Created by: Raúl Castillo

Contents

Created by: Raúl Castillo

Table of figures

Created by: Raúl Castillo

JVM Garbage Collector

1. Introduction

The aim of this document is to give insights about the garbage collection process on Java Virtual Machines. This process, which helps programmers to forget about allocating and freeing memory, is often misunderstood, which leads to problems such as running out of memory, spending too much time on processes such as freeing or allocating memory and generating too long pauses (stop-the-world events, which are explained below) during its execution. Following, general concepts about the automatic management of memory are explained and then the main ways to perform garbage collection in Java are explained. This document does not deal with flags needed to tune the garbage collector since these will be analysed in future publications.

2. The garbage collector

A garbage collector is a process used by languages based on automatic memory management that allows programmers to forget about complex routines to manage memory, giving them more time to focus on the goal of the software they have to write. The garbage collector also solves two problems that are common in languages with explicit memory management, which are:

- Dangling references, which appear when an object that is still referenced by other(s) is deleted, generating unpredictable behaviour.
- Leaks of memory, which happen when objects are not deleted from memory leading to the depletion of the heap (Sun Microsystems, 2006).

While many system administrators often think, the garbage collector is just responsible for deleting dead objects, according to Sun Microsystems (2006) the responsibilities of the garbage collector are:

- Allocation of memory
- Avoiding dangling references
- Deleting useless or dead objects

Importantly, Sun Microsystems (2006) remarks that even the use of garbage collectors cannot avoid memory leak problems caused by objects that are created indefinitely and that are referenced until the memory is depleted.

Sun Microsystems (2006) remarks the following desirable characteristics for a garbage collector.

- Safe and comprehensive, which means that garbage (dead objects or object without references) must not stay in memory for long time and that live objects must not be collected.
- Avoid introducing long pauses, which means that the garbage collector must avoid stopping applications during long time. There is a trade-off between frequency, size and time. For example, small heaps are collected in short time, but are executed frequently. On the other hand, collections over big heaps are executed less frequently, but each executions takes more time.

- Minimize fragmentation, after collecting dead objects free memory is divided in small chunks that affects the allocation performance, this problem is solved with a routine called compaction.

3. Some concepts regarding to garbage collection

I this section some important concepts to understand the garbage collections process are presented.

- **Heap**, is a space of memory that stores objects used by a program, in this article a program written in Java, which is represented by a directed graph where its edges are references to objects in the heap. An object can be referenced by a source node (object in the heap) or by a root (outside the heap) (Jones, R et al., 2012). The following figure shows a heap's representation and its objects.

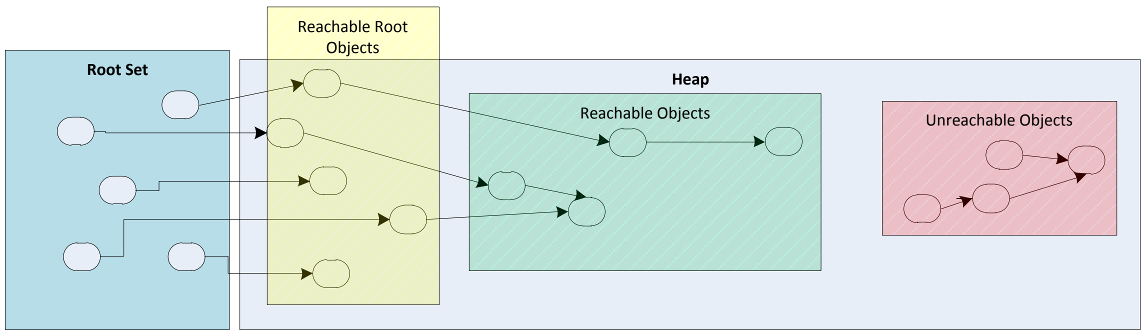


Figure 3-1 A representation of the root set and the heap

- **Garbage collector based on referencing counting**, this is a class of algorithm used to determine whether an object is alive or not. The idea is simple, each time an object is referenced by a variable a referencing counter is increased by one. On the other hand, when the relationship between referenced object and variable is broken, the counter is decreased by one. Hence, objects whose referencing counter is greater than zero are alive while objects whose counter is equal to zero are dead. The problem with this approach is that generates overhead caused by the update of referencing counters (Jones, R et al., 2012).

- **Garbage collector based on tracing algorithms**, with this approach the garbage collector looks for objects that are directly or indirectly reachable from the root set. For example, in Figure 3-1, reachable root objects are directly reachable from the root set and reachable objects are indirectly reachable from the root set. The goal of these algorithms is to discover these objects, which are alive so all the rest represent dead objects (Jones, R et al., 2012). This is the approach used by the Java Virtual Machine.

- **Mutator threads**, these are application's threads whose goal is to allocate new objects and to mutate the graph of objects based on references between them. Importantly, these references can set relationships with objects in the heap and with roots; known examples of roots are static variables, thread stacks, etc. In addition, mutator threads can generate unreferenced or unreachable objects during the program execution, which are seen by the garbage collector as garbage that can be collected (Jones, R et al., 2012). In Figure 3-1 roots (root set), root objects, reachable objects and unreachable objects are shown.

