Java Garbage Collection

All Java objects are stored on the heap (as opposed to the stack).

Garbage collection: process of automatically freeing heap objects no longer referenced by the program:

Step 1. Detect garbage objects:

- Define a set of *roots* (on the stack) and determining *reachability* from the roots.
- An object is reachable if there is some path of references from the roots by which the executing program can access the object.
- The roots are always accessible to the program. Any objects that are reachable from the roots are considered *live*.
- Objects that are not reachable are considered garbage, because they can no longer affect the future course of program execution

2. Reclaim the heap space used by garbage objects and make it available to the program.

Tracing collectors

- trace out the graph of object references starting with the root nodes.
- Objects that are encountered during the trace are marked in some way.
- After the trace is complete, unmarked objects are known to be unreachable and can be garbage collected.

The basic tracing algorithm is called *mark and sweep*.

- In the mark phase, the garbage collector traverses the tree of references and marks each object it encounters.
- In the sweep phase unmarked objects are freed, and the resulting memory is made available to the executing program.

Most collectors are “stop-the-world”.

GC frequency increase as heap fills up.

GC takes over application time, leading to performance problems. On servers, response time becomes too long, and CPUs are idle. JVMs up through 1.3.1 do not have parallel garbage collection:

=> an application that spends only 1% of the time in garbage collection on a uniprocessor system. has a 20% loss in throughput on 32 processor systems.

=> small improvements in garbage collection can produce large gains in performance.

Copying garbage collectors: use a strategy to combat heap fragmentation**.**

- Copying garbage collectors move all live objects to a new area.
- As the objects are moved to the new area, they are placed side by side, thus eliminating any free spaces that may have separated them in the old area. The old area is then known to be all free space.
- The advantage of this approach is that objects can be copied as they are discovered by traversals from the root nodes. There are no separate mark and sweep phases. Objects are copied to the new area on the fly.

A common copying collector is called *stop and copy*.

- heap is divided into two regions.
- Objects are allocated from one of the regions until space in that region has been exhausted.
- program execution is stopped and the heap is traversed. Live objects copied to other region
- When stop and copy is finished, program execution resumes. Memory allocated from new heap region until it too runs out of space … repeat with role of regions reversed.

Cost: twice as much memory is needed for a given amount of heap space because only half of the available memory is used at any time.

The default Java garbage collector will be adequate for the majority of applications.

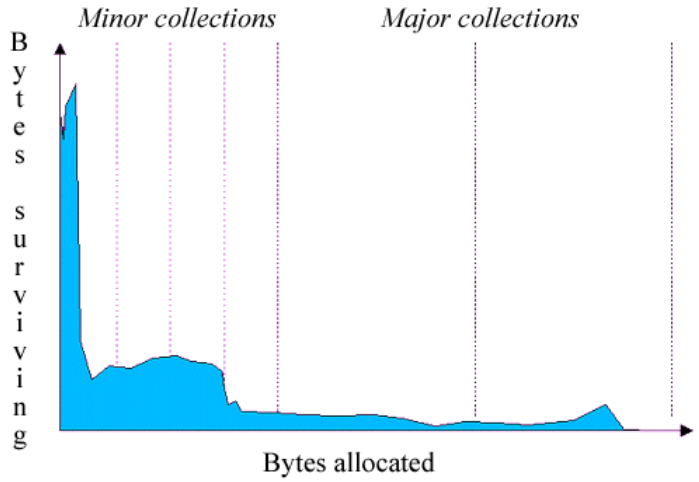
Other collectors perform better for special application behaviors or for hardware with a large amount of memory and a large number of processors.

Generations

Naive garbage collection algorithms simply iterate over every reachable object. Takes time proportional to the number of live objects, which may be very large.

Generational collection exploits properties of applications to avoid extra work.

Infant mortality: majority of objects "die young".



Memory is managed in *generations* holding objects of different ages.

Garbage collection occurs in each generation when the generation fills up. Objects are allocated in the *young* generation, and most (2/3) objects die there.

When the *young* generation fills up it causes a *minor collection:*

- Cost proportional to the number of live objects being collected. A *young* generation full of dead objects is collected very quickly.
- Some surviving objects are moved to an *tenured* generation. When the *tenured* generation needs to be collected there is a *major collection* that is often much slower because it involves all live objects.

