  972.775771        Collection operation, simple word frequency test
frac9  247.63905499998   Mandelbrot fractal
sort11  62.24679          Sorting routines, mostly quick sort
wide13  94.820562         Wide finder project, 10k and smaller number of lines
```





Source for Results and Code

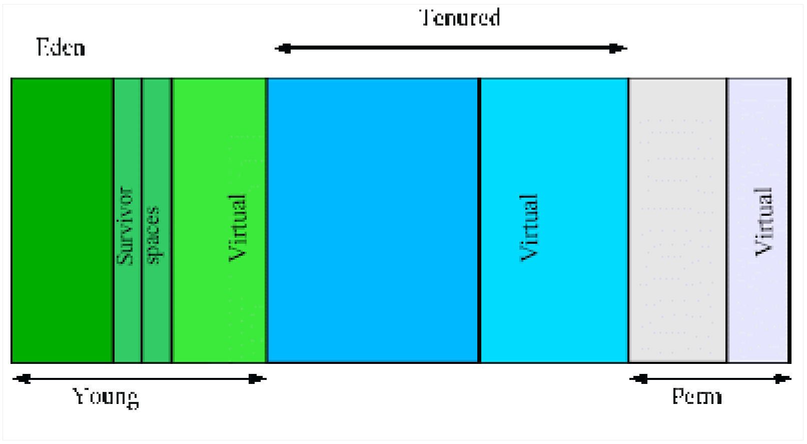
All of the source is available (common document license/BSD license) through the google-code browsable SVN URL. Download the results, language source code or gnuplot source. Unfortunately, this data is spread out haphazardly throughout the repository.

JVM Notebook - Test Results, Charts and Gnuplot scripts

JVM Notebook - Java, Clojure, JRuby, Scala Source

On Garbage Collection

"If the garbage collector has become a bottleneck, you may wish to customize the generation sizes. Check the verbose garbage collector output, and then explore the sensitivity of your individual performance metric to the garbage collector parameters." -- Tuning Garbage Collection [4]



The default arrangement of generations (for all collectors with the exception of the throughput collector) looks something like this.

```
Here is some output from the garbage colection report:
...
[GC 1277K->390K(5056K), 0.0006050 secs]
[GC 1286K->399K(5056K), 0.0005540 secs]
[GC 1294K->407K(5056K), 0.0005580 secs]
[GC 1303K->416K(5056K), 0.0009580 secs]
[GC 1311K->424K(5056K), 0.0006540 secs]
[GC 1320K->431K(5056K), 0.0007520 secs]
[GC 1327K->431K(5056K), 0.0012980 secs]
...
...
```

"With the line below, indicate the combined size of live objects before and after garbage collection, respectively. After minor collections the count includes objects that aren't necessarily alive but can't be reclaimed, either because they are directly alive, or because they are within or referenced from the tenured generation. The number in parenthesis"

(776768K)(in the first line)

"is the total available space, not counting the space in the permanent generation, which is the total heap minus one of the survivor spaces. The minor collection took about a quarter of a second." [5]

The usage of the entire heap was reduced to about 51%

196016K->133633K(261184K)

The output is generated with the 'verbosegc' and other JVM options:
(Note: The duplicated 'verbosegc' options gives more information about the garbage collecting)

java -verbosegc -verbosegc -verbosegc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps CrashJVM -Xms24m -Xmx32m > gc_anal.txt

There are 1400 data points in the output file and in the image for the Clojure sort example. Each GC line in the gc_anal analysis file contains GC information about a minor garbage collection.

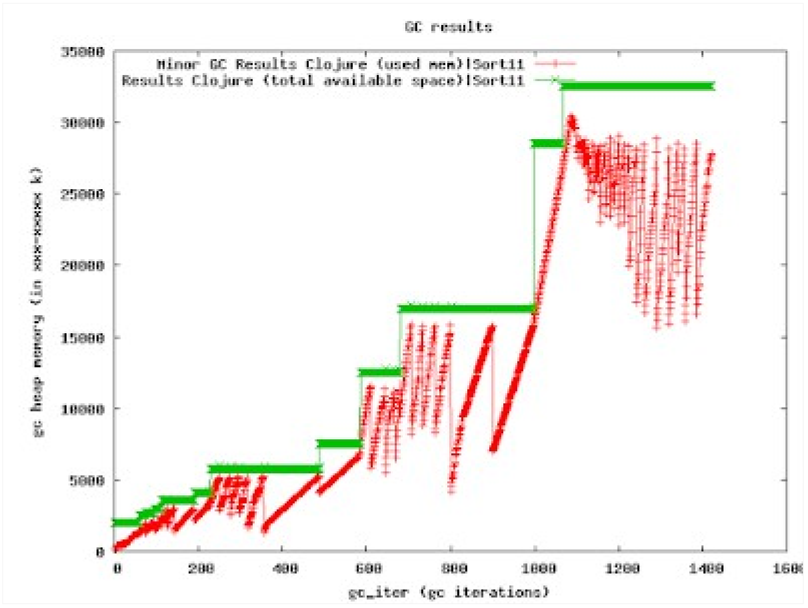
```
[GC [<collector>: <starting occupancy1> -> <ending occupancy1>, <pause time1> secs]
    <starting occupancy3> -> <ending occupancy3>, <pause time3> secs]
```

"If the incremental collection of the tenured generation cannot be completed before tenured generation is full, a major collection occurs and is indicated in the verbosegc output by the 'MSC' in the output."

```
<starting occupancy1> is the occupancy of the young generation
   before the collection
<ending occupancy1> is the occupancy of the young generation
   after the collection
<starting occupancy3> is the occupancy of the entire heap
   before the collection
<ending occupancy3> is the occupancy of the entire heap
  after the collection
```

see http://java.sun.com/docs/hotspot/gc1.4.2/example.html

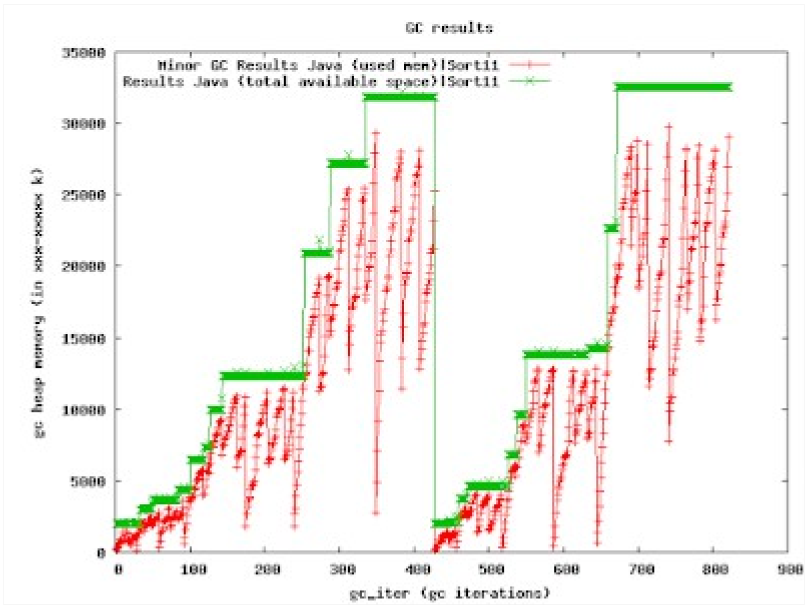
Our GC chart will only show 'used memory after the minor GC' and the total available memory after.



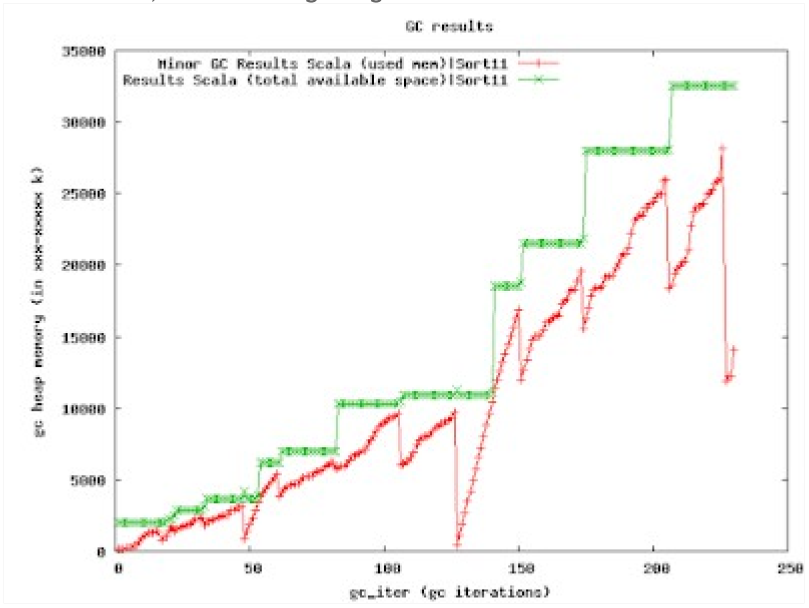
Here is the gnuplot representation of the used memory and total available after a minor garbage collection. There were 1400 data points in the clojure test. This is after running the quick sort code above.

```
Data Points used in graph:
used after gc | total after gc
...
...
26605 32576
26744 32576
27180 32576
27190 32576
27362 32576
27551 32576
line: 1421 -- 27684 32576
line: 1422 -- 27843 32576
```

The results for the pure Java version are presented below:



Scala results, 250+ minor garbage collections.

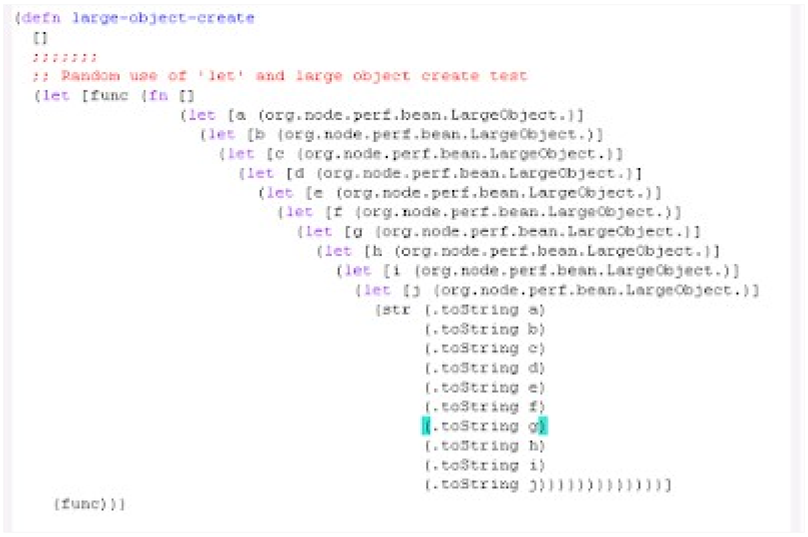


Large Objects

The quick sort example is not a good test to really push the garbage collector. Here is another test, one with Clojure and one with Java. I instantiate a large number of large objects and do the same for small objects. The garbage collector is better at handling smaller objects and consequently not as good at handling large objects (using the default GC rules). So, if you are looking at performance issues, you might look at how often large objects are being created and how long you are holding onto those objects. It is better to create many small objects and retain them for a short time than creating a few number large objects and retaining them for a long time. For example, I guess it is better to write short, static, utility methods and only create objects local to that method. "Large objects might be too big for Eden and will go directly to the old generation area; they will take longer to initialize (when setting fields to their default values such as null and zero); and they might cause fragmentation" [6]. A large object is one that the size of the allocation of the large object within a memory heap exceeds a maximum contiguous free space within the Java heap.

I attempted to create a large object that had many fields with lots of String data. Using my approximate sizeof utility method, it looks like it takes 800kb to create this object.

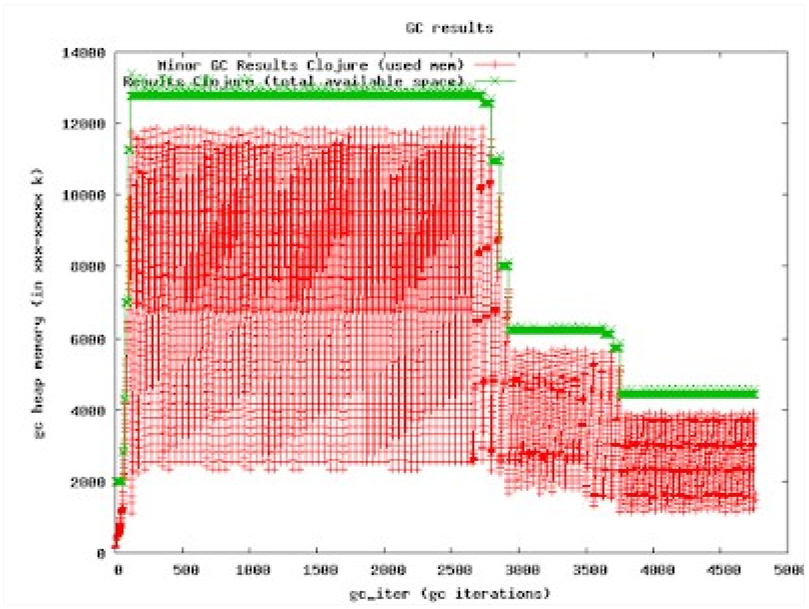
```
private Object obj5b = StringUtils.concat(StringUtils.randomString(random, (14 *
kb)),
  StringUtils.randomString(random, (14 * kb)));
```



```
(defn large-object-create
  []
  ;;;;;;;
  ;; Random use of 'let' and large object create test
  (let [func (fn []
                  (let [a (org.node.perf.bean.LargeObject.)]
                    (let [b (org.node.perf.bean.LargeObject.)]
                      (let [c (org.node.perf.bean.LargeObject.)]
                        (let [d (org.node.perf.bean.LargeObject.)]
                          (let [e (org.node.perf.bean.LargeObject.)]
                            (let [f (org.node.perf.bean.LargeObject.)]
                              (let [g (org.node.perf.bean.LargeObject.)]
                                (let [h (org.node.perf.bean.LargeObject.)]
                                  (let [i (org.node.perf.bean.LargeObject.)]
                                    (let [j (org.node.perf.bean.LargeObject.)]
                                      (str (.toString a)
                                           (.toString b)
                                           (.toString c)
                                           (.toString d)
                                           (.toString e)
                                           (.toString f)
                                           (.toString g)
                                           (.toString h)
                                           (.toString i)
                                           (.toString j))))))))))))))]
    (func)))
```

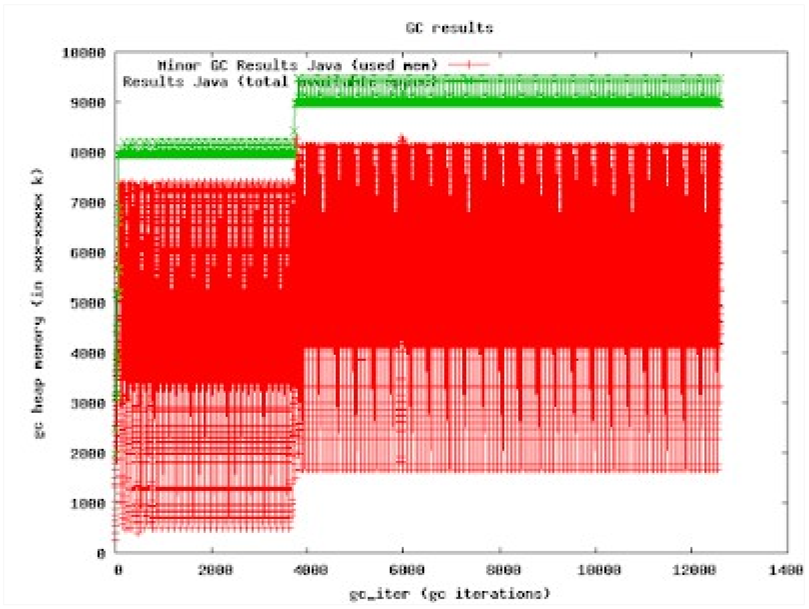
With this code, there were 4700 minor garbage collections. It was interesting that total available memory never goes over 14mb. Here is the final console system output after running the program. The last line shows the used memory and total available JVM memory.