- **Mutator roots**, these represent a set of pointers that are not part of the heap and are directly reachable from mutator threads. For example, mutator roots can be reference variables in the Java stack, static reference variables or reference variables using Java Native Interface. In addition, these pointers are connected to objects in the heap, which are called root objects. Therefore, mutator threads use mutator roots that point to root objects to access the graph of objects (Jones, R et al., 2012). Figure 3-1 shows these objects. The following figure shows an example of roots within the Java stack; as can be seen the reference variable "ce", which is stored in the Java process' stack points to an object called "CExample" allocated in the Java process' heap. Hence, "ce" belongs to the root set and "CExample" is a reachable root object.

```
…
CExample ce = new CExample();
…
```

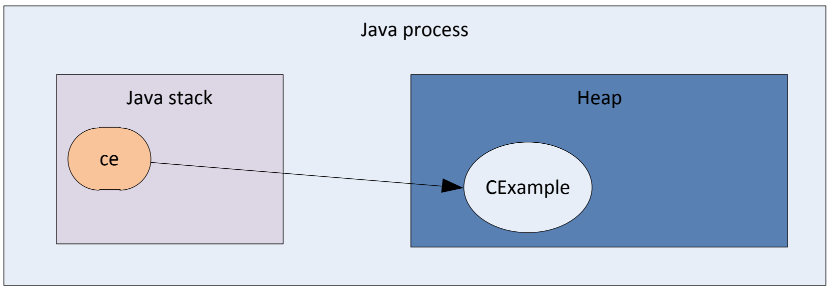


Figure 3-2 A reference variable from the stack that points a Java object in the heap

- **Collector thread**, it looks for unreachable objects to reclaim the space occupied by them; an unreachable object cannot be reached from the root set neither directly nor indirectly, it is important to remark this is a low priority thread that runs in the background (Jones, R et al., 2012). Figure 3-1 roots shows a group of unreachable objects remarked in red, these objects can be collected by the collector thread.

- **Memory allocation**, as was stated at the beginning of this article, the garbage collector process is not only about deleting dead or unreferenced objects from the heap, but is also about allocating memory. One of the challenges for the allocation process is the use of multiple threads so if the heap was seemed as an atomic block, these threads would be put in a queue to avoid conflicts and to be multithread-safe (Sun Microsystems, 2006). This situation would generate contention, which is shown in the following figure.

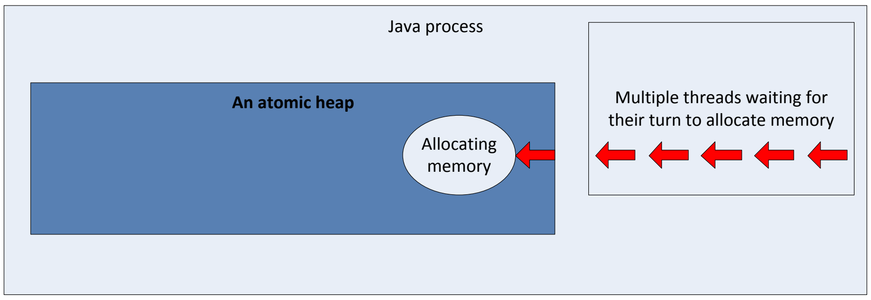


Figure 3-3 A hypothetical scenario where multiple threads are waiting to allocate memory in a safe way

The hypothetical problem shown in previous figure is solved using an approach called **Thread-Local Allocation Buffers (TLAB)**, which means that every thread has its own buffer to allocate memory inside the heap, which avoids the contention imposed by thread-safe operation on the same heap. The following figure shows several threads working in parallel because each one has its own TLAB.

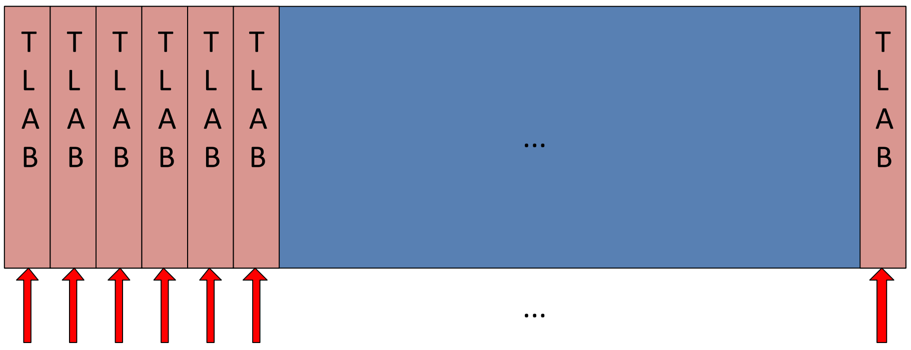


Figure 3-4 Thread-local allocation buffers used to avoid contention within the memory allocation process

In most cases, garbage collectors included a routine called compaction that organize the heaps to avoid fragmentation. Thus, there is an approach called bump-the-pointer that is used to allocate memory quickly. This method decides whether the space in the heap is suitable for the new object or not. In case the answer is yes, the pointer is updated to the next position and the object is allocated (Sun Microsystems, 2006).