In the diagram above:

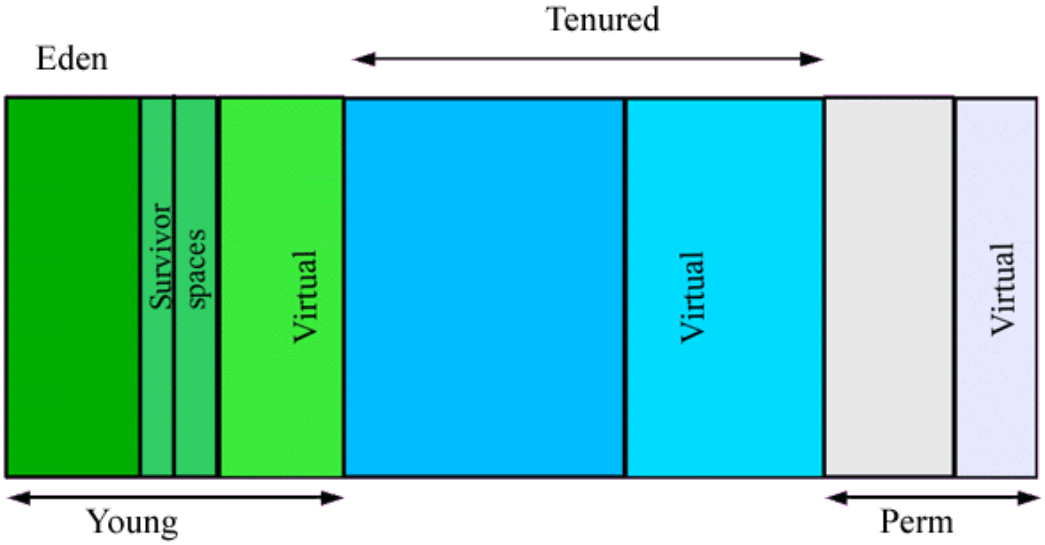
- *minor collections* occurring at intervals long enough to allow many of the objects to die between collections.
- the young generation is large enough (and thus the period between minor collections long enough) that the minor collection can take advantage of the high infant mortality rate.

At initialization, a maximum address space is virtually reserved but not allocated to physical memory unless it is needed.

The *young* generation consists of *eden* plus two *survivor* spaces .

- Objects are initially allocated in *eden*.
- One *survivor* space is empty at any time, and serves as a destination of the next, copying collection of any live objects in *eden* and the other survivor space.
- Objects are copied between survivor spaces in this way until they old enough to be tenured, or copied to the *tenured* generation.

The permanent generation holds all the reflective data of the virtual machine itself, such as class and method objects.



Performance Considerations

Throughput is the percentage of total time not spent in garbage collection, considered over long periods of time.

Pauses are the times when an application appears unresponsive because garbage collection is occurring.

Footprint is the working set of a process, measured in pages and cache lines

Tradeoffs.

- large *young* generation : maximize throughput, but higher footprint and pause times.
- small young generation : *minimize pause* but lower throughput.

Measurement

Command line argument -verbose:gc prints information at every collection:

```
[GC 325407K->83000K(776768K), 0.2300771 secs]
[GC 325816K->83372K(776768K), 0.2454258 secs]
[Full GC 267628K->83769K(776768K), 1.8479984 secs]
```

where:

- 325407K->83000K (in the first line): combined size of live objects before and after garbage collection,
- (776768K): total available space, not counting the space in the *permanent* generation = total heap - one of the survivor spaces.
- 0.2300771 secs: time for garbage collection

Command line argument -XX:+PrintGCDetails:

```
[GC [DefNew: 64575K->959K(64576K), 0.0457646 secs]
    196016K->133633K(261184K), 0.0459067 secs]]
```

where:

- DefNew: 64575K->959K(64576K) : minor collection recovered 98% of *young* generation
- 0.0457646 secs: taking 46 milliseconds.
- 196016K->133633K(261184K): The usage of the entire heap was reduced to about 51%

Sizing the Generations

At initialization of the virtual machine, the entire space for the heap is reserved.

-Xmx option: max size of space reserved for heap.

-Xms: min size.