```
53.236: [GC 53.236: [DefNew: 542K->28K(576K), 0.0002510 secs] 2155K->1642K(4448K),
0.0002997 secs]
53.243: [GC 53.244: [DefNew: 540K->25K(576K), 0.0002716 secs] 2154K->1639K(4448K),
0.0003217 secs]
53.252: [GC 53.252: [DefNew: 537K->18K(576K), 0.0002698 secs] 2151K->1631K(4448K),
0.0003206 secs]
"Elapsed time: 9691.390453 msecs"
Done
Performing simple garbage collection cooldown
(used:2M/2M [4M,31M ])
(used:2M/2M [4M,31M ])
Format for the memory usage line (clojure source to create the line):
(defn *memory-usage* []
  (str "(used:" (*used-memory-m*) "M/" (*free-memory-m*) "M [" (*total-memory-m*)
"M," (*max-memory-m*) "M ])"))
```



Shown below is the output for the Java test:

```
...
87.459: [GC 87.459: [DefNew: 441K->48K(640K), 0.0002922 secs] 5147K->4905K(8968K),
0.0003323 secs]
87.468: [GC 87.468: [DefNew: 624K->64K(640K), 0.0004192 secs] 5481K->5217K(8968K),
0.0004631 secs]
87.470: [GC 87.470: [DefNew: 213K->51K(640K), 0.0002372 secs] 5367K->5269K(8968K),
0.0002763 secs]
87.473: [GC 87.473: [DefNew: 627K->11K(640K), 0.0003707 secs] 5845K->5774K(8968K),
0.0004112 secs]
Elapsed time: 47982.607108 msecs
(used:5M/3M [8M,31M ])
(used:5M/3M [8M,31M ])
(used:5M/3M [8M,31M ])
Done
```

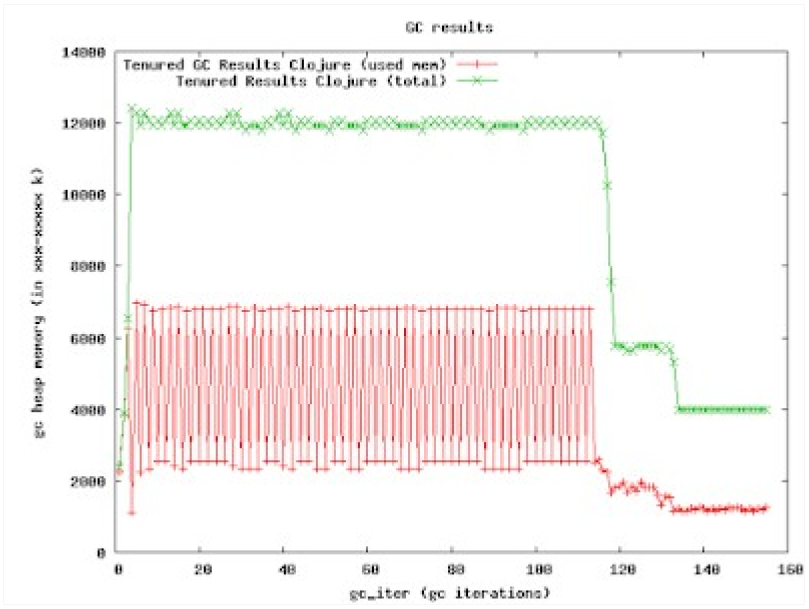


So far, we have only shown information on minor garbage collections. Here are the results of the major collects. The major collection are delineated by Tenured.

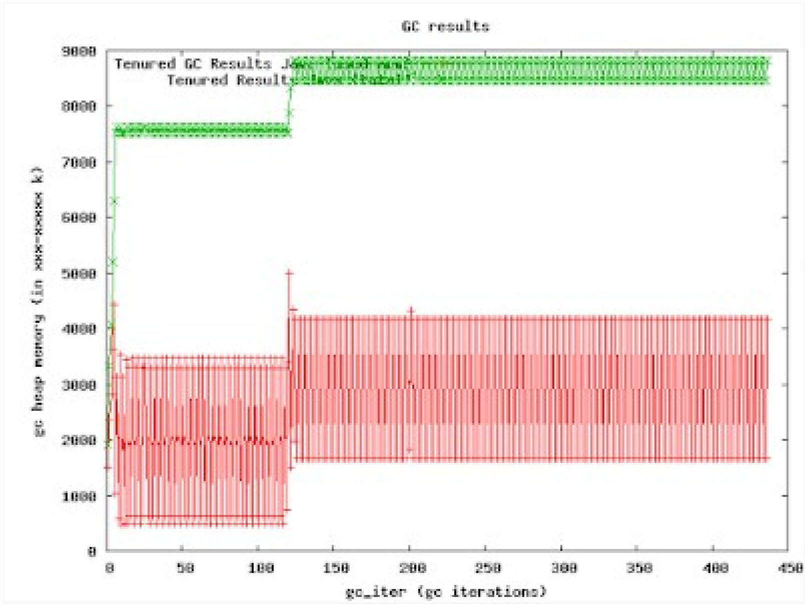
```
3.657: [GC 3.657: [DefNew: 568K->63K(576K), 0.0003546 secs]3.658: [Tenured: 7464K->506K(7508K), 0.0117550 secs] 7934K->506K(8084K), 0.0122011 secs]
```

You can see, major garbage collects seem to take more time and there are fewer of them in this result set.

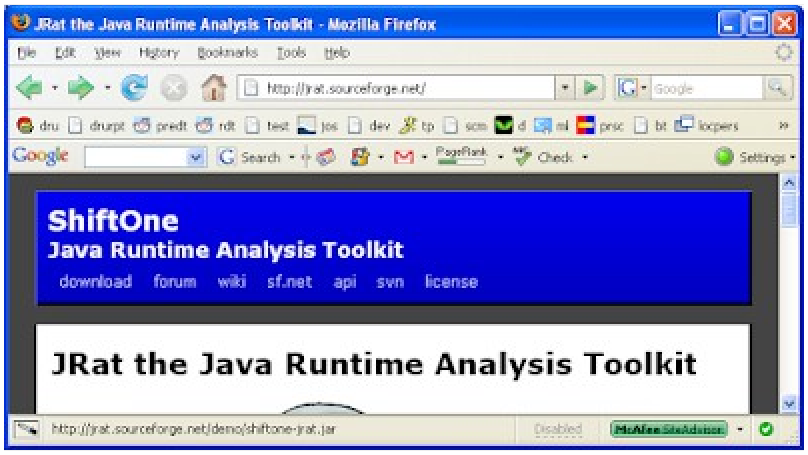
```
...
1167 4000
1235 4000
1193 4000
line: 156 -- 1260 4000
```



And the Java results. There were 440+ GC collects.



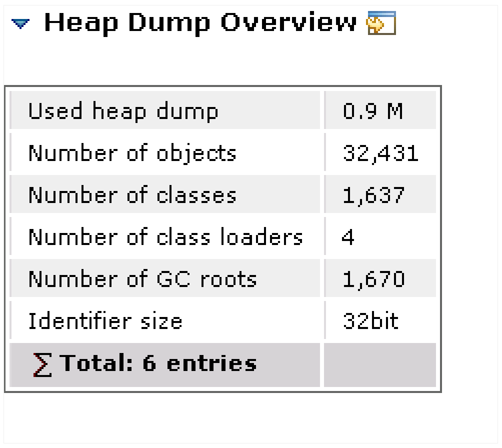
Additional Tools


JRat the Java Runtime Analysis Toolkit - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://jrat.sourceforge.net/
ShiftOne
Java Runtime Analysis Toolkit
download forum wiki sf.net api svn license
JRat the Java Runtime Analysis Toolkit
http://jrat.sourceforge.net/demo/shiftone-jrat.jar

There is no shortage of tools for monitoring JVM performance. Mature, open and proprietary monitoring applications are available. Most of them focus garbage collection, heap memory, cpu monitoring, and method trace calls. The netbeans monitor, Sun's jconsole, jrat, and the Eclipse memory analysis tool are a few that I have used recently.

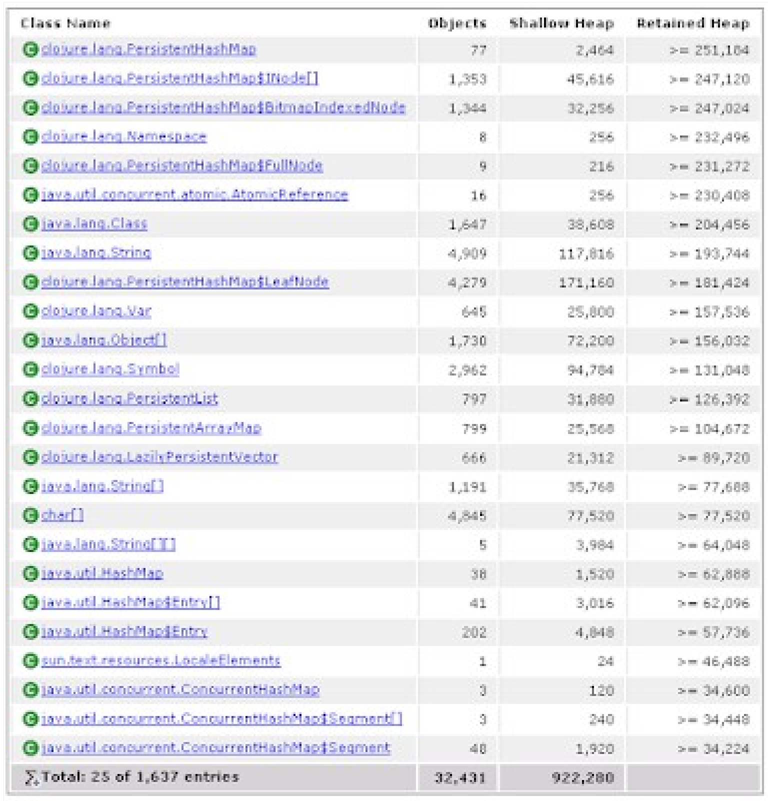
Looking at the top objects with the test examples

The chart report output from Eclipse's Memory Analysis Tool is depicted below. The overview contains the number of objects, number classes and classloader. The MAT tool also looks at potential memory leaks and the objects that are causing the problem, "biggest consumers" and a heap object histogram. To generate the data, we simply generate a heap dump and open the heap dump file.

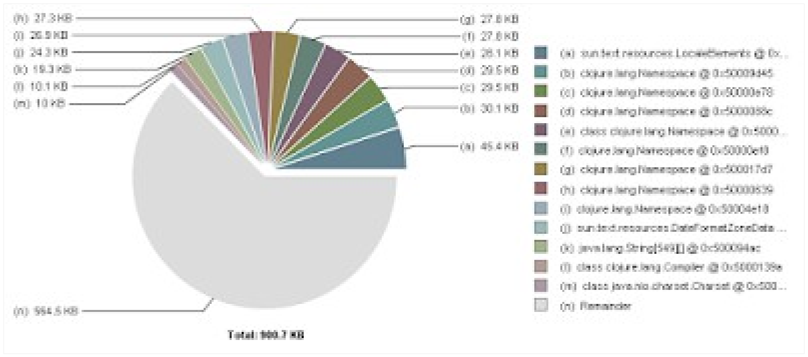


Heap Dump Overview

Used heap dump	0.9 M
Number of objects	32,431
Number of classes	1,637
Number of class loaders	4
Number of GC roots	1,670
Identifier size	32bit
∑ Total: 6 entries	

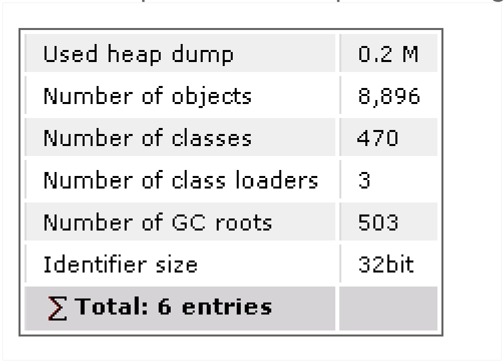


Class Name	Objects	Shallow Heap	Retained Heap
clojure.lang.PersistentHashMap	77	2,464	>= 251,184
clojure.lang.PersistentHashMap$INode[]	1,353	45,616	>= 247,120
clojure.lang.PersistentHashMap$BitmapIndexedNode	1,344	32,256	>= 247,024
clojure.lang.Namespace	8	256	>= 232,496
clojure.lang.PersistentHashMap$FullNode	9	216	>= 231,272
java.util.concurrent.atomic.AtomicReference	16	256	>= 230,408
java.lang.Class	1,647	38,608	>= 204,456
java.lang.String	4,909	117,816	>= 193,744
clojure.lang.PersistentHashMap$LeafNode	4,279	171,160	>= 181,424
clojure.lang.Var	645	25,800	>= 157,536
java.lang.Object[]	1,730	72,200	>= 156,032
clojure.lang.Symbol	2,962	94,784	>= 131,048
clojure.lang.PersistentList	797	31,880	>= 126,392
clojure.lang.PersistentArrayMap	799	25,568	>= 104,672
clojure.lang.LazilyPersistentVector	666	21,312	>= 89,720
java.lang.String[]	1,191	35,768	>= 77,688
char[]	4,845	77,520	>= 77,520
java.lang.String[][]	5	3,984	>= 64,048
java.util.HashMap	38	1,520	>= 62,888
java.util.HashMap$Entry[]	41	3,016	>= 62,096
java.util.HashMap$Entry	202	4,848	>= 57,736
sun.text.resources.LocaleElements	1	24	>= 46,488
java.util.concurrent.ConcurrentHashMap	3	120	>= 34,600
java.util.concurrent.ConcurrentHashMap$Segment[]	3	240	>= 34,448
java.util.concurrent.ConcurrentHashMap$Segment	48	1,920	>= 34,224
∑ Total: 25 of 1,637 entries	32,431	922,280	

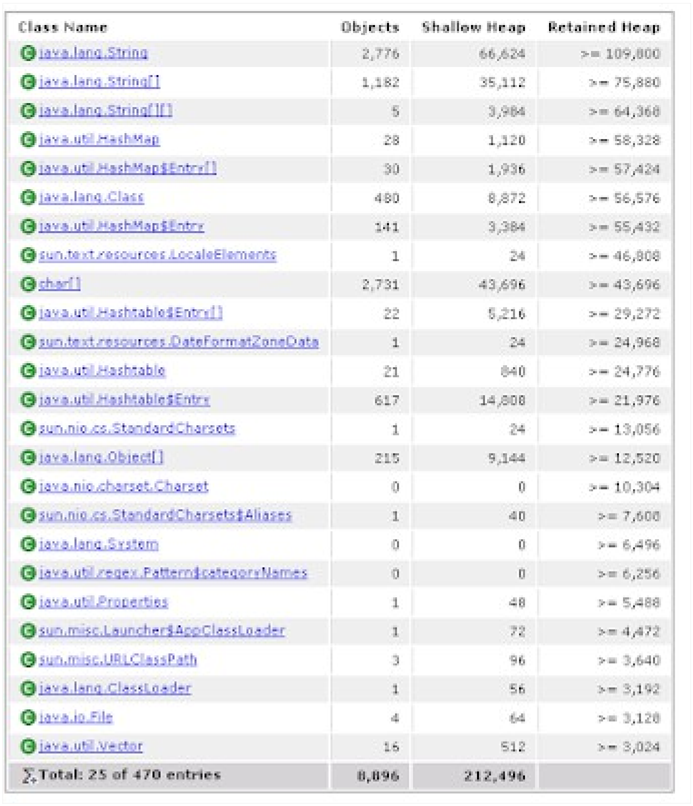