The following figure shows the bump-the-pointer allocation approach on a TLAB. In this case a fragmented TLAB is compacted and the position of the pointer that controls the allocation is set. Thus, when a new object arrives, the pointer is moved based on the new object's size and the new object is allocated.

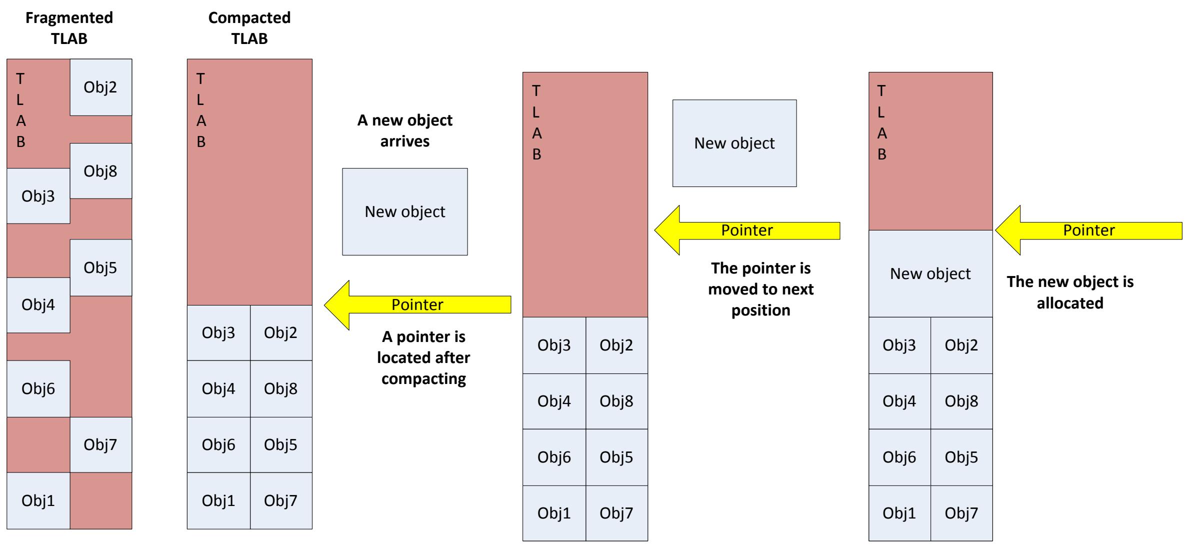


Figure 3-5 Bum-the-pointer process on a thread local allocation buffer

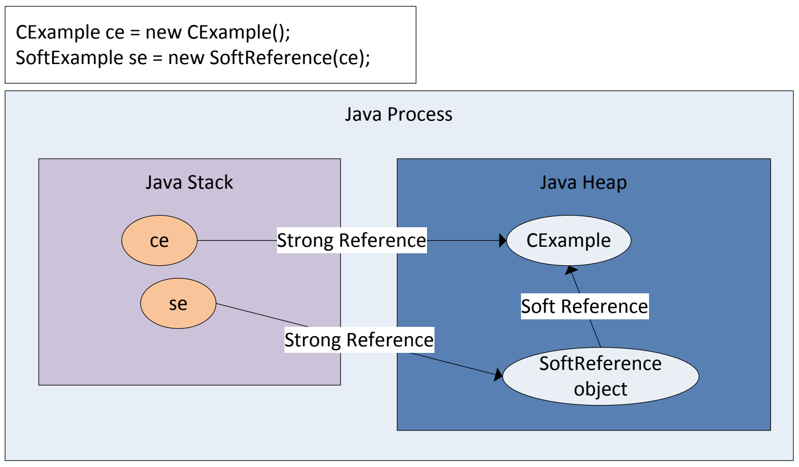
- **Liveness**, above the garbage collector based on tracing algorithms was explained and it was stated that an object is alive when is directly or indirectly reachable from the root set as can be seen on Figure 3-1. However, it is important to realize that Java has two different kinds of references; these are strong references and weak references.

4. Types of references in Java

- **Strong references**, these are references as the one shown in Figure 3-2, which define a strongly reachable object (Sharan, K, 2014).

- **Weak references**, these are not strong references that are implemented using the following classes java.lang.ref.WeakReference, java.lang.ref.SoftReference and java.lang.ref.PhantomReference. In broad sense these classes defines weakly reachable objects. However, it is possible to be more specific to say an object is weakly reachable (java.lang.ref.WeakReference), softly reachable (java.lang.ref.SoftReference) or phantom reachable (java.lang.ref.PhantomReference) (Sharan, K, 2014).

  The following figure shows the generation of a soft reference also known as a weak reference in a broad sense.



```
CExample ce = new CExample();
SoftExample se = new SoftReference(ce);
```

Figure 4-1 Generating a soft reference. Source: (Sharan, K, 2014).

The previous figure shows that creating a soft reference is composed of two steps the first step is the generation of an object in this case a CExample object and the second step is the generation of a SoftReference object that points to CExample (Sharan, K, 2014).

According to Sharan, K (2014) the difference between strong references and soft references is defined by the garbage collector, which can collect weakly referenced objects, but cannot collect objects with at least a strong reference. Thus, weak references have different uses; according to Jones, R et al., (2012) java.lang.ref.SoftReference objects are useful to create and shrink caches, java.lang.ref.PhantomReference objects can be used to control the object's finalisation order and java.lang.ref.WeakReference objects can be used to implement canonicalization tables.