If Xms < Xmx, not all of the space that is reserved is immediately committed to the virtual machine. The uncommitted space is labeled "virtual" in this figure.

Different parts of the heap (*permanent* generation, *tenured* generation, and *young* generation) **can grow to the limit of the virtual space as needed.**

Total Heap
Since collections occur when generations fill up, throughput is inversely proportional to the amount of memory available.

Total available memory is the most important factor affecting garbage collection performance.

By default, the virtual machine grows or shrinks the heap at each collection to try to keep the proportion of free space to live objects at each collection within a specific range.

This target range is set as a percentage by the parameters -XX:MinHeapFreeRatio=<minimum> and -XX:MaxHeapFreeRatio=<maximum>, and the total size is bounded below by -Xms and above by -Xmx .

Default parameters for Solaris:

-X:MinHeapFreeRatio=	40
-X:MaxHeapFreeRatio=	70
-Xms	3670k
-Xmx	64m

- If the percent of free space in a generation falls below 40%, the size of the generation will be expanded so as to have 40% of the space free, assuming the size of the generation has not already reached its limit.
- If the percent of free space exceeds 70%, the size of the generation will be shrunk so as to have only 70% of the space free as long as shrinking the generation does not decrease it below the minimum size of the generation.

The *Young* Generation
The bigger the *young* generation, the less often minor collections occur.

However, for a bounded heap size a larger *young* generation implies a smaller *tenured* generation, which will increase the frequency of major collections.

Choice depends on the lifetime distribution of the objects allocated by the application.

The *young* generation size is controlled by NewRatio.

Example: setting -XX:NewRatio=3 means that the ratio between the *young* and *tenured* generation is 1:3. In other words, the combined size of the eden and survivor spaces will be one fourth of the total heap size.

Types of Collectors

Throughput collector:

- -XX:+UseParallelGC
- uses a parallel (many threads) version of the *young* generation collector.
- stop-the-world
- Larger young generations, and smaller old generations
- The *tenured* generation collector is the same as the default collector.

Concurrent low pause collector:

- -XX:+UseConcMarkSweepGC.
- The concurrent collector is used to collect the *tenured* generation and does most of the collection concurrently with the execution of the application. The application is paused for short periods during the collection. (One thread for GC, others for application.)
- A parallel version of the *young* generation copying collector is used with the concurrent collector (i.e. same as UseParNewGC).
- Use this if your application would benefit from shorter garbage collector pauses and can afford to share processor resources with the garbage collector when the application is running. (Smaller young generations and larger older generations.)

Incremental (sometimes called *train*) low pause collector:

- -Xincgc
- the incremental garbage collector collects just a portion of the *tenured* generation at each minor collection, trying to spread the large pause of a major collection over many minor collections (by doing portions of the major collection work at each minor collection.)
- slower than the default *tenured* generation collector when considering overall throughput.
- Use when your application can afford to trade longer and more frequent *young* generation garbage collection pauses for shorter *tenured* generation pauses. (Situations in which a larger *tenured* generation is required (lots of long-lived objects), a smaller *young* generation will suffice (most objects are short-lived and don't survive the *young* generation collection), and only a single processor is available.

Garbage Collection extensions in J2SE 5.0.

Server class machines = ≥ 2 CPUs, >= 2 Gbytes

1. On server-class machines running the **server VM**, the garbage collector (GC) has changed from the previous serial collector (-XX:+UseSerialGC) to a parallel collector (-XX:+UseParallelGC). You can override this default by using the -XX:+UseSerialGC command-line option to the java command.

2. On server-class machines running **either VM** (client or server) with the parallel garbage collector (-XX:+UseParallelGC) the initial heap size and maximum heap size have changed as follows.

   initial heap size:
   Larger of 1/64th of the machine's physical memory on the machine or some reasonable minimum. You can override this default using the -Xms command-line option.

   maximum heap size:
   Smaller of 1/4th of the physical memory or 1GB. Before J2SE 5.0, the default maximum heap size was 64MB. You can override this default using the -Xmx command-line option.