Even with the 'hello world' test, you can see that many objects were created. And no, the Clojure hello world application that I provided is not the typical hello world app. I actually do some basic multiplication calculations over 1,000,000 iterations. Hello world in this case is a misnomer.

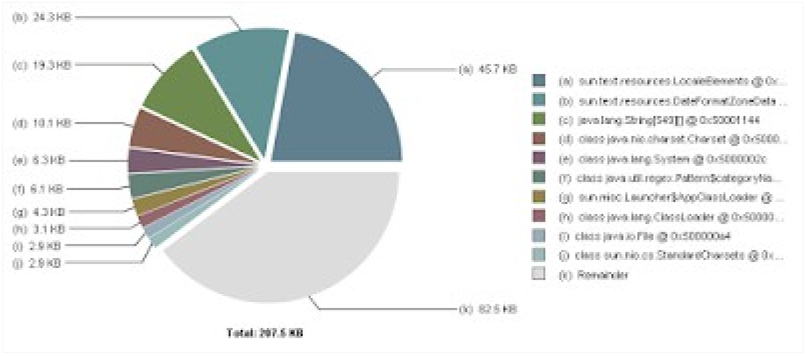
Here is comparable Java output. According to the Eclipse mat tool, almost 9000 objects were created.



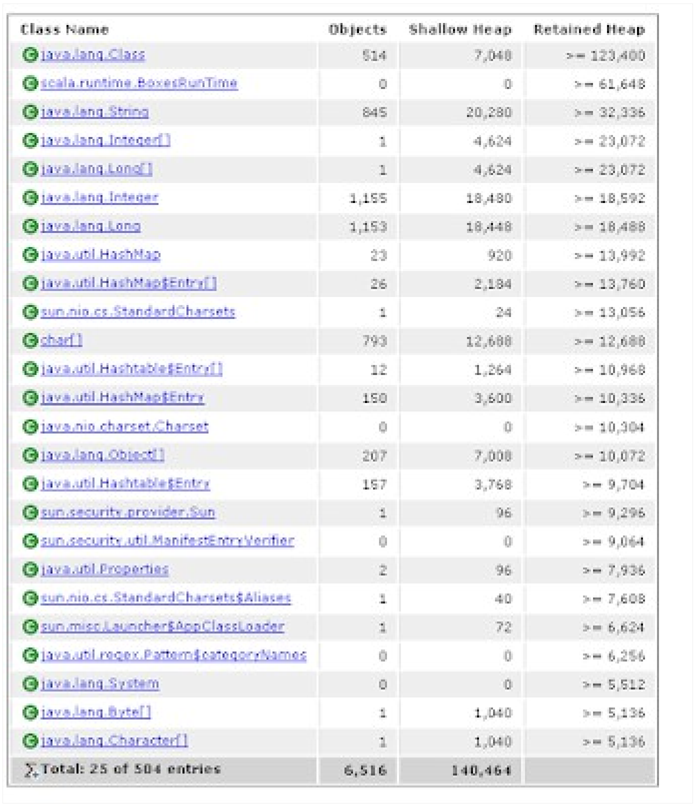
Used heap dump	0.2 M
Number of objects	8,896
Number of classes	470
Number of class loaders	3
Number of GC roots	503
Identifier size	32bit
∑ **Total: 6 entries**	



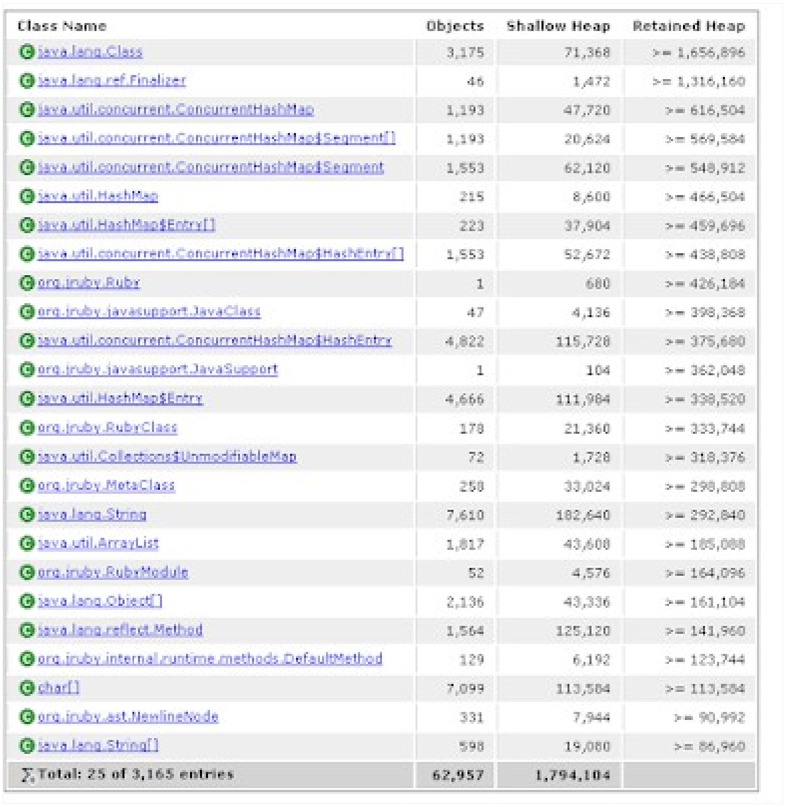
Class Name	Objects	Shallow Heap	Retained Heap
java.lang.String	2,776	66,624	>= 109,800
java.lang.String[]	1,182	35,112	>= 75,880
java.lang.String[][]	5	3,984	>= 64,368
java.util.HashMap	28	1,120	>= 58,328
java.util.HashMap$Entry[]	30	1,936	>= 57,424
java.lang.Class	480	8,872	>= 56,576
java.util.HashMap$Entry	141	3,384	>= 55,432
sun.text.resources.LocaleElements	1	24	>= 46,808
char[]	2,731	43,696	>= 43,696
java.util.Hashtable$Entry[]	22	5,216	>= 29,272
sun.text.resources.DateFormatZoneData	1	24	>= 24,968
java.util.Hashtable	21	840	>= 24,776
java.util.Hashtable$Entry	617	14,808	>= 21,976
sun.nio.cs.StandardCharsets	1	24	>= 13,056
java.lang.Object[]	215	9,144	>= 12,520
java.nio.charset.Charset	0	0	>= 10,304
sun.nio.cs.StandardCharsets$Aliases	1	40	>= 7,608
java.lang.System	0	0	>= 6,496
java.util.regex.Pattern$categoryNames	0	0	>= 6,256
java.util.Properties	1	48	>= 5,488
sun.misc.Launcher$AppClassLoader	1	72	>= 4,472
sun.misc.URLClassPath	3	96	>= 3,640
java.lang.ClassLoader	1	56	>= 3,192
java.io.File	4	64	>= 3,128
java.util.Vector	16	512	>= 3,024
∑ **Total: 25 of 470 entries**	**8,896**	**212,496**	



Scala and JRuby Class Histogram Results:



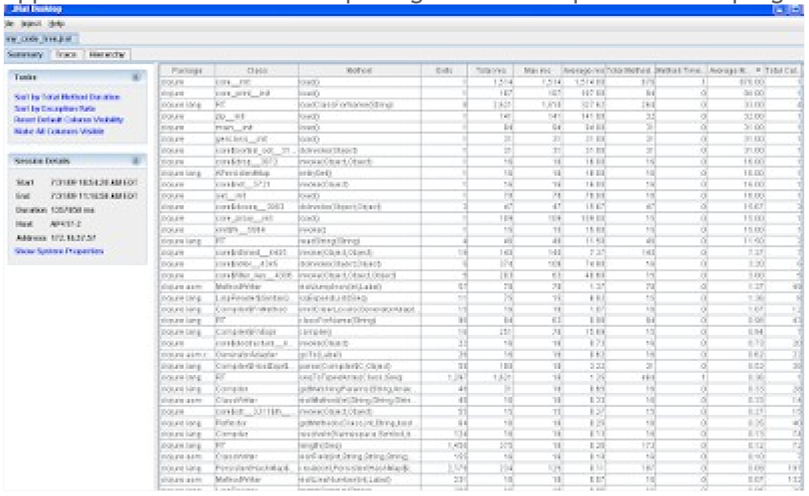
Class Name	Objects	Shallow Heap	Retained Heap
java.lang.Class	514	7,048	>= 123,400
scala.runtime.BoxesRunTime	0	0	>= 61,648
java.lang.String	845	20,280	>= 32,336
java.lang.Integer[]	1	4,624	>= 23,072
java.lang.Long[]	1	4,624	>= 23,072
java.lang.Integer	1,155	18,480	>= 18,592
java.lang.Long	1,153	18,448	>= 18,488
java.util.HashMap	23	920	>= 13,992
java.util.HashMap$Entry[]	26	2,184	>= 13,760
sun.nio.cs.StandardCharsets	1	24	>= 13,056
char[]	793	12,688	>= 12,688
java.util.Hashtable$Entry[]	12	1,264	>= 10,968
java.util.HashMap$Entry	150	3,600	>= 10,336
java.nio.charset.Charset	0	0	>= 10,304
java.lang.Object[]	207	7,008	>= 10,072
java.util.Hashtable$Entry	157	3,768	>= 9,704
sun.security.provider.Sun	1	96	>= 9,296
sun.security.util.ManifestEntryVerifier	0	0	>= 9,064
java.util.Properties	2	96	>= 7,936
sun.nio.cs.StandardCharsets$Aliases	1	40	>= 7,608
sun.misc.Launcher$AppClassLoader	1	72	>= 6,624
java.util.regex.Pattern$categoryNames	0	0	>= 6,256
java.lang.System	0	0	>= 5,512
java.lang.Byte[]	1	1,040	>= 5,136
java.lang.Character[]	1	1,040	>= 5,136
∑ Total: 25 of 504 entries	6,516	140,464	



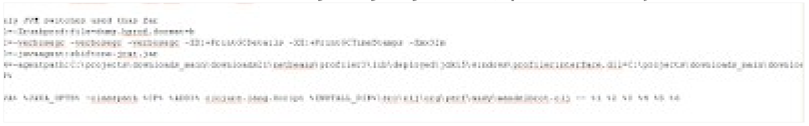
Class Name	Objects	Shallow Heap	Retained Heap
java.lang.Class	3,175	71,368	>= 1,656,896
java.lang.ref.Finalizer	46	1,472	>= 1,316,160
java.util.concurrent.ConcurrentHashMap	1,193	47,720	>= 616,504
java.util.concurrent.ConcurrentHashMap$Segment[]	1,193	20,624	>= 569,584
java.util.concurrent.ConcurrentHashMap$Segment	1,553	62,120	>= 548,912
java.util.HashMap	215	8,600	>= 466,504
java.util.HashMap$Entry[]	223	37,904	>= 459,696
java.util.concurrent.ConcurrentHashMap$HashEntry[]	1,553	52,672	>= 438,808
org.jruby.Ruby	1	680	>= 426,184
org.jruby.javasupport.JavaClass	47	4,136	>= 398,368
java.util.concurrent.ConcurrentHashMap$HashEntry	4,822	115,728	>= 375,680
org.jruby.javasupport.JavaSupport	1	104	>= 362,048
java.util.HashMap$Entry	4,666	111,984	>= 338,520
org.jruby.RubyClass	178	21,360	>= 333,744
java.util.Collections$UnmodifiableMap	72	1,728	>= 318,376
org.jruby.MetaClass	258	33,024	>= 298,808
java.lang.String	7,610	182,640	>= 292,840
java.util.ArrayList	1,817	43,608	>= 185,088
org.jruby.RubyModule	52	4,576	>= 164,096
java.lang.Object[]	2,136	43,336	>= 161,104
java.lang.reflect.Method	1,564	125,120	>= 141,960
org.jruby.internal.runtime.methods.DefaultMethod	129	6,192	>= 123,744
char[]	7,099	113,584	>= 113,584
org.jruby.ast.NewlineNode	331	7,944	>= 90,992
java.lang.String[]	598	19,080	>= 86,960
∑ Total: 25 of 3,165 entries	62,957	1,794,104	

Running JRat, Netbeans Profilers

Running JRat or Netbeans profilers is as simple as running with the appropriate JVM args allowing your application to run and then opening the files output after the program has exited.



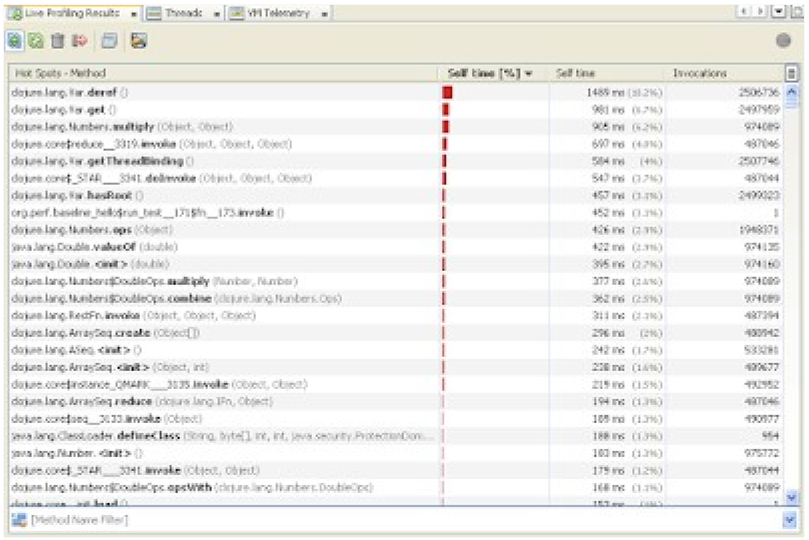
I downloaded the shiftone-jrat.jar jar file, placed it my current working directory. And then added these args:



```
[illegible]
-Djava.compiler=NONE [illegible]
-verbose:gc -verbose:gc -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps [illegible]
-javaagent:shiftone-jrat.jar
-agentpath:[illegible]netbeans\profiler3\lib\deployed\jdk15\windows\profilerinterface.dll=[illegible]

[illegible] clojure.lang.Script [illegible]mandelbrot.clj -- [illegible]
```

The screenshot above contains the jrat output after running the Clojure mandelbrot shootout application. I didn't want to analyze the results but there are a couple of stats worth taking a quick look at. The clojure.lang.Numbers.lt method had 87 million exits and a total method time of 142 milliseconds. The Numbers.ops method was called 367 million times.

The netbeans profiler contains similar profiling statistics as the jrat tool but the Netbeans profiler contains live, realtime results and a host of other metrics.



Hot Spots - Method	Self time [%]	Self time	Invocations
clojure.lang.Var.**deref** ()	🟥	1489 ms (10.2%)	2506736
clojure.lang.Var.**get** ()	🟥	981 ms (6.7%)	2497959
clojure.lang.Numbers.**multiply** (Object, Object)	🟥	905 ms (6.2%)	974089
clojure.core$reduce__3319.**invoke** (Object, Object, Object)	🟥	697 ms (4.8%)	487046
clojure.lang.Var.**getThreadBinding** ()	🟥	584 ms (4%)	2507746
clojure.core$_STAR__3341.**doInvoke** (Object, Object, Object)	🟥	547 ms (3.7%)	487044
clojure.lang.Var.**hasRoot** ()	🟥	457 ms (3.1%)	2499023
org.perf.baseline_hello$run_test__171$fn__173.**invoke** ()	🟥	452 ms (3.1%)	1
clojure.lang.Numbers.**ops** (Object)	🟥	426 ms (2.9%)	1948371
java.lang.Double.**valueOf** (double)	🟥	422 ms (2.9%)	974135
java.lang.Double.**<init>** (double)	🟥	395 ms (2.7%)	974160
clojure.lang.Numbers$DoubleOps.**multiply** (Number, Number)	🟥	377 ms (2.6%)	974089
clojure.lang.Numbers$DoubleOps.**combine** (clojure.lang.Numbers.Ops)	🟥	362 ms (2.5%)	974089
clojure.lang.RestFn.**invoke** (Object, Object, Object)	🟥	311 ms (2.1%)	487394
clojure.lang.ArraySeq.**create** (Object[])	🟥	296 ms (2%)	488942
clojure.lang.ASeq.**<init>** ()	🟥	242 ms (1.7%)	533281
clojure.lang.ArraySeq.**<init>** (Object, int)	🟥	238 ms (1.6%)	489677
clojure.core$instance_QMARK__3135.**invoke** (Object, Object)	🟥	219 ms (1.5%)	492952
clojure.lang.ArraySeq.**reduce** (clojure.lang.IFn, Object)	🟥	194 ms (1.3%)	487046
clojure.core$seq__3133.**invoke** (Object)	🟥	189 ms (1.3%)	490977
java.lang.ClassLoader.**defineClass** (String, byte[], int, int, java.security.ProtectionDom...	🟥	188 ms (1.3%)	954
java.lang.Number.**<init>** ()	🟥	183 ms (1.3%)	975772
clojure.core$_STAR__3341.**invoke** (Object, Object)	🟥	179 ms (1.2%)	487044
clojure.lang.Numbers$DoubleOps.**opsWith** (clojure.lang.Numbers.DoubleOps)	🟥	168 ms (1.1%)	974089

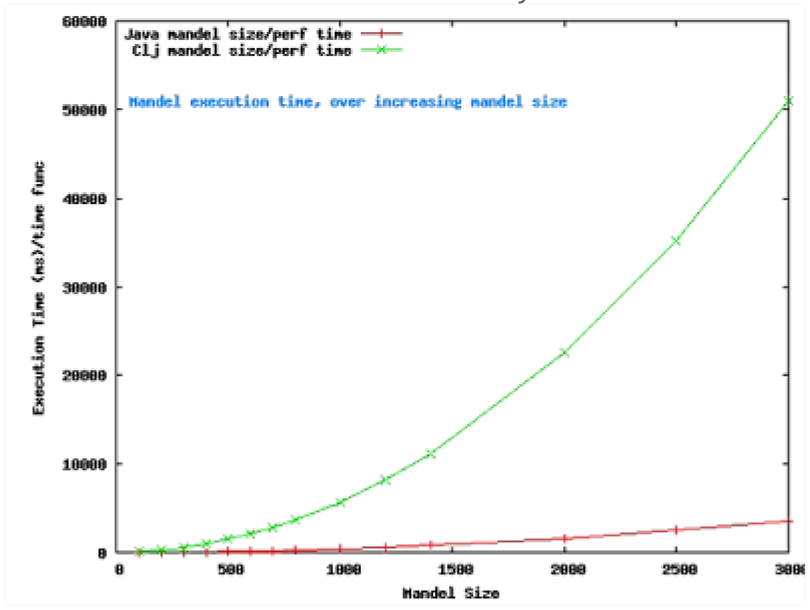
Additional Tests from Third Party (from AndyF):

Andy, a developer provided great Clojure performance information. I ran some of his Clojure source. Here are the runtime, garbage collection and memory results.
Source
JVM Notebook SVN source (Andy Benchmark) - Java
JVM
Notebook SVN source (Andy Benchmark) - Clojure

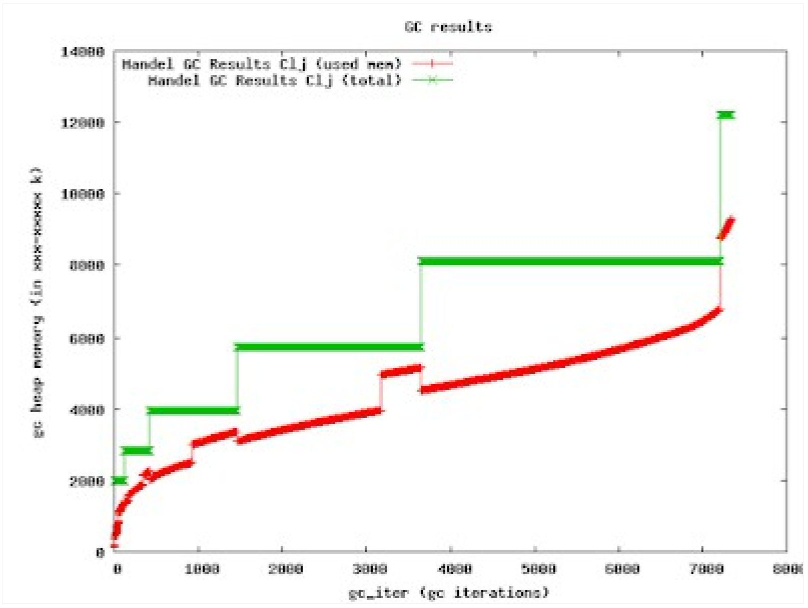
Original Source

Updated Benchmark Results (the rcomp test shows 1:15 performance speed for Clojure)

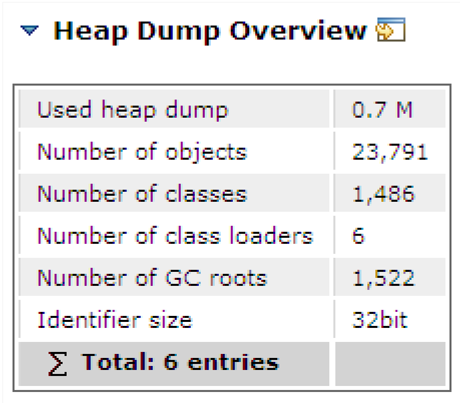
I ran two types of shootout like tests, the threaded, non-threaded Mandelbrot test in Java and Clojure. Performance for these were similar to the all of the tests that we have run up to this point. The mandelbrot code seemed to perform slower than the 1:10 (java:clojure) speed ratio that we encountered. For example, the Java threaded code ran at '1700.062' ms. The Clojure version ran at 50998.30 ms.



MandelSiz|	exectime.java(ms)	exectime.clojure(ms)
#####	#########	#####
100	11.515956	90.97791
200	22.154556	263.893038
300	42.036901	610.872097
400	70.4424729	982.452582
500	105.705857	1495.433005
600	151.296857	2091.412938
700	201.6435589	2843.495665
800	260.195961	3650.13744
1000	407.073079	5680.39932
1200	577.353532	8208.967304
1400	785.8052849	11126.994209
2000	1593.239278	22588.712673
2500	2487.326002	35210.526506
3000	3576.071694	50998.301209

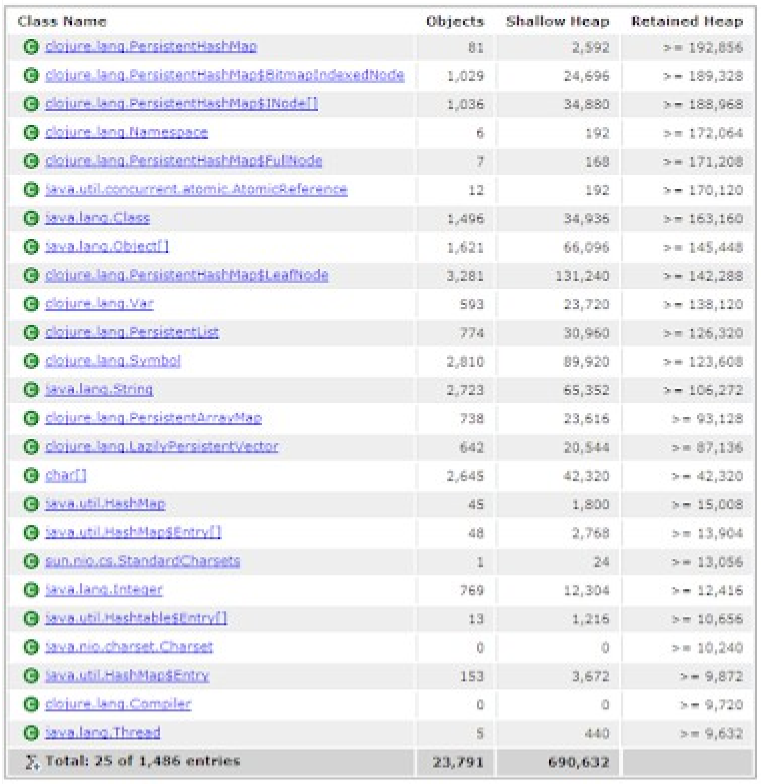


The chart above contain the garbage collection results after running the Clojure mandelbrot shootout test.



▼ **Heap Dump Overview**

Used heap dump	0.7 M
Number of objects	23,791
Number of classes	1,486
Number of class loaders	6
Number of GC roots	1,522
Identifier size	32bit
∑ **Total: 6 entries**	



Class Name	Objects	Shallow Heap	Retained Heap
clojure.lang.PersistentHashMap	81	2,592	>= 192,856
clojure.lang.PersistentHashMap$BitmapIndexedNode	1,029	24,696	>= 189,328
clojure.lang.PersistentHashMap$INode[]	1,036	34,880	>= 188,968
clojure.lang.Namespace	6	192	>= 172,064
clojure.lang.PersistentHashMap$FullNode	7	168	>= 171,208
java.util.concurrent.atomic.AtomicReference	12	192	>= 170,120
java.lang.Class	1,496	34,936	>= 163,160
java.lang.Object[]	1,621	66,096	>= 145,448
clojure.lang.PersistentHashMap$LeafNode	3,281	131,240	>= 142,288
clojure.lang.Var	593	23,720	>= 138,120
clojure.lang.PersistentList	774	30,960	>= 126,320
clojure.lang.Symbol	2,810	89,920	>= 123,608
java.lang.String	2,723	65,352	>= 106,272
clojure.lang.PersistentArrayMap	738	23,616	>= 93,128
clojure.lang.LazilyPersistentVector	642	20,544	>= 87,136
char[]	2,645	42,320	>= 42,320
java.util.HashMap	45	1,800	>= 15,008
java.util.HashMap$Entry[]	48	2,768	>= 13,904
sun.nio.cs.StandardCharsets	1	24	>= 13,056
java.lang.Integer	769	12,304	>= 12,416
java.util.Hashtable$Entry[]	13	1,216	>= 10,656
java.nio.charset.Charset	0	0	>= 10,240
java.util.HashMap$Entry	153	3,672	>= 9,872
clojure.lang.Compiler	0	0	>= 9,720
java.lang.Thread	5	440	>= 9,632
Total: 25 of 1,486 entries	**23,791**	**690,632**	

The following test results are taken verbatim from the clojure-benchmark result set:

```
Additional Results straight from the shootout source (by AndyF):
See:
http://github.com/jafingerhut/clojure-benchmarks/
---------------------------------------------------
Times are real / user / sys on iMac
---------------------------------------------------
        |  sbcl  |  perl  |   ghc  |  java  |   clj
---------------------------------------------------
```