Sharan, K (2014) states that an object can be reached from the root set by a chain of references composed by a mix of strong, soft, weak or phantom references. Thus, in order to get the kind of reachability that correspond to objects it is necessary to consider this.

Strong References > Soft References > Weak References > Phantom References

With this in mind, when an object is reached by a chain of different references the weakest reference within the chain sets the object's reachability. For example, in the following picture the object CExample is phantom reachable.

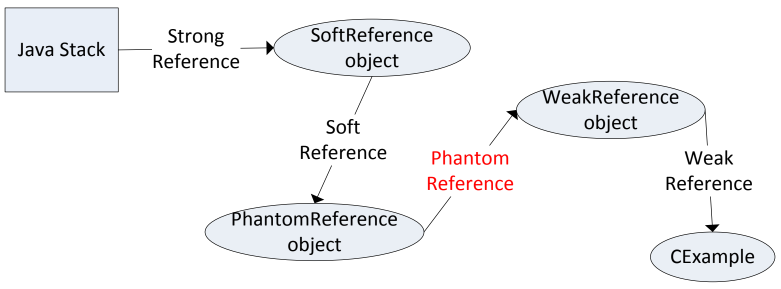


Figure 4-2 A phantom reference object

Sharan, K (2014) also states that when an object is reachable from the root set through different chains, the strongest one stablishes the object's reachability. For example, in the following figure the object CExample is strongly reachable.

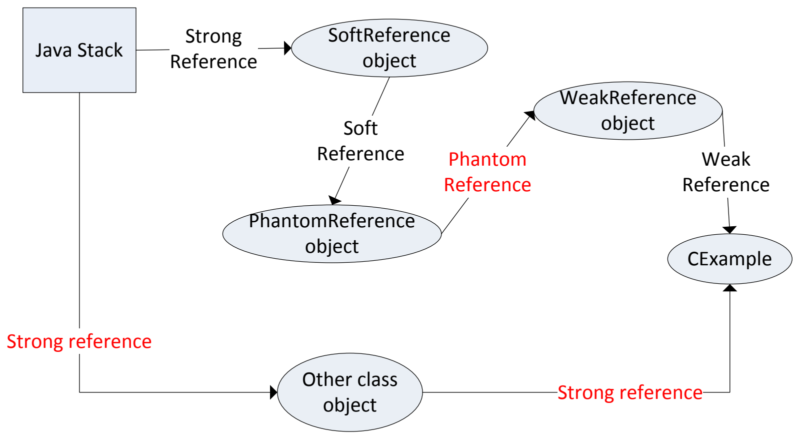


Figure 4-3 A strongly reachable object

5. Heap based on generations

Dividing the heap into generations is based on an empirical observation of the objects' distribution over the processes' lifetime, which is known as **weak generational hypothesis.** This hypothesis demonstrates most objects die when they are younger (Oracle, 2016).

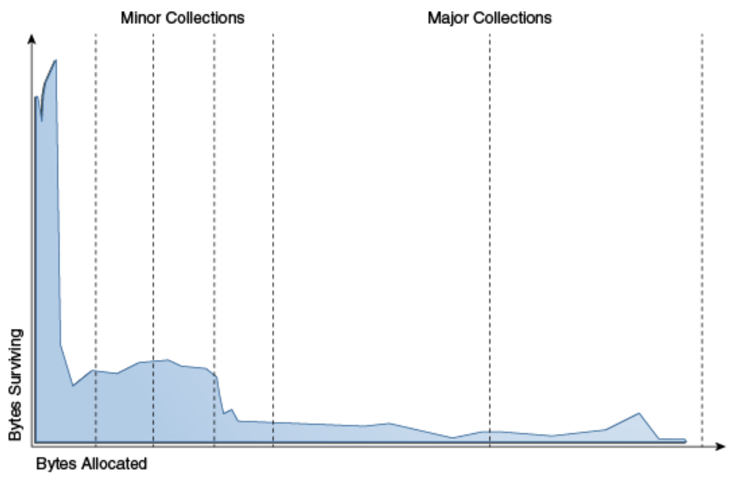


Figure 5-1"Typical distribution for Lifetimes of Objects" Source: Oracle (2016)

The previous figure shows the vast majority of objects survive just for a short time where the time is described by the X-axis based on the allocation of bytes during the process' lifetime. Following the peak of survivor objects, there is another group of objects that survive and after that is possible to see that just a little amount of objects survive for a long time.

Writing a garbage collection algorithm could be easy if the algorithm look for live objects through the whole graph, which represents the heap. However, this strategy is inefficient and then algorithms have been implemented to take advantage of the weak generational hypothesis (Oracle, 2016).

Based on the weak generational hypothesis, the heap is collected using a generational strategy that boosts the garbage collection process' performance using two algorithms.

- An algorithm whose best performance is reached when most objects are dead. Hence, this algorithm should be used on the young generation because most objects die in a short time (Oracle, 2016). According to Sun Microsystems (2006), the focus of this algorithm is the speed since it is executed frequently. This algorithm is based on the work presented by Cheney, C (1970).