The implementation of -XX:+UseAdaptiveSizePolicy used by default with the

-XX:+UseParallelGC garbage collector has changed to consider three goals:

- a desired maximum GC pause goal
- a desired application throughput goal
- minimum footprint

The implementation checks (in this order):

1. If the GC pause time is greater than the pause time goal then reduce the generations sizes to better attain the goal.
2. If the pause time goal is being met then consider the application's throughput goal. If the application's throughput goal is not being met, then increase the sizes of the generations to better attain the goal.
3. If both the pause time goal and the throughput goal are being met, then the size of the generations are decreased to reduce footprint.

Flags

-XX:MaxGCPauseMillis=*nnn*

A hint to the virtual machine that pause times of *nnn* milliseconds or less are desired. The VM will adjust the java heap size and other GC-related parameters in an attempt to keep GC-induced pauses shorter than *nnn* milliseconds. Note that this may cause the VM to reduce overall throughput, and in some cases the VM will not be able to meet the desired pause time goal.

By default there is no pause time goal. There are definite limitations on how well a pause time goal can be met. The pause time for a GC depends on the amount of live data in the heap. The minor and major collections depend in different ways on the amount of live data. This parameter should be used with caution. A value that is too small will cause the system to spend an excessive amount of time doing garbage collection.

-XX:GCTimeRatio=*nnn*
A hint to the virtual machine that it's desirable that not more than 1 / (1 + nnn) of the application execution time be spent in the collector.

For example -XX:GCTimeRatio=19 sets a goal of 5% of the total time for GC and throughput goal of 95%. That is, the application should get 19 times as much time as the collector.

By default the value is 99, meaning the application should get at least 99 times as much time as the collector. That is, the collector should run for not more than 1% of the total time. This was selected as a good choice for server applications. A value that is too high will cause the size of the heap to grow to its maximum.

Suggested strategy

Do not choose a maximum value for the heap unless you know that the heap is greater than the default maximum heap size. Choose a throughput goal that is sufficient for your application.

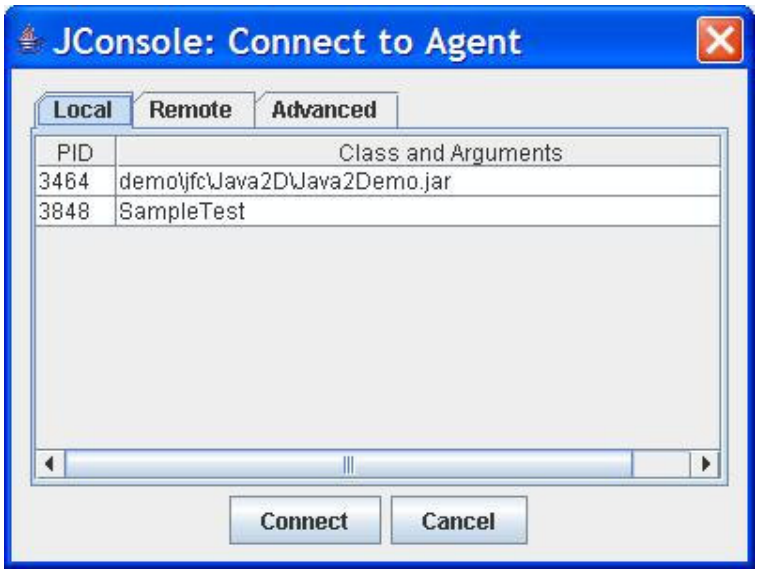
In an ideal situation the heap will grow to a value (less than the maximum) that will support the chosen throughput goal.