```
           noT      noT            |  noT
long test on iMac:
rcomp   |   8.3  |  11.9  |        |   8.5  | no implementation yet
        |   4.8  |   7.9  |        |   3.6  |
        |   2.1  |   2.3  |        |   1.4  |
long test on iMac:
mand-   | wrong  | out of |  32.7  |  28.6  | 340.4
elbrot  | output | mem    |  59.3  |  54.4  | 350.5
        |        | (?)    |   0.8  |   0.4  |   4.7
           noT      noT            |    T
long test on iMac:
k-nuc-  | 190.9  | 306.0  |  90.5  |  52.4  | 1677.6 (27m 57.6s)
leotide | 187.9  | 302.7  | 130.8  |  89.6  | 2245.1 (37m 25.1s)
        |   2.4  |   1.9  |   4.6  |   1.8  |   24.2 (    24.2s)
Result: Replacing dna-char-to-code-val with a macro did
not speed things up, and may have slowed them down:
                                            | 1800.0 (30m  0.0s)
                                            | 2317.0 (38m 37.0s)
                                            |   30.6 (    30.6s)
-------------------------------------------------
k-nucleotide long test on benchmark shootout machine:
        | 164.9  | 249.8  |  52.0  |  20.6  |
        | 164.9  | 246.7  | 112.5  |  58.8  |
        |     ?  |     ?  |     ?  |     ?  |
-------------------------------------------------
k-nucleotide medium test on iMac:
        |   8.6  |  12.7  |   3.9  |   3.9  |   64.2 / 69.6 / 69.1 / 67.1
        |   7.9  |  12.5  |   5.4  |   5.7  |   98.4 / 92.9 / 93.1 / 88.1
        |   0.6  |   0.1  |   0.3  |   0.2  |    1.5 /  1.6 /  1.6 /  1.7
k-nucleotide medium test, all clj, modified-pmap with specified number
of parallel threads, on iMac:
        |   1    |   2    |   3    |   4    |   5    |   6    |
        |  74.9  |  70.7  |  77.2  |  76.8  |  82.5  |  77.8  |
        | 125.9  | 122.1  | 134.6  | 134.0  | 143.4  | 134.0  |
        |   2.5  |   2.8  |   2.8  |   2.3  |   3.1  |   2.3  |
-------------------------------------------------
Hardware and software used
iMac with Intel Core 2 Duo, 2.16 GHz, 2 GB RAM, Mac OS X 10.5.7
% java -version
java version "1.6.0_13"
Java(TM) SE Runtime Environment (build 1.6.0_13-b03-211)
Java HotSpot(TM) 64-Bit Server VM (build 11.3-b02-83, mixed mode)
% javac -version
javac 1.6.0_13
user> (clojure-version)
"1.1.0-alpha-SNAPSHOT"
End of Andy's Configuration:
-------------------------------------------------
The following benchmark information were provided
by Andy from his website.   Visit:
http://github.com/jafingerhut/clojure-benchmarks/
Some additional google clojure threads:
http://groups.google.com/group/clojure/browse_thread/thread/e4299911f4ae8167
```

Testing Environment

For this document, all of the tests were run with Java 5. (Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_11-b03)). Some tests were run with both the -client and -server options. The majority of tests were run with the -client option. Headius describes the -server option, "The -server option turns on the optimizing JIT along with a few other "server-class" settings.". Updates to the this document will include tests with Java 6.

Operating System and Hardware Environment:

•**Java:** Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_11-b03)

•**Windows:** Windows XP Professional Service Pack 2

•**Hardware:** Intel Core 2 CPU 6300 - 1.86ghZ, 3.49 GB of RAM

Virtual Machine Languages:

The focus of these tests are mostly to look at Clojure and how it compares to pure Java code. Clojure is included in all of the tests. There are only two tests for Scala.

•**Clojure: 1.0.0/ 2009-05-04**

•**JRuby: 1.3.1 / 2009-06-15; 44 days ago**

•**Scala: 2.7.5/ 2009-06-03; 50 days ago**

Resources
[1] - measure quote - http://www.javaperformancetuning.com/news/newtips087.shtml - quote from the java performance tips. I have not found the original source of this quote.
[2] - Rick Ross quote - Farewell to the J in JVM
[3] - Zed Shaw on Statistics - http://zedshaw.com/essays/programmer_stats.html
[4] - Clojure Home - http://clojure.org/
[5] - Tuning Garbage Collection - http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html
[6] - On large vs small - http://chaoticjava.com/posts/gc-tips-and-memory-leaks/
[7] - Scala's, static typing - http://en.wikipedia.org/wiki/Scala_(programming_language)#Static_typing
[8] - http://creativekarma.com/ee.php/weblog/comments/static_typing_and_scala/

Additional Links

http://clj-me.blogspot.com/2008/06/widefinder-2-in-clojure-naive-port-from.html
http://onjava.com/pub/a/onjava/2001/05/30/optimization.html
http://onjava.com/onjava/2001/05/30/examples/Test.java
http://www.javaperformancetuning.com/news/newtips087.shtml
http://blog.headius.com/2009/01/my-favorite-hotspot-jvm-flags.html

http://jrat.sourceforge.net/
http://www.netbeans.org/features/java/profiler.html

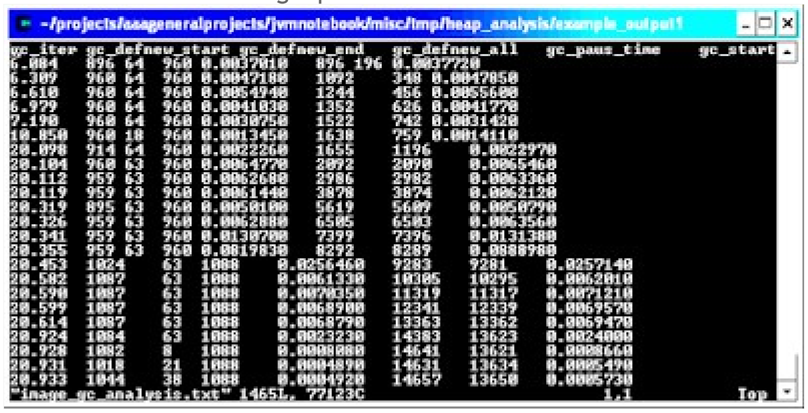
Download or Browse All Source:
JVM Notebook - Test Results, Charts and Gnuplot scripts
JVM Notebook - Java, Clojure, JRuby, Scala Source

Download All Source as Archived File

Partial PDF version of this document
Tex source for this document

Addendum (gnuplot and regex)

I always feel that charts and graph are great visuals for displaying simple or complex concepts. I only know the surface level details of gnuplot, but here are some notes on creating the images above. I used a simple python script to analyze the GC statements generated by the JVM verbosegc output and then created a tabulated data file that can used with gnuplot.



```
~/projects/asagneralprojects/jvmnotebook/misc/tmp/heap_analysis/example_output1
gc_iter gc_defnew_start gc_defnew_end   gc_defnew_all   gc_paus_time    gc_start
6.004   896 64  960 0.0037010   896 196 0.0037720
6.309   960 64  960 0.0047100   1092    348 0.0047850
6.610   960 64  960 0.0054940   1244    456 0.0055600
6.979   960 64  960 0.0041030   1352    626 0.0041770
7.190   960 64  960 0.0030750   1522    742 0.0031420
10.850  960 18  960 0.0013450   1638    759 0.0014110
20.098  914 64  960 0.0022260   1655    1196    0.0022970
20.104  960 63  960 0.0064770   2072    2070    0.0065460
20.112  959 63  960 0.0062600   2986    2982    0.0063360
20.119  959 63  960 0.0061440   3878    3874    0.0062120
20.319  895 63  960 0.0050100   5619    5609    0.0050790
20.326  959 63  960 0.0062880   6505    6503    0.0063560
20.341  959 63  960 0.0130700   7399    7396    0.0131380
20.355  959 63  960 0.0819830   8292    8289    0.0800980
20.453  1024    63  1088    0.0256460   9283    9281    0.0257140
20.582  1087    63  1088    0.0061330   10305   10295   0.0062010
20.590  1087    63  1088    0.0070350   11319   11317   0.0071210
20.599  1087    63  1088    0.0068900   12341   12339   0.0069570
20.614  1087    63  1088    0.0068790   13363   13362   0.0069470
20.924  1084    63  1088    0.0023230   14383   13623   0.0024000
20.928  1082    0   1088    0.0000000   14641   13621   0.0008660
20.931  1018    21  1088    0.0004890   14631   13634   0.0005490
20.933  1044    38  1088    0.0004920   14657   13650   0.0005730
"image_gc_analysis.txt" 1465L, 77123C                          1,1           Top
```

I typically will create a shell script to set all the gnuplot commands and then launch the application:

```
#!/bin/sh
# Also see:
# http://www.duke.edu/~hpgavin/gnuplot.html
#
# Use 'eog' eye of gnome to view the images
OUTPUT_IMG_FILE=image_gc_analysis.png
INPUT_PLOT_FILE=image_gc_analysis.txt
# Output these commands to a temp file and then run gnu plot
echo "set terminal png
set output '${OUTPUT_IMG_FILE}'
set title 'GC results'
set size 1,1
set key left top
set autoscale
set xlabel 'gc_iter'
set ylabel 'gc_main_paus_time'
plot '${INPUT_PLOT_FILE}' using 1:6 title 'GC Results'
" > gnuplot_tmp_cmd.tmp
# Run the gnu plot command
gnuplot gnuplot_tmp_cmd.tmp > /dev/null
# For future reference, add a command and title to compare results
# E.g.
# '/tmp/data2' using 10 with lines title 'Benchmark 2',
```

On Regex
In the gc_parse.py python application, I used a couple of lines of regex to find the GC patterns generated by the verbose gc JVM output.

```
def get_gc_stats(line):
        # Example Output java 1.5:
        # 20.049: [GC 20.049: [DefNew: 1941K->137K(1576K), 0.0014153 secs] 11194K-
>11117K(1984K), 0.0014628 secs]
        # Example Output java 1.6:
        # 20.049: [GC 28.200: [DefNew: 1921K->137K(1984K), 0.0006890 secs] 23030K-
>21247K(27320K), 0.0007550 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

        # Group1 = First Iter Value
        # Group2 = Second 'GC' Iter value
        regex_begin_iter = '^(\S*):\s*\[GC\s*(\S*):\s*'
        # Group 3 = Young Gen Start Occupancy
        # Group 5 = Young Gen End Occupancy
```