- An algorithm that because of its impact on the application should be executed rarely. Hence, this algorithm should be used on the old generation because few objects live for a long time (Oracle, 2016). According to Sun Microsystems (2006), the focus of this algorithm is space efficiency because the old generation is larger than the new generation and there is not much garbage as in the new generation. The algorithm used here is based on the work of McCarthy J. (1960), who coined the term "garbage collector", and is known as mark and sweep.

The following figure shows the heap structure based on generations.

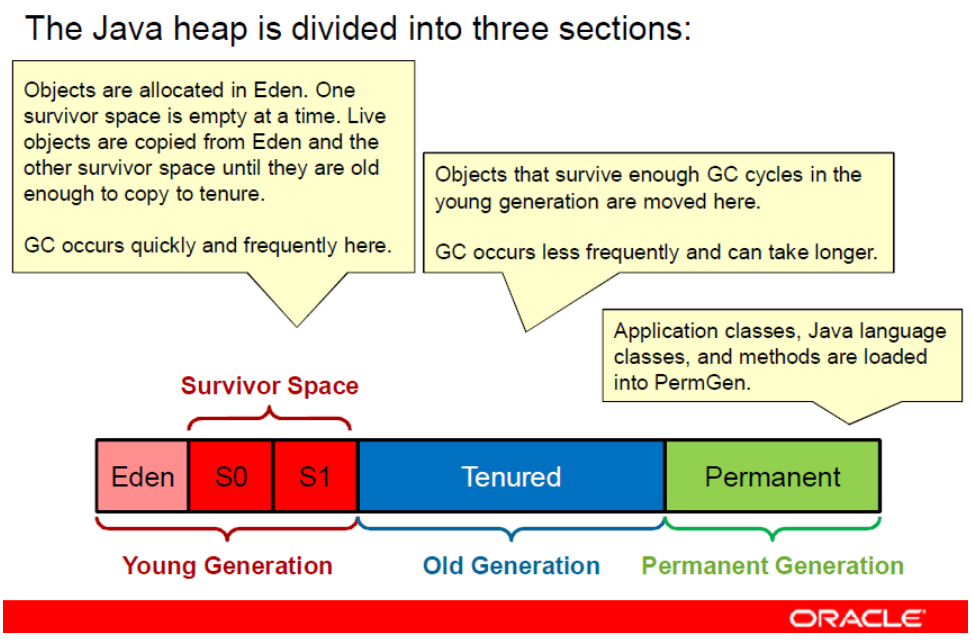

The Java heap is divided into three sections:
Objects are allocated in Eden. One survivor space is empty at a time. Live objects are copied from Eden and the other survivor space until they are old enough to copy to tenure.
GC occurs quickly and frequently here.
Objects that survive enough GC cycles in the young generation are moved here.
GC occurs less frequently and can take longer.
Application classes, Java language classes, and methods are loaded into PermGen.
Survivor Space
Eden
S0
S1
Tenured
Permanent
Young Generation
Old Generation
Permanent Generation
ORACLE

Figure 5-2 A heap divided into generation Source: Oracle (2014)

Following a description of each one of these spaces is provided.

- **Young generation**, this is the region where most new objects (i.e ExampleClass ec =new ExampleClass) are placed. However, few very large objects can be placed on the old generation directly. This generation is composed by two parts, the Eden that receives new objects and two survivor spaces that are used one by one to put objects that have survived at least a minor collection execution (Sun Microsystems, 2006).

- **Old generation**, this region storages objects that have survived minor collections and objects that are located there directly (Sun Microsystems, 2006).

- **Permanent generation**, this zone saves metadata about classes and methods as well as methods and classes (Sun Microsystems, 2006). Importantly **this region of memory is not taken from the heap**; this is another region so a Java process' memory consumption is the sum of heap plus permanent and others. Importantly**,** the permanent generation has been removed in JDK 8 to use a region of native memory called metaspace (Oracle, 2016).

6. Garbage collection types

As was stated above there are three main segments that compose the heap and non-heap area used by a Java process. Additionally there are two suitable algorithms for the young and the old generation, which are known as minor and major collections respectively (Sun Microsystems, 2006). A minor collection is triggered when the young generation is depleted, which means the Eden space is full. On the other hand, a major collection is executed when the old generation is full. A major collection is executed on the heap and permanent generation so when the old generation becomes depleted, the young generation is collected using its own algorithm and the old and permanent generations are collected using the old generation algorithm (Sun Microsystems, 2006).

However, in some cases when a major collection is triggered, there is not enough space on the old generation to afford the execution of a minor collection (which moves objects from the new generation to the old one) on the young generation so in this case the old generation algorithm is executed on the new generation too. This method is not used by the concurrent mark and sweep collector (Sun Microsystems, 2006).

It is important to remark that both minor collections and major collections are stop-the-world events (Oracle, 2012).

7. Garbage collection strategies

Beyond the kind of collections used on each generation, there are different strategies to execute them depending on the available resources and tuning goals such as throughput or response time. For example, there are strategies that improve the throughput (amount of data processed per unit of time) while others improve the response time, but affect the throughput. Thus, these are the available strategies for generational heaps.

a. Serial collector

This collector uses just a CPU (do not be confused with a CPU with several cores, it means just one hardware thread) and collects the young and old generation and a serial way (Sun Microsystems, 2006).