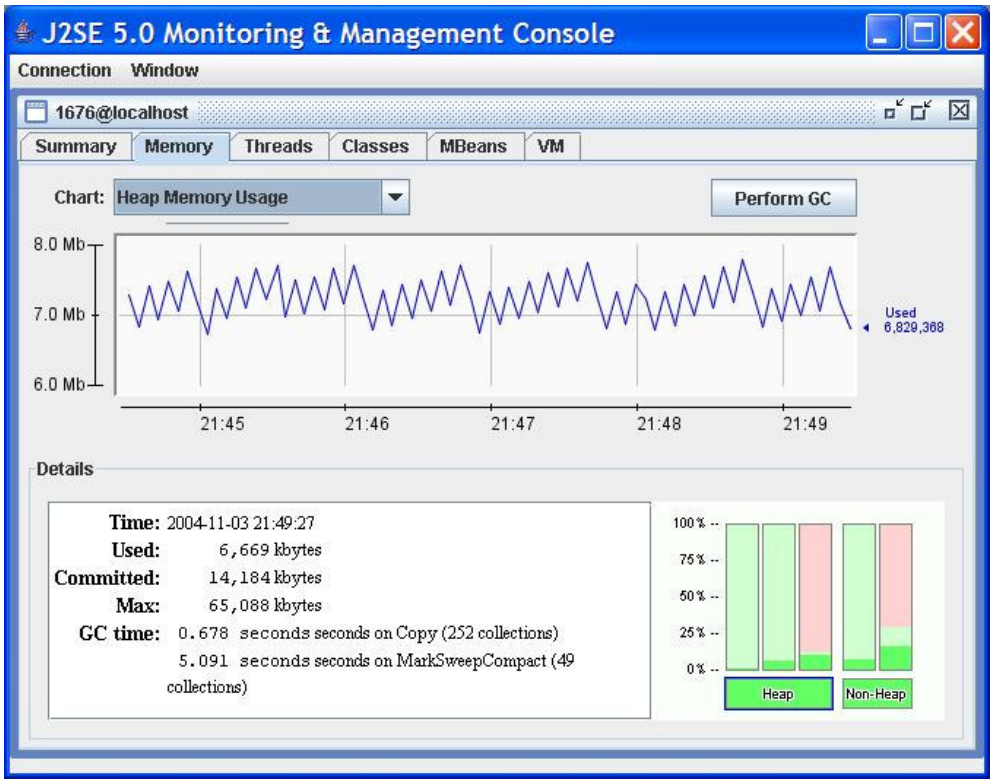
If the heap grows to its maximum, the throughput cannot be met within that maximum. Set the maximum heap as large as you can, but no larger than the size of physical memory on the platform, and execute the application again. If the throughput goal can still not be met, then it is too high for the available memory on the platform.

If the throughput goal can be met but there are pauses that are too long, select a pause time goal. This will likely mean that your throughput goal will not be met, so choose values that are an acceptable compromise for the application.

Jconsole

Provides information on performance and resource consumption of applications running on the Java platform.





Eden Space (heap) where memory is initially allocated for most objects.

Survivor Space (heap) Pool containing objects that have survived GC of Eden space.

Tenured Generation (heap) objects that have existed for some time in the survivor space.

Permanent Generation (non-heap). Holds all the reflective data of the virtual machine itself, such as class and method objects. With JVMs that use class data sharing, this generation is divided into read-only and read-write areas.

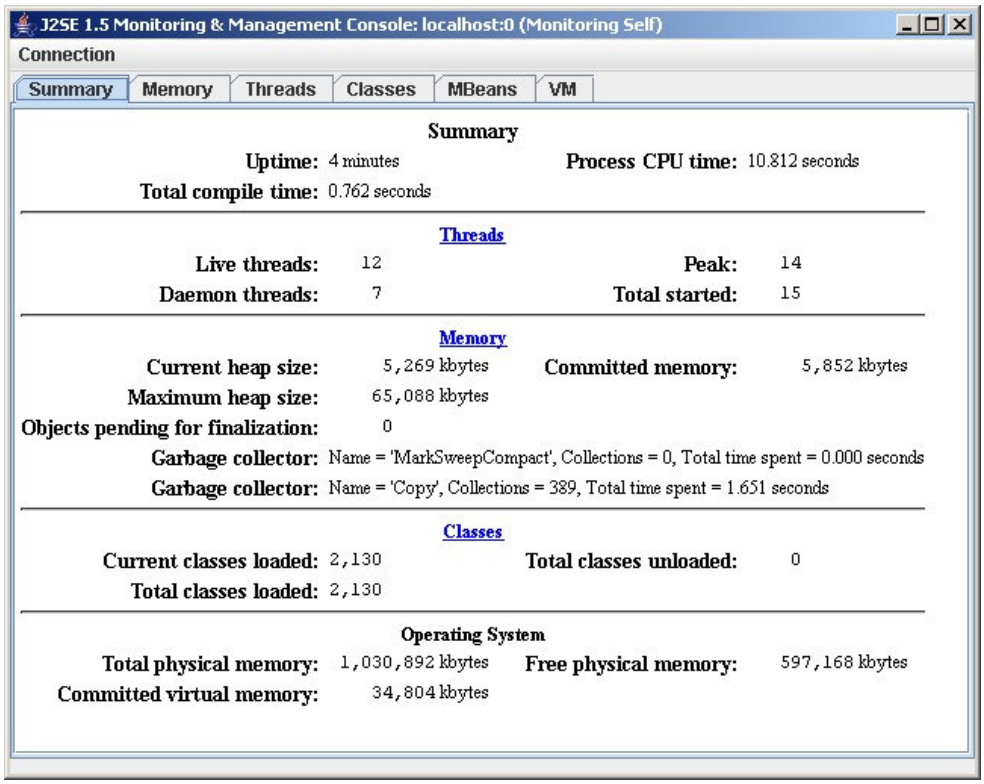
Code Cache (non-heap) memory used for compilation and storage of native code.