```
regex_def_new    = '\[DefNew:\s*(\S*)(K|M)\-\>(\S*)(K|M)\((.*)$'
regex_full_str   = '%s%s' % (regex_begin_iter, regex_def_new)
pattr = re.compile(regex_full_str)
# Group 1 = Main Start Occupancy
# Group 2 = Main End Occupancy
pattr_sub = re.compile('^(.*) (\S*)(K|M)\-\>(\S*)(K|M)\((.*)$')
...
...
```

Java and Unit Testing

I have read and I am reading about four or five posts a day about unit testing. It really has been a long time obsession for me. I have moved past the technical and practical considerations on unit testing frameworks and done with the debates with "should you use Junit or Mockito or Karma?" I am more interested in the psychology of unit testing, who does it, likes it, hates it? It really is one of those easy to learn, hard to master concepts. For example, many many may play chess when they are young and can end up being horribly chess players most of their life, I am part of that majority. Unfortunately, I have never played chess and sat down for hours and tried to master it. I never see the common patterns or have a developed end game. I mostly just play with a knowledge of the basic rules. Following good unit testing practices within your software development shop is a lot like playing chess. It is easy to learn and difficult master. Actually, there are a lot of big differences, chess is a game, chess is not coding, and people take their software development very seriously. So if you don't master unit testing, but are able to complete your job tasks, some might argue that is an acceptable risk in the world of software development. And why master chess or master unit testing? If developers are fine without unit testing, then why even suggest it. Some developers just don't want to invest the energy to master the practice. And in some development shops, there is no hard requirement to do so.

I am not going to convince you to write unit tests with this one post, I will leave that up to software guru Martin Fowler and the people at ThoughtWorks who have written large tomes on the subject. But I will present my thoughts on why some developers won't write unit tests but why they should. Those developers and architects that do advocate unit testing generally fall into that category where they have written just enough unit tests to find it useful and they generally love the practice, they also encourage others to follow along. I am sort of in that camp, I have almost become religious about it. I can't imagine my real code without unit tests and I just feel guilty by only testing through manual functional testing.

Jeff Atwood of Coding Horror wrote a short blog post on the topic, "I Pity the Fool Who Doesn't Write Unit Tests". Here is the one blurb that stuck out for me, "Even if you only agree with a quarter of the items on that list-- and I'd say at least half of them are true in my experience-- that is a huge step forward for software developers". And this one, "It's more fun to code with them than without". That is the essence of this unit testing religion, we can't force it on developers and we can't force developers to write unit tests only a certain way. I and many others don't believe in the practice of 100% coverage. You will rarely get there anyway, depending on the project or company. Some will argue that you shouldn't break the rule on non-determinism and this is a big one. Basically, the unit test should return the same output every time you run the test. You should avoid breaking this rule for unit tests but you can still write and add automated integration tests to your suite and not waste time, combine a collection of unit tests and integration tests. A simple integration test might test connecting to your REST microservice and validating the HTTP status code. At that point, your test moves into the

integration testing category. If you connect to the database, run a particular SQL statement and validate data model returned from the SQL invocation, then your test is basically integration. Both scenarios are not units are non-deterministic but I would still consider them to be useful. Also, as a start for new developers getting familiar with unit testing, writing integration tests may be more familiar to them than decomposing or refactoring their code for a real unit test. There is a benefit in database or HTTP integration tests, you can add them to a test suite and run them in a automated form after a code change and after a build. Even bad tests can be useful.
Misko Hevery is creator of one of the most popular JavaScript frameworks to emerge in the last couple of years. It is a Google project that he started working as an Agile Coach. As he puts it, he wants to maintain the high level of automated testing culture at Google. Most of his published articles are not about AngularJs but on the benefits of automated testing. I can only imagine that he developed the MVC JavaScript framework because the old crop of frameworks were a pain to work with for developers. They were not testable.
I have given my advocacy speech on unit testing, but how do I use it, what practices do I follow?

•For every piece of new code, I formulate a unit test case. New code could include my model structure or interface into my Java services. This is critical, unit testing encourages you to write testable early code. Meaning, I try to use interfaces and abstract classes which allow me to inject mock objects early in the development process.
•For local development, I can build, write code, write and update my unit tests and then run the automated suite of tests. The key part is re-running the test suite. Normally I want my unit tests to pass, if they don't pass then I can look at my code and refactor. Also, the code I write today, I can run a year from now, I should expect the same result.
•As you are writing your unit tests. Have fun, this is not production code, the unit tests don't run in production, you can test input as little or as much as possible.
•I try to avoid unit tests around code that doesn't do anything. Write unit tests around your modules that have some kind of behavior. We shouldn't write model POJO code with setters and getters, but there is no reason to test a setter method. It is more fun to code around the real functionality.
•Writing unit tests also encourages the developer to write testable code

•Write Java code that doesn't use static methods or variables. Imagine that, try writing code that doesn't make use of the static keyword. Why would you do this? Static, class level routines are procedural and inherently hard to test. You can override their functionality, they are completely class level.

•Writing unit tests encourages refactoring. Some refactoring may include the use of OOP techniques. Use interfaces and abstract classes.

•Use a DI/Dependency Injection framework like AngularJS (yea I called AngularJS DI), Spring or Guice. DI frameworks encourages the container to create new objects for you. Managing objects on your own and using the 'new' operator encourages untestable code.

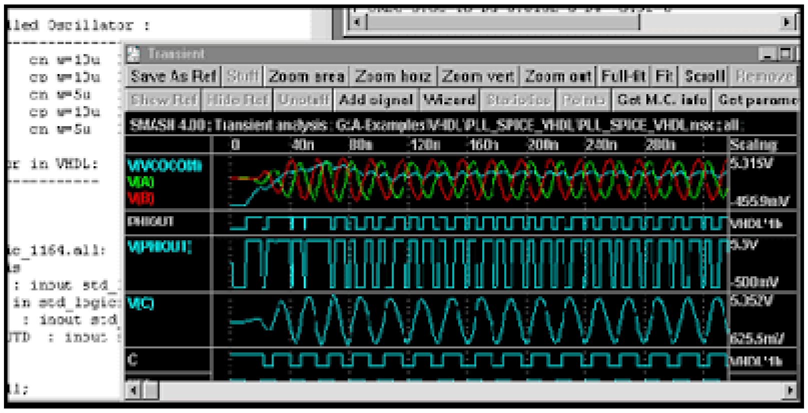
In Summary, see what Jeff Atwood, Martin Fowler and Misko Hevery have said about Unit Testing. And we pity the fool that don't do it.

Math

Wolfram Cellular Automata

Overview - Playing the Game Of Life

When most computer users upload a profile image from their desktop to Facebook's website they don't stop to think about the simple binary math rules that are fundamental to most digital devices. We realize that 4 gigabytes of RAM is more memory than 512 megabytes but we don't visualize the logic chips that are involved in an xor $0x100, eax operation for a 32-bit CISC processor. Software developers have to consider memory management or how a computer's operating system loads their programs into memory. They don't normally consider VHDL logic circuit designs, the data paths, arithmetic logic units or the millions of transistors that make up a modern CPU. Those low-level details have been intentionally hidden from the user application developer. The modern CPU may have changed dramatically over the last decade but at the heart of early digital computing were simple Boolean operations. These simple rules were combined together and logic replicated to load programs into memory and then execute. The rules that control most digital devices are based on elementary Boolean rules. Cellular automata has a similar bottom-up approach, rules consist of simple programs (as Stephen Wolfram calls them) that apply to a set of cells on a grid.



Conway's Game of Life cellular automaton is one of the most prominent examples of cellular automata theory. The one dimensional program consists of a cell grid typically with several dozen or more rows and similar number of columns. Each cell on the grid has an on or off Boolean state. Every cell on the grid survives or dies to the next generation depending on the game of life rules. If there are too many neighbors surrounding a cell then the cell dies due to overcrowding. If there is only one neighbor cell, the base cell dies due to under-population. Activity on a particular cell is not interesting but when you run the entire system for many generations, a group of patterns begin to form.

More on Conway's Game of Life

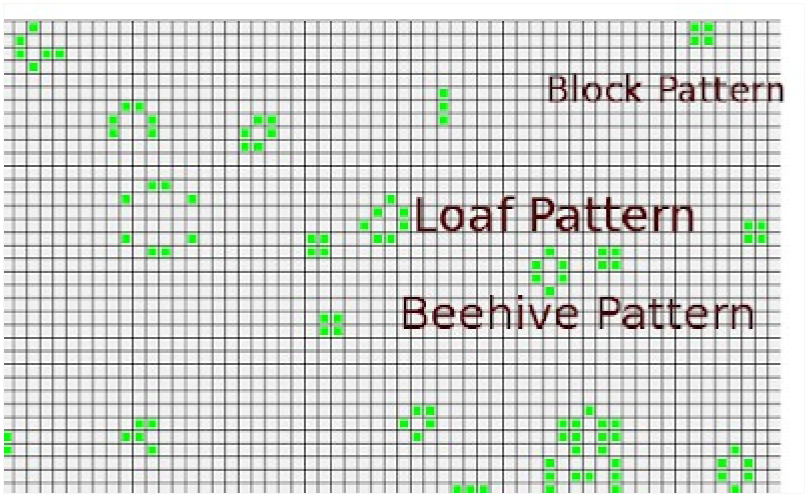


Figure: Game of Life Output

You may notice some common patterns in the figure. After so many iterations through the game of life rules, only a few cells tend to stay alive. We started with a large random number of alive cells and over time those cells died off. In a controlled environment you may begin with carefully placed live cells and monitor the patterns that emerge to model some other natural phenomena.

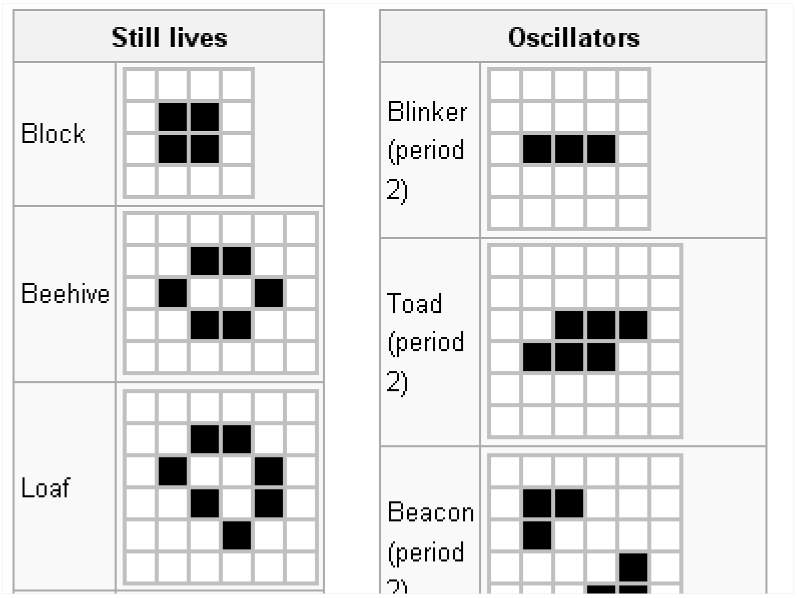


Figure: Common Game of Life Surviving and Oscillating Patterns

A New Kind of Science

The name Stephan Wolfram has been mentioned several times in this post. He is the founder of Wolfram| Research, his company is known for the popular Mathematica software suite and Wolfram|Alpha knowledge engine. He did not initially discover cellular automata but recently he has been a prominent figure in its advocacy. He spent 10 years working on his book, A New Kind of Science. In the 1300 page tome, he discusses how cellular automata can be applied to every field of science from biology to physics. NKA is a detailed study of cellular automata programs.

Basic Cellular Automata

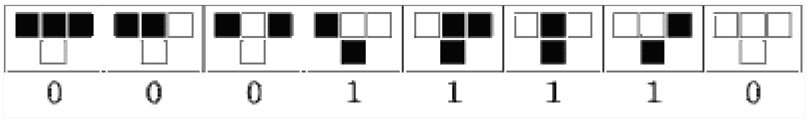


Figure: Wolfram's Elementary CA Rule 30. Look at 3 bit input and 1 bit output.

The diagram above depicts the rule 30 program (or rule 30 elementary cellular automaton). There are 8 input states (2 ^ 3) and an output state of one or zero. If you look at the diagram from left to right. The first sequence of blocks on the left depict an input state of { 1 1 1 } with an output of 0. Given input of cells { 1 1 1}, the output will be set to 0. Subsequently, the next set of blocks consist of an input state of { 1 1 0 } with an output of 0.

Here is python pseudo code for processing rule30 input:

```
def rule30(inputCell_0, inputCell_1, inputCell_1): {
  if inputCell_0 == 1 and inputCell_1 == 1 and inputCell_2 == 1
    return 0:
  else if inputCell_0 == 1 and inputCell_1 == 1 and inputCell_2 == 0:
    return 0:
  ...
}
grid = new Grid(100, 100)
grid[row0, col50] = 1 # Enable first cell on row zero
for j until 100:
  for i until 100:
    valsForNextRow[i] = rule30(inputLastRow[i - 1], inputLastRow[i], inputLastRow[i
+ 1])
```

```
Example of first three cases using a boolean notation:
{ 1 1 1 } -> 0
{ 1 1 0 } -> 0
{ 1 0 1 } -> 0
...
Example of first few cases with Scala programming language:

class CellularAutomataRule extends Rule {
 def rule(inputState:(Int, Int, Int)) : Rules.Output =
   inputState match {
  case (1, 1, 1) => 0
  case (1, 1, 0) => 0
                case (1, 0, 1) => 0
                case (1, 0, 0) => 1
                ...
 }
} // End of Rule
```

Figure: Scala Example with pattern matching

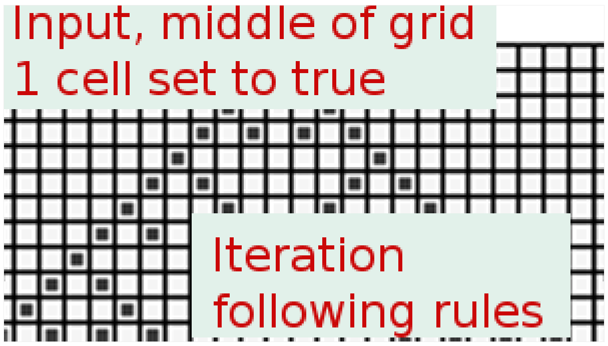


Figure: Elementary Automata Grid after several iterations, look at image from top to bottom

Cellular Automata and Squaring Application

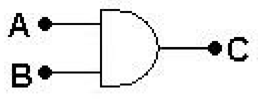
How do you square two numbers?

With most popular programming languages you could use infix notation providing an input parameter on the left and an input parameter on the right side of some arithmetic function. With Java, you might write the following code:

```
int x = 4 * 4;
Output : 16
```

The above snippet is valid code used to multiply four times four with a result of sixteen but it does not say much about the native implementation of the multiplication operator. There are many layers involved with that particular function but they aren't visible to the developer. Is the function implemented and optimized by the compiler or implemented by the runtime environment? It is possible that the operating system may cache the result or build an implementation for the arithmetic operation. Ultimately for most basic integer multiplication or addition, those operations are performed at the hardware level. So how then does the hardware do it?

In the figure depicted below is an AND gate and truth table, the gate takes two Boolean input values and returns the output AND operation. If one is entered in input A and zero is entered into input B, then the output C returned by the AND gate is one. An arithmetic logic unit may perform basic Boolean operations or possibly some form of basic arithmetic. An ALU may consist of AND, XOR and other similar simple gates combined to ultimately perform basic arithmetic, increment, decrement or jump operations. (Most of my comments focus on older generation basic circuits, modern circuit design may not use such techniques or basic components)



A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Figure: Boolean AND Gate, InputA, B and Output

If you start from that basic piece of Java code 4 * 4, there are many levels of software and hardware layers that are involved to implement that operation and then return a result.

I wanted to present basic Boolean arithmetic so that you can see how basic rules can lead to more complex patterns and behavior. One two input AND gate will generate a Boolean result. Several million logic circuits may be used to build a complete CPU. You may already be familiar with the Conway's game of life, an initial grid is created with a random number of initial live cells. We can use a simple cellular automata program to square two integers use the rules described in Wolfram's A New Kind of Science. After so many iterations, a common pattern will emerge and that pattern holds the result of N * N. In our squaring example we started with the input number of enabled cells (N = 4) and after so many iterations a pattern emerged that contained the squaring of the input. In many of Wolfram's Elementary rules, a binary sequence is used for input and output. With the general CA squaring rule, an input and output number ranging from 0 to 7 are defined for each cell.

Squaring Rule

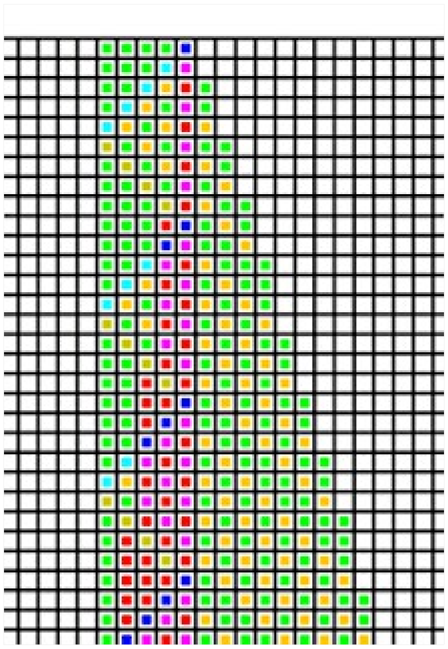


Figure: Applet Visual Output Grid for Squaring Cellular Automata

```
CellularAutomaton[{
 { 0, Blank[], 3} -> 0,
 { Blank[], 2, 3} -> 3,
 { 1, 1, 3 }   -> 4,
 { Blank[], 1, 4} -> 4,
 { Alternatives[1, 2], 3, Blank[]} -> 5,
 { Pattern[$`p, Alternatives[0, 1]], 4, Blank[]} -> 7 - $`p,
 { 7, 2, 6} -> 3,
 { 7, Blank[], Blank[]} -> 7,
 { Blank[], 7, Pattern[$`p, Alternatives[1, 2]]} -> $`p,
 { Blank[], Pattern[$`p, Alternatives[5, 6]], Blank[]} -> 7 - $`p,
 { Alternatives[5, 6], Pattern[$`p, Alternatives[1, 2]], Blank[]} -> 7 - $`p,
 { Alternatives[5, 6], 0, 0} -> 1,
 { Blank[], Pattern[$`p, Alternatives[1, 2]], Blank[]} -> $`p,
 { Blank[],  Blank[], Blank[]} -> 0}, {
 ...
 Append[Table[1, {$CellContext`n$$}], 3], 0},
 Table -> Expression to N
 Append -> Table to 3
```

Figure: Notebook Source File For Mathematica, General CA Rule for Squaring Automaton

The general rules for the squaring automaton are similar to the rules that were mentioned for the elementary rule30 program. Integer values (range 0 - 7) are used instead of binary inputs and outputs. The initial row and initial number of cells are represented by the input parameter (N = 4 in our example).