Execution on the young generation, the following figure describes the execution of the serial collector on the young generation. After the Eden has been filled up, a minor collection is triggered and then Eden's live objects are copied into the "To" survivor space. The "From" survivor space is also collected and live objects are copied into the "To" survivor space if they are young enough to continue living in the young generation. Older objects from the "From" survivor space are moved to the old generation. Importantly, objects from the young generation that are too large to be moved to the "To" survivor space are copied to the old generation or tenured. When the “To” survivor space becomes full during the process, remaining live objects from Eden and “From” survivor space are copied to the old generation directly (Sun Microsystems, 2006).

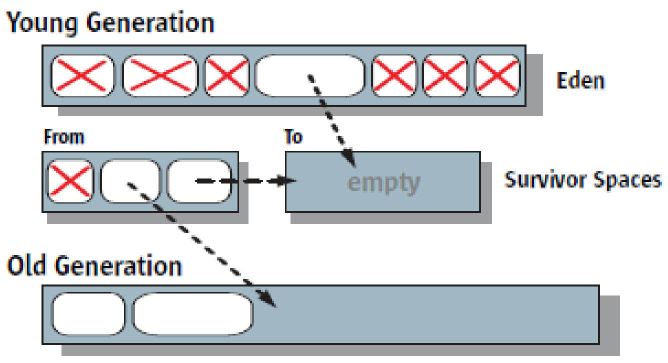


Figure a-1 Moving objects between generations Source: Sun Microsystems (2006)

After copying the live objects to the “To” survivor space and to the old generation, the remaining objects are garbage and they are not checked by the collector and then the Eden is empty. The “To” survivor space changes roles with “From” so at the end an empty “To” survivor space is generated again and memory looks like the following figure (Sun Microsystems, 2006).

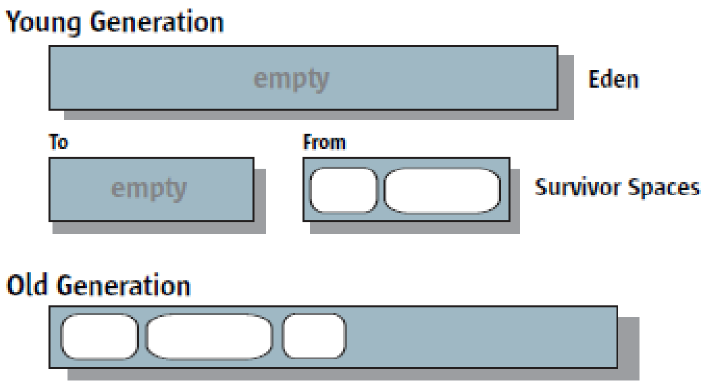


Figure a-2 Starting the process again Source: Sun Microsystems (2006)

Execution on the old generation, in this case the algorithm used is called mark-sweep-compact. The mark phase identifies live objects while the sweep phase explores the heap to identify garbage and then a compaction process is executed on the live objects. This process is described on the following figure (Sun Microsystems, 2006).

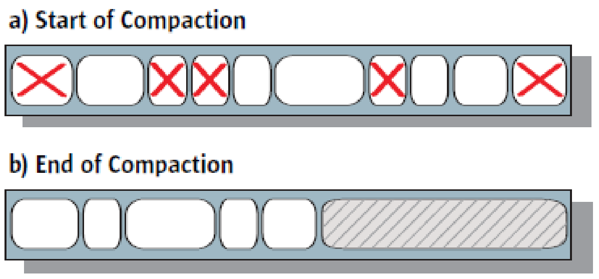


Figure a-3 Garbage collection on the old generation Source: Sun Microsystems (2006)

This collector is used on machines with a single CPU o hardware thread. However, understanding it is important because other collectors such as the parallel collector use the same algorithms, but in other way. This collector is enabled with this command line option -XX:+UseSerialGC (Sun Microsystems, 2006).

b. Parallel collector

According to Oracle (2016) this collector is also known as the throughput collector because it uses several hardware threads in order to minimize the time used to execute garbage collection on the young generation increasing the amount of data processed by unit of time (throughput).

Execution on the young generation, this collector implements the same algorithm used by the serial collector stop-the-world and copying, but using several threads instead of one that optimize the time needed to execute this task (Sun Microsystems, 2006).

Execution on the old generation, this collector uses the same strategy as in the case of the serial collector, which means a mark-sweep-compact algorithm using just one thread. This is also executed on the permanent generation. This collector is used through the following command line -XX:+UseParallelGC (Sun Microsystems, 2006).

c. Parallel compacting collector

This collector uses the same strategy used by the parallel collector on the young generation. However, the old generation is also collected in a parallel way.

Execution on the old generation, the process on the old generation is composed of three phases.

- **Marking**, after dividing the generation on fixed regions the first set of live objects (directly reachable from the source code) are **marked using several threads** (GC threads in a parallel way) in order to discover all the live objects that are reachable since this first set. For each live object marked, the region where it lives is updated with the object's location and the object's size (Sun Microsystems, 2006). The following figure describes the result of this process where an old generation is divided into six regions.

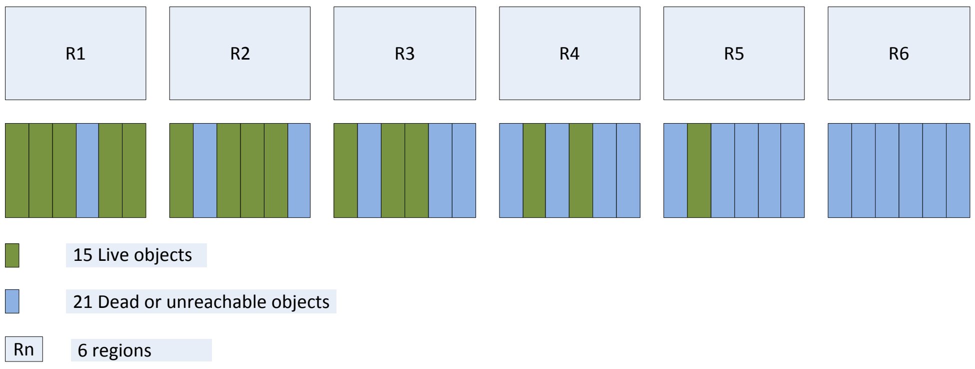


Figure c-1 Old generation divided into six regions

- **Summary**, this phase identifies dense regions and gets information needed to compact them. This phase is executed in a **serial way** and works in this way; as previous garbage collector executions have compacted the old generation, it is likely that each region has dense areas with live objects in the left side. Thus, running compactions on these regions is not necessary so the heap is marked with a prefix to identify from which region compaction is needed. It means that any object will not be moved to the left side of the prefix so all the movements will be done on the right side of that prefix. All the information about these movements is generated in this phase (Sun Microsystems, 2006). The following figure shows the possible movements calculated during this phase.

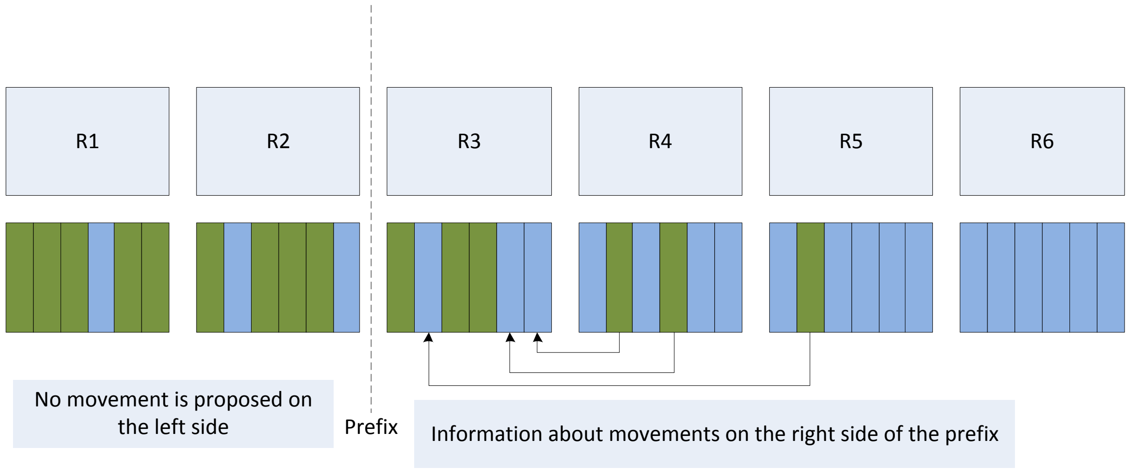

R1
R2
R3
R4
R5
R6
No movement is proposed on the left side
Prefix
Information about movements on the right side of the prefix

Figure c-2 The prefix and possible movements on the right side of the prefix

- **Compaction**, this phase uses several threads in a parallel way to compact all the regions identified during the summary phase. In this way, the heap has a high density in one extreme and is empty on the other (Sun Microsystems, 2006). The following figure shows the result of this process.

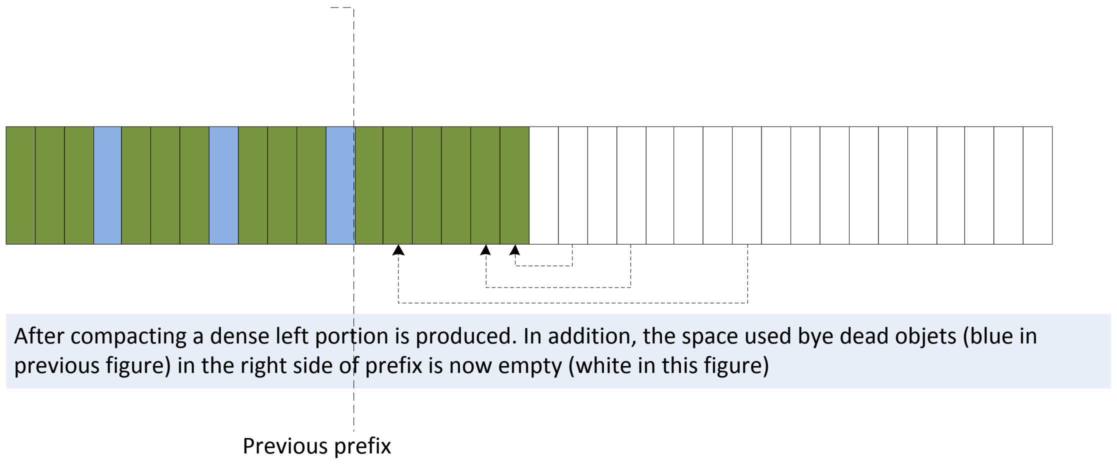

After compacting a dense left portion is produced. In addition, the space used bye dead objets (blue in previous figure) in the right side of prefix is now empty (white in this figure)
Previous prefix

Figure c-3 Compaction process' result

There are two command line options for this collector -XX:+UseParallelOldGC to activate this collector and –XX:ParallelGCThreads=n to set the number of hardware threads used by the collector (Sun Microsystems, 2006). According to Oracle (2016) if the number of hardware threads ("N") is greater than 8, the JVM will use (5/8)*N threads for the garbage collector by default; otherwise (i.e. 4 hardware threads available) it will use the "N" available threads.

d. Concurrent mark and sweep (CMS) collector

The goal of this collector is to improve the responsiveness especially for collections that happen on the old generation, which can last even seconds when large heaps are used. The main idea is to execute the garbage collector process while the application is executed in order to avoid pauses that affect the responsiveness of applications.

Execution on the young generation, the young generation is collected using a parallel collector.

Execution on the old generation, the CMS collector on the old generation is divided in four processes. The following figure shows the processes of this collector.

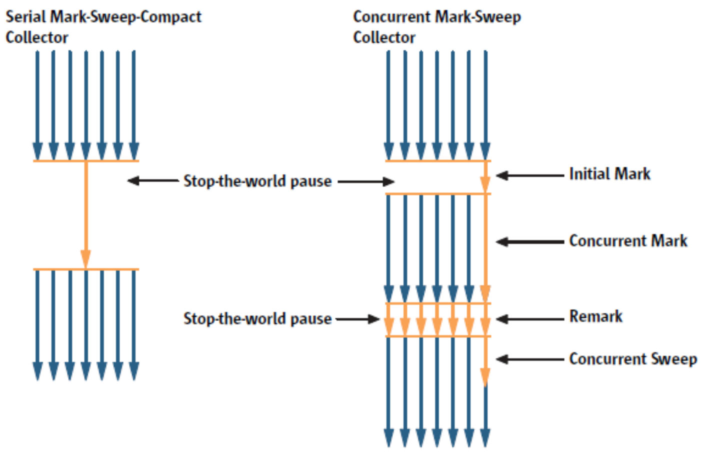


Figure d-1 CMS collector on the Old Generation Source: Sun Microsystems (2006)

- **Initial mark**, this process lasts for a short time and is a **stop the world** event executed in a serial way, which means only a hardware thread is used and the application is stopped. The goal of the initial mark is recognizing live objects that have a direct relationship with the source code (Sun Microsystems, 2006).
- **Concurrent mark**, this process analyses the set of objects produced by the previous phase to reach all the objects that have relationships (transitively reachable) with this set. This is a non-stop the world event and is concurrent, which means the application runs while the garbage collector is executed. Therefore, since the hardware threads are shared between the application and the garbage collector, the throughput is affected. Additionally, the concurrence between the garbage collector and the application produces floating garbage because the application modifies references between objects while the garbage collector is marking live objects (Oracle, 2016).
- **Remark**, this process faces the problem of the floating garbage that is generated by the concurrent mark. This is a **stop the world** event since the application is stopped in order to mark live objects. Thus, in order to decrease the pause caused by this stop the world event, this process is executed

using several hardware threads or in a parallel way. This process ensures that all live objects are marked (Sun Microsystems, 2006).

- **Concurrent sweep,** after live objects are identified by previous processes, this process recovers the space occupied by garbage (non-referenced) objects. This process does not execute compaction on the heap so the CMS garbage collector will impact the allocation on the old generation because it is impossible to use a bump-the-pointer technique to allocate objects. Then, the CMS garbage collector implements a linked list to allocate objects on the old generation. In addition, the lack of compaction can generate fragmentation on the heap (Sun Microsystems, 2006). The following figure shows the lack of compaction.

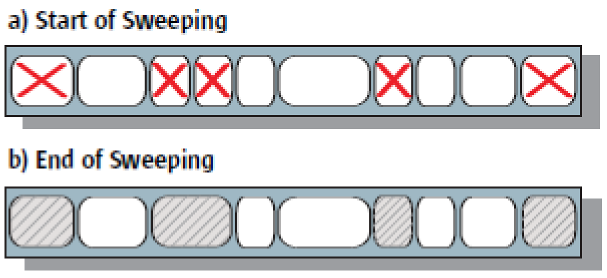


Figure d-2 Concurrent sweeping Source: Sun Microsystems (2006)

Since the CMS collector allows the execution of the application while the garbage collector is executed, it is necessary to use **larger heaps** because the application allocates memory during the execution of the marking process. In addition, because of this behaviour, the CMS collector on the old generation is **executed before the old generation is depleted**. Hence, the moment to execute the CMS collector is calculated using statistics about previous garbage collections executions and how fast the old generation is filled (Sun Microsystems