Used The amount of memory currently used (reachable and unreachable objects.)

Committed The amount of memory guaranteed to be available for use by the JVM. *committed* can be less than the amount of memory initially allocated at startup. *Committed* ≥ *used*.

Max The maximum amount of memory that can be used for memory management.

GC time The cumulative time .spent on garbage collection and the total number of invocations



Summary

- **Uptime**: how long the JVM has been running
- **Total compile time**: the amount of time spent in just-in-time (JIT) compilation.
- **Process CPU time**: the total amount of CPU time consumed by the JVM
- **Live threads**: Current number of live daemon threads plus non-daemon threads
- **Peak**: Highest number of live threads since JVM started.
- **Daemon threads**: Current number of live daemon threads
- **Total started**: Total number of threads started since JVM started (including daemon, non-daemon, and terminated).
- **Current heap size**: Number of Kbytes currently occupied by the heap.
- **Committed memory**: Total amount of memory allocated for use by the heap.
- **Maximum heap size**: Maximum number of Kbytes occupied by the heap.
- **Objects pending for finalization**: Number of objects pending for finalization.
- **Garbage collector information**: Information on GC, including the garbage collector names, number of collections performed, and total time spent performing GC.
- **Current classes loaded**: Number of classes currently loaded into memory.
- **Total classes loaded**: Total number of classes loaded into memory since the JVM started, included those subsequently unloaded.
- **Total classes unloaded**: Number of classes unloaded from memory since JVM started.
- **Total physical memory**: Amount of random-access memory (RAM) that the OS has.
- **Free physical memory**: Amount of free RAM the OS has.
- **Committed virtual memory**: Amount of virtual memory guaranteed to be available to the running process.

jmap

Prints shared object memory maps or heap memory details of a given process or core file or a remote debug server.

> ps -Al | grep java

0 S 3249 29778 1 99 86 10 - 166167 184466 ? 06:07:57 java

> jmap –histo:live 29778

```
num     #instances         #bytes  class name
----------------------------------------------
  1:      3568255      85638120  java.util.HashMap$Entry
  2:       845833      77840840  [Ljava.lang.Object;
  3:       498776      43458784  [Ljava.util.HashMap$Entry;
  4:        37955      37439672  [I
  5:       248637      23869152  Event
  6:       209335      23497448  [C
  7:       845420      20290080  java.util.ArrayList
  8:       498769      19950760  java.util.HashMap
  9:           14      14778720  [Ljava.io.ObjectInputStream$HandleTable$HandleList;
 10:       700668      11210688  OpenEvent
 11:       497312       7956992  vectorTimeStamp
 12:       475616       7609856  java.util.HashMap$EntrySet
 13:       265610       6374640  java.lang.String
 14:         1161       4111736  [B
 15:       248637       3978192  callerList
 16:        56347       3155432  eventInfo
 17:        56347       2253880  Trans
 18:        57095       1370280  java.util.LinkedList$Entry
 19:        57474        919584  java.lang.Integer
 20:        56347        901552  arec
 21:         8156        861640  <constMethodKlass>
 22:         8156        655232  <methodKlass>
 23:        26936        646464  Process
 28:         9567        229608  State
 29:         9567        153072  alist
 30:          783         75168  java.lang.Class
 31:         1048         66784  [S
 32:         2073         66336  SrSeqPO
 33:         1018         53408  [[I
 34:         2071         49704  VariantGroup
 35:         2071         33136  RaceTable
…
Total       9317239     401041256
```