```
Example Row: 0 0 0 0 0 3 3 3 3 1 0 0 0 0
```

Besides the first row, the initial grid contains all zeros. On the next sequence, the CA rule for squaring is run against each cell on the second row. On the sequence after that, the CA rule is run against the third row and so on until the last row in the grid has been reached. With a 100 x 100 grid, the output pattern will emerge before row 100 is reached.

```
class SquaringRule extends Rules.GeneralRule {
 def ruleId() = 132
 def rule(inputState:Rules.RuleInput) : Rules.Output =
   inputState match {
  case (0, _, 3) => 0
  case (_, 2, 3) => 3
  case (1, 1, 3) => 4
  case (_, 1, 4) => 4
  case (1 | 2, 3, _) => 5
  case (0 | 1, 4, _) => 7 - inputState._1
  case (7, 2, 6) => 3
  case (7, _, _) => 7
  case (_, 7, 1 | 2) => inputState._3
  case (_, 5 | 6, _) => 7 - inputState._2
  case (5 | 6, 1 | 2, _) => 7 - inputState._2
  case (5 | 6, 0, 0) => 1
  case (_, 1 | 2, _) => inputState._2
  case _   => 0
 }
} // End of Rule
```

Figure: Scala Source for Squaring Rule uses Pattern Matching

Applied Cellular Automata

Cellular automata is often used with data compression, cryptography, artificial intelligence, urban planning, financial market modeling, music generation, and 3D terrain generation. If you are a software engineer, you may have to step back and consider how cellular automata patterns emerge and understand the nature of the dynamic system before looking for a typical software library. CA is not normally seen in everyday applications. Consider this when you look at some random pattern, don't think of the phenomenon as a random sequence of events that cannot be replicated, think of the event in terms of a cellular automaton. Try to imagine the rules that could model that natural behavior. Modeling seemingly random patterns is an area where cellular automata is being widely used. Urban planning departments are integrating geographic information systems (GIS) with cellular automata in an attempt to predict growth in an area of a city.

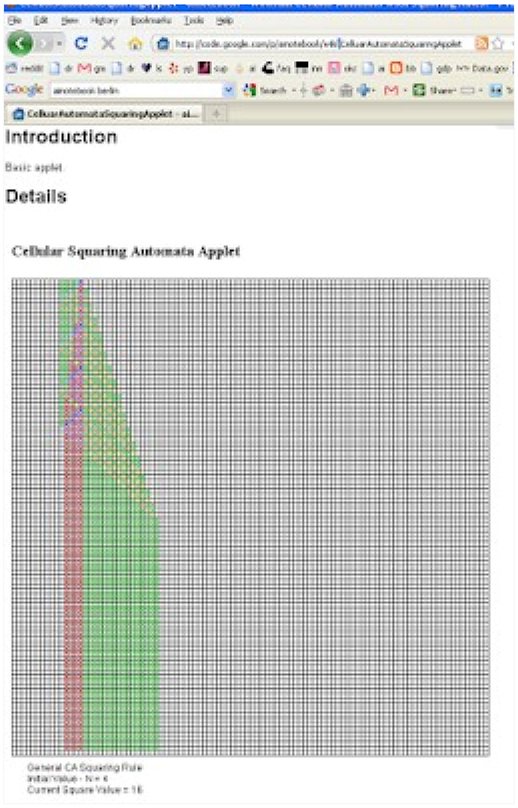
Summary

The simple squaring example mentioned in this post merely gives you an overview of a basic cellular automata system. Scientists, biologists, computer scientists and software engineers want to find better ways to observe relationships and patterns that occur in our world. Review Stephen Wolfram's A New Kind of Science to give you an idea for what is possible with seemingly simple rules.

Source Code and Applet

1. doingitwrongnotebook/wiki/CelluarAutomataSquaringApplet
2. SVN source repository directory
3. CelluarAutomataApplet - Test of Elementary Rules
4. Game of Life Applet

5. Full Download Applet Examples (keywords: Scala, Rule30, Rule190, GameOfLife, Wolfram Squaring Rule)


Introduction

Basic applet

Details

Cellular Squaring Automata Applet

General CA Squaring Rule
Initial Value - N = 4
Current Square Value = 16

Figure: Squaring Cellular Automaton Output, Input = 4 (top of grid), Output = 16 (pattern towards the bottom)

Simple Lisp

Overview
I have never had much affinity for writing (in natural language); I have grown up and enjoy programming and that is what I continue to do 20 years later. But I really wanted to get this blog entry out and it helped me rethink and reframe how I look at a particular piece of code. So I am going to move in all different directions starting with one of the most elementary data structures in Computer Science, the linked list. Instead of thinking of Lisp code (for now anyway) as a "listing" of elements. Think of Lisp code as a linked list of elements.
•*How do you implement a linked list in C (or Java or Fortran)?*

•*How would you print all of the elements of the linked list?*

•*How would you sum all of the elements?*

After thinking about the last numeric operation? Can you think of possible ways to implement various List operations, Lisp operations like '**map**'?

The Linked List

Why are linked lists important to Lisp? "The name Lisp derives from "List Processing Language". Linked lists are one of Lisp languages' major data structures"

In a Java/C/C++ or other procedural language program, the linked list is normally composed of two fields, the data itself (composed of any Object type) and a "pointer" or reference to the next element A list that is linked. In the Java code snippet below, data refers to the element at the current node in the list. The Node object "next" is a reference to the next node in the list chain.

```java
package org.bot.jscheme.linkedlist;
/**
 * A Node has two fields, data and next (or car and cdr).
 * This forms the basis of a simple, singly-linked list structure
 * whose contents can be manipulated with cons, car, and cdr.
 */
public class Node {
    /**
     * The first element of the pair.
     */
    public Object data;
    /**
     *  The other element of the pair.
     */
    public Node next;
    /** Build a pair from two components. * */
    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }
    /**
     * Return a String representation of the pair.
     */
    public String toString() {
        return "" + data;
    }
}
```

The linked list Java class is also as simple as the Node class. There is one field "head" which is a Node type. The head is the first Node in the list chain. The insert at head operation inserts a newly created Node object. It replaces the head Node with the new node and the head's next node becomes the previous node. For example, the code below contains a complete implementation.

```java
package org.bot.jscheme.linkedlist;
public class LinkedList {
    /**
     * Head element in linked list.
     */
    public Node head;
    private LinkedList() { }

    public LinkedList(Node head) {
        this.head = head;
    }
    public LinkedList(String data) {
        this.head = new Node(data, null);
    }
    /**
     * Add a new node and replace the "root" node with the new one.
     *
     * @param node
     * @return
     */
    public LinkedList insertHead(final Node node) {
        node.setNext(head);
        head = node;
        return this;
    }
    public void insertTail(final Node node) {
        if (head == null) {
            head = node;
        } else {
            Node p, q;
            // Traverse to the end of the list
            for (p = head; (q = p.getNext()) != null; p = q) {
                ;
            }
            p.setNext(node);
        }
    }
}
```

The insertTail operation is more inline with the lisp "cons" constructor operation. Traverse to the last Node in the list chain and set the next node reference to the new Node we want to add to the end of the chain.

If we needed to print the data at each element in the Node, we create a reference to the head node and then iterate to each subsequent "next" Node until the end Node contains a null reference.

```
public String traverse() {
        // Append the contents to a string
        StringBuffer buf = new StringBuffer();
        buf.append("(");
        buf.append(this.getHead());
        for (Node node = this.getHead().getNext(); node != null; node =
node.getNext()) {
            buf.append(' ');
            buf.append(node.getData());
        }
        buf.append(")");
        return buf.toString();
    }
```

Once we start getting into the Lisp functionality we will cover the List operations in more detail but here is another traversal example that is similar to the Lisp lisp Pair to string methods. The other difference between the Lisp implementation and this code are the member names.

```
public String stringifyNode(StringBuffer buf) {
        buf.append('(');
        buf.append(data);
        Object tail = this.getNext();
        while (tail instanceof Node) {
            buf.append(' ');
            buf.append(((Node) tail).getData());
            tail = ((Node) tail).getNext();
        }
        if (tail != null) {
            buf.append(" . ");
            ((Node) tail).stringifyNode(buf);
        }
        buf.append(')');
        return buf.toString();
```

```
    }
```

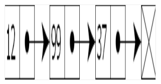


Figure: singly-linked list containing two values: the value of the current node and a link to the next node

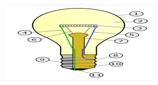
Linked List Test Case

Here are a set of tests for our Linked List. The toString method for our LinkedList returns a String composed of a Lisp symbolic expression left parenthesis, list arguments and a right parenthesis.

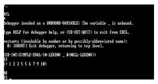
```
    public static void main(String [] args) {
        System.out.println("Running linked list test");

        final LinkedList list = new LinkedList("+");
        list.assertEquals("" + list, "(+)");

        // Start off backwards; insert using insertHead.
        list.insertHead(new Node("1", null));
        list.insertHead(new Node("2", null));
        list.insertHead(new Node("3", null));
        list.assertEquals("" + list, "(3 2 1 +)");
        list.testStatus();
        list.assertEquals(
                "(3 2 1 +)",
                list.getHead().stringifyNode());

        final Node node = new Node("+", new Node("1", null));
        list.assertEquals(
                "(+ 1)",
                node.stringifyNode());

        final Node node2 = new Node("+", new Node("1", new Node("2", null)));
        list.assertEquals(
                "(+ 1 2)",
                node2.stringifyNode());

        final Node node3 = new Node("+", new Node("1", new Node("2", null)));
        final LinkedList list2 = new LinkedList(node3);
        list.assertEquals(
                "(+ 1 2)",
                list2.getHead().stringifyNode());
        list.assertEquals(
                "(+ 1 2)", "" + list2);

        final LinkedList list3 = new LinkedList("+");
        list3.assertEquals("" + list3, "(+)");
        list3.insertTail(new Node("1", null));
        list3.insertTail(new Node("2", null));
        list3.insertTail(new Node("3", null));
        list.assertEquals("" + list3, "(+ 1 2 3)");
        list.testStatus();

        final LinkedList list4 = new LinkedList("1");
        list4.insertTail(new Node("2", null));
        list4.insertTail(new Node("3", null));
        list4.insertTail(new Node("4", null));
        list.assertEquals("" + list4, "(1 2 3 4)");
        list.assertEquals("" + list4.numCompute('+'), "10.0");
        list.assertEquals("" + list4.numCompute('*'), "24.0");
    }
```

Ding, Ding, Ding, *DING*

You may have noticed that I left out some implementation details on some of the methods used in the test case. They aren't totally relevant, use the source download at the bottom of the page to get all of the source. NumCompute is a relevant method and leads us into our Lisp implementation in Java. Essentially, the numCompute method is a traversal method to sum all of the Nodes in the list chain. This is kind of important, Lisp is a List processing language. Our traversal method is an operation on a singly linked list. I hope the light-bulb is going off for you.



```
    /**
     * Perform an arithmetic operation on the List.
     * @param operation
     * @return
     */
    public double numCompute(final char operation) {
        double result = 0.0;
        if (operation == '+' || operation == '-') {
            result = 0.0;
        } else if (operation == '*' || operation == '/') {
            result = 1.0;
        } else {
            throw new RuntimeException("Invalid Operation: try +, -, *, /");
        }
        for (Node node = this.getHead(); node != null; node = node.getNext()) {
            Double d = new Double((String) node.getData());
            double x = d.doubleValue();
            switch (operation) {
            case '+':
                result += x;
                break;
            case '-':
                result -= x;
                break;
            case '*':
                result *= x;
                break;
            case '/':
                result /= x;
                break;
            default:
                throw new RuntimeException("Invalid Operation: try +, -, *, /");
            }
        }
        return result;
    }
```



```
*_
NIL
*
debugger invoked on a UNBOUND-VARIABLE: The variable _ is unbound.

Type HELP for debugger help, or (SB-EXT:QUIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [ABORT] Exit debugger, returning to top level.

(SB-INT:SIMPLE-EVAL-IN-LEXENV _ #<NULL-LEXENV>)
0]
(+ 1 2 3 5 5 6 7 9 10)

48
0] _
```

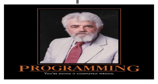
Figure: SBCL prompt with the Lisp call (+ 1 2 3 5 5 6 7 9 10)

```
The lisp code:
(+ 1 2 3 4)
```

Is essentially a representation of the following Java code:

```
final LinkedList list4 = new LinkedList("1");
list4.insertTail(new Node("2", null));
list4.insertTail(new Node("3", null));
list4.insertTail(new Node("4", null));
list4.numCompute('+')
```

On Lisp



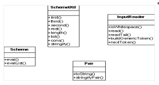
"On LISP's approximate 21st anniversary, no doubt something could be said about coming of age, but it seems doubtful that the normal life expectancy of a programming language is three score and ten. In fact, LISP seems to be the second oldest surviving programming language after Fortran, so maybe we should plan on holding one of these newspaper interviews in which grandpa is asked to what he attributes having lived to 100." -- John McCarthy

I don't want to delve too much into Lisp history or even the pros and cons; essentially Lisp has a rich history and is a very simple, powerful language. Lisp programs are represented by symbolic expressions as a simple linked list structure that is stored memory. We can determine the lisp function by the first ATOM in a list and perform that operation on the rest of the arguments. And there are a small set of selector and constructor operations expressed as functions (car, cdr and cons).

Lisp in Java, Mini JScheme

Figure: Picture of Peter Norvig, inventor of JScheme

Peter Norvig created JScheme, a Lisp Scheme dialect written in Java. It was created in 1998 and it has since been forked, extended and used quite extensively in other modern projects for the web and elsewhere. I mention JScheme because the code I am about present is all based on the JScheme interpreter. The code is copyrighted to Peter Norvig (including the picture of him above), I have only made modifications to make our Lisp even lighter than it was originally. Norvig's most recent implementation of JScheme contains 1905 lines of Java code. My implementation contains 905 lines wrapped in additional Javadoc comments.



There are eight classes in this smaller implementation. The core classes are shown in the UML class diagram. The InputReader is the most interesting class; if you are interested in learning about basic character/token parsing routines, you should look at this code. The read method is the first core method invoked. It is needed to read each character from the InputStream (E.g. an implementation of a FileInputStream) and return the tokens to the eval method.

```
    public Object read() {
        try {
            Object token = nextToken();
            if (token == "(") {
                return readTail();
            } else if (token == ")") {
                System.out.println("WARN: Extra ')' ignored.");
                return read();
            } else {
                return token;
            } // End of the if - else
        } catch (IOException e) {
            System.out.println("WARN: On input, exception: " + e);
            return EOF;
        } // End try - catch
    }
    private Object readTail() throws IOException {
        Object token = nextToken();
        System.out.println("trace: readTail(): " + token);
        if (token == EOF) {
            final String msg = "ERROR: readTail() - EOF during read.";
            System.err.println(msg);
            throw (new RuntimeException(msg));
        } else if (token == ")") {
            return null;
        } else if (token == ".") {
            Object result = read();
            token = nextToken();
            if (token != ")") {
                System.out.println("WARN: Missing ')'? Received " + token + "
after .");
            }
            return result;
        } else {
            tokenStack.push(token);
            return SchemeUtil.cons(read(), readTail());
        }
    }
    private Object nextToken() throws IOException {
        int ch;
        // See if we should re-use a pushed char or token
        // Task 1: Pop the token and character stacks
        if (!this.tokenStack.empty() && (this.tokenStack.peek() != null)) {
            return this.tokenStack.pop();
        } else if (!this.charStack.empty() && (this.charStack.peek() != null)) {
            ch = ((Character) this.charStack.pop()).charValue();
        } else {
            ch = inputReader.read();
        }
        // Ignore whitespace
        // Task 2: Check for and ignore whitespace
        while (isWhitespace((char) ch)) {
            ch = inputReader.read();
        }
        System.out.println("trace: nextToken() -> " + (char) ch + " $" + ch);
        // See what kind of non-white character we got
        // Task 3: Check if the character is of various token types.
        switch (ch) {
            case -1:
                return EOF;
            case TOK_LEFT_PAREN:
                return "(";
            case TOK_RIGHT_PAREN:
                return ")";
            ...
            ...
            default:
                buff.setLength(0);
                int c = ch;
                buildGenericToken(ch);
                // Try potential numbers, but catch any format errors.
                if (c == '.' || c == '+' || c == '-' || (c >= '0' && c <= '9')) {
                    try {
                        // Number type is currently in the buffer queue
                        return new Double(buff.toString());
                    } catch (NumberFormatException e) {
                        ;
                    }
                } // End of If
                return buff.toString().toLowerCase();
        } // End of the Switch
    }
```

Linked Lists, Cons and Pairs,

In that cluster of code, this small snippet should stick out in your mind:

```
readTail:
tokenStack.push(token);
return SchemeUtil.cons(read(), readTail());
```

Invocation of our cons operator. Cons, short for the term constructor, returns a new Pair object based on some data Node and possibly some pointer to another Pair. Notice that I am interchanging the term Pair and Node; where Node is the class in our LinkedList implementation. Essentially, a Pair is a Node and most of our Scheme processing is some variation of the LinkedList operations.

```
    /**
     * cons(x, y) is the same as new Pair(x, y).
     *
     * Cons presents and interesting function that is fundamental to lisp.
     * Here are some examples of cons usage (tested in common lisp).
     * <code>
     * (cons 1 2):
     * Pair pair = SchemeUtil.cons("1", "2");
     * assertEquals("" + pair, "(1 . 2)");
     *
     * (cons 1 nil):
```

```
     * Pair pair = SchemeUtil.cons("1", null);
     * assertEquals("" + pair, "(1)");
     *
     * (cons 1 (cons 2 nil)):
     *
     * Pair pair = SchemeUtil.cons("1", SchemeUtil.cons("2", null));
     * assertEquals("" + pair, "(1 2)");
     *
     * </code>
     */
    public static Pair cons(Object a, Object b) {
        return new Pair(a, b);
    }
///
/// WHAT IS NEW PAIR????
///
/**
 * A Pair has two fields, first and rest (or car and cdr).
 * This forms the basis of a simple, singly-linked list structure
 * whose contents can be manipulated with cons, car, and cdr.
 *
 * "One mathematical consideration that influenced LISP was to
 * express programs as applicative expressions built up
 * from variables and constants using functions" -- John McCarthy
 * http://www-formal.stanford.edu/jmc/history/lisp/node3.html
 */
public class Pair {
    /**
     * The first element of the pair.
     */
    private Object first;
    /**
     *  The other element of the pair.
     */
    private Object rest;
    /** Build a pair from two components. * */
    public Pair(Object first, Object rest) {
        this.first = first;
        this.rest = rest;
    }
    /**
     * Return a String representation of the pair.
     */
    public String toString() {
        return SchemeUtil.stringify(this);
    }
    /**
     * Build up a String representation of the Pair in a StringBuffer.
     */
    void stringifyPair(StringBuffer buf) {
        buf.append('(');
        SchemeUtil.stringify(first, buf);
        Object tail = rest;
        while (tail instanceof Pair) {
            buf.append(' ');
            SchemeUtil.stringify(((Pair) tail).first, buf);
            tail = ((Pair) tail).rest;
        }
        if (tail != null) {
            buf.append(" . ");
            SchemeUtil.stringify(tail, buf);
        }
        buf.append(')');
    }
}
```

The input reader is our IO class for reading from the Stream and returning a PAIR to to the calling method. Once we have a Pair object (essentially a LinkedList structure), we can then perform Lisp operations on the given list elements. For example, here are two Test Case methods:

```
public void testInputReaderPair() {
        ByteArrayInputStream stream =
            new ByteArrayInputStream("(+ 1 2 3)".getBytes());
        InputReader reader = new InputReader(stream);
        Object o = reader.read();
        Pair p = (Pair) o;
        assertEquals("(+ 1.0 2.0 3.0)", "" + p);
    }
    public void testEval1() {
        Environment globalEnvironment = new Environment();
        BuiltInFunction.installBuiltInFunctions(globalEnvironment);
        ByteArrayInputStream stream = new ByteArrayInputStream("(+ 1 2
3)".getBytes());
        InputReader reader = new InputReader(stream);
        Object o = reader.read();
        Scheme scheme = new Scheme();
        Object res = scheme.eval(o, globalEnvironment);
        double d = ((Double) res).doubleValue();
        assertEquals(d, 6.0d, 0.01d);
    }
```

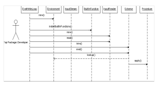
How do we get to performing Lisp operations on Lists?

In the testInputReaderPair test case, we convert a String representation of Lisp code and then create an InputReader object. Invoke the read method to return a Pair object. In this case, Pair is the head node in the LinkedList. The data in this Node is the String representation of the PLUS/ADD/+ operation. The rest of the elements are arguments to the function. The testEval1 test adds one more call to our first test. Eval. The Pair class is in memory, now we actually have to invoke Lisp operations to get any work done. In my example, I want to add those numbers to together to return a sum. The first element in the list is the function, the rest are my arguments.

"Based on the "token" and the parse rule being matched, the parser does something -- push the token on the parse stack, pop the previous token and emit some code, implement some "Semantic Action" based on the tokens and rules matched so far, etc. etc" -- SaveTheHubble (CrazyOnTap.com)

The Scheme.eval method is a driver method where the execution on the Pair begins.

```
    public Object eval(Object x, Environment env) {
        while (true) {
            if (x instanceof String) {
                // VARIABLE
                System.out.println("trace: eval - instance of String: " + x);
                // Look up a variable or a procedure (built in function).
                return env.lookup((String) x);
            } else if (!(x instanceof Pair)) {
                // CONSTANT
                System.out.println("trace: eval - instance of Pair =>" + x);
                return x;
            } else {
                // Procedure Call
                System.out.println("trace: eval[t1] - instance of procedure call");
                Object fn = SchemeUtil.first(x);
                Object args = SchemeUtil.rest(x);
                System.out.println("trace: eval[t2] - fn => [" + fn + "] args => " +
args);
                fn = eval(fn, env);
                // Coerce the object to a procedure (E.g. '+/add' procedure)
                Procedure p = Procedure.proc(fn);
                return p.apply(this, evalList(args, env));
            } // End of if - else
        } // End of While
    } // End of Method
```



What about the environment?

The environment class is essentially a Hashtable data-structure for storing variable definitions. The key represents the name of the function and the value is an instance of the BuiltInFunction class. Other user defined variables are also stored here (this feature is not implemented in MiniJScheme). There is a major variation from Norvig's JScheme implemention of the Environment class. He used a LinkedList structure to store all of the variables. To lookup a variable by name, you traverse the list until the variable/function is found. I used simple Hashtable get and put operations. It is the least interesting of all of the classes, but probably the most important. If you don't have a reference to the ADD operation, then you can't a sum on a set of numbers.

```
public class Environment {

    private Map mapDataVars = new HashMap();

    public Environment() {
    }
    /**
     * Find the value of a symbol, in this environment or a parent.
     */
    public Object lookup(String symbol) {
        Object o = this.mapDataVars.get(symbol);
        if (o != null) {
            return o;
        } else {
            return SchemeUtil.error("Unbound variable: [" + symbol + "]");
        }
    }
    /** Add a new variable,value pair to this environment. */
    public Object define(Object var, Object val) {

        this.mapDataVars.put(var, val);
        if (val instanceof Procedure
                && ((Procedure) val).getName().equals(Procedure.DEFAULT_NAME)) {
            ((Procedure) val).setName(var.toString());
        }
        return var;
    }
    public Environment defineBuiltInProc(String name, int id, int minArgs) {
        define(name, new BuiltInFunction(id, minArgs, minArgs));
        return this;
    }
}
```

Installing/Loading the Built-In Functions

We are finally at the point to talk about calling Lisp functions. Here are some basic "Scheme" operations. The cons/constructor(new Pair) operation should be pretty straightforward. Same for the "first" function. I didn't include the code for numCompute because the implementation is the same as the LinkedList code shown earlier.

```
public static Environment installBuiltInFunctions(Environment env)  {
        int n = Integer.MAX_VALUE;
        env

         .defineBuiltInProc("cons",     CONS,      2)

         .defineBuiltInProc("*",        TIMES,     0, n)
         .defineBuiltInProc("+",        PLUS,      0, n)
         .defineBuiltInProc("-",        MINUS,     1, n)
         .defineBuiltInProc("/",        DIVIDE,    1, n)

         .defineBuiltInProc("caaaar",         CXR,       1)
         .defineBuiltInProc("caar",           CXR,       1)
         .defineBuiltInProc("cadr",         SECOND,      1)
         .defineBuiltInProc("cdar",           CXR,       1)
         .defineBuiltInProc("cddr",           CXR,       1)

         .defineBuiltInProc("car",            CAR,       1)
         .defineBuiltInProc("cdr",            CDR,       1)

         .defineBuiltInProc("*",        TIMES,     0, n)
         .defineBuiltInProc("+",        PLUS,      0, n)
         .defineBuiltInProc("-",        MINUS,     1, n)
         .defineBuiltInProc("/",        DIVIDE,    1, n);
        return env;
    }
/**
     * Apply a primitive built-in-function to a list of arguments.
     */
    public Object apply(Scheme interp, Object args) {
        // First make sure there are the right number of arguments.
        int nArgs = SchemeUtil.length(args);
        if (nArgs < minArgs) {
            return SchemeUtil.error("too few args, " + nArgs + ", for " +
this.getName() + ": "
                    + args);
        } else if (nArgs > maxArgs) {
            return SchemeUtil.error("too many args, " + nArgs + ", for " +
this.getName()
                    + ": " + args);
        } // End of the If
        Object x = SchemeUtil.first(args);
        Object y = SchemeUtil.second(args);
        switch (idNumber) {
            case PLUS:
                return numCompute(args, '+', 0.0);
            case MINUS:
                return numCompute(SchemeUtil.rest(args), '-', num(x));
            case TIMES:
                return numCompute(args, '*', 1.0);
            case DIVIDE:
                return numCompute(SchemeUtil.rest(args), '/', num(x));
            case THIRD:
                return SchemeUtil.third(x);
            case CONS:
                return SchemeUtil.cons(x, y);
            case CAR:
                return SchemeUtil.first(x);
            case CDR:
                return SchemeUtil.rest(x);
            case CXR:
                for (int i = this.getName().length() - 2; i >= 1; i--) {
                    x = (this.getName().charAt(i) == 'a') ? SchemeUtil.first(x) :
SchemeUtil.rest(x);
                }
                return x;
            default:
                return SchemeUtil.error("internal error: unknown primitive: " + this
                        + " applied to " + args);
        }
    } // End of the Apply
```

More Test Cases

Here are some more Java test cases and their Scheme/Common Lisp equivalent:

```
public class TestPairList extends TestCase {

    public static final String test1 = "(cons 1 2)";
    public static final String test2 = "(cons 1 (cons 2 ()))";

    public void testBasicBuildPair() {
        // Construct a linked list/pair of
        // a function '+', 1 and 2
        // (+ 1 2)
        Pair pair = new Pair("+", new Pair("1", new Pair("2", null)));
        assertEquals("" + pair, "(+ 1 2)");
    }
    public void testPair1() {
        // Construct a linked list/pair of
        // a function '+', 1 and 2
        // (+ 1 2)
        Pair pair = new Pair("+", new Pair("1", new Pair("2", ":test")));
        assertEquals("" + pair, "(+ 1 2 . :test)");
    }
    /**
     * (cons 1 2).
     */
    public void testCons() {
        Pair pair = SchemeUtil.cons("1", "2");
        assertEquals("" + pair, "(1 . 2)");
    }
    /**
     * (cons 1 nil).
     */
    public void testConsOne() {
        Pair pair = SchemeUtil.cons("1", null);
        assertEquals("" + pair, "(1)");
    }
    /**
     * (cons 1 (cons 2 nil)).
     */
    public void testConsTwo() {
        Pair pair = SchemeUtil.cons("1", SchemeUtil.cons("2", null));
        assertEquals("" + pair, "(1 2)");
    }

    public void testEvalCons1() {
        Environment globalEnvironment = new Environment();
        BuiltInFunction.installBuiltInFunctions(globalEnvironment);
        ByteArrayInputStream stream = new ByteArrayInputStream(test1.getBytes());
        InputReader reader = new InputReader(stream);
        Object o = reader.read();
        Scheme scheme = new Scheme();
        Object res = scheme.eval(o, globalEnvironment);
        System.out.println(res);
    }
    public void testEvalCons2() {
        Environment globalEnvironment = new Environment();
        BuiltInFunction.installBuiltInFunctions(globalEnvironment);
        ByteArrayInputStream stream = new ByteArrayInputStream(test2.getBytes());
        InputReader reader = new InputReader(stream);
        Object o = reader.read();
        Scheme scheme = new Scheme();
        Object res = scheme.eval(o, globalEnvironment);
        assertEquals("" + res, "(1.0 2.0)");
    }
}
```

Conclusion

This concludes my run down of a Scheme/Lisp implementation in Java starting from Linked List data structures and continuing to the interpreter code. There is a lot that can be added to Norvig's implementation. Add full support for a R5RS Scheme Implementation. Add better input/output support. Copy interesting functions from Arc or Common Lisp. Write a Scheme compiler. In fact, the source code in my implementation and Norvig's is fairly straightforward, I was planning on writing a Forth interpreter based on some of this code. Lisp's defining data-structure is the List. Forth's is the stack. It won't be nearly as complete as the Factor programming language, but it would still be a fun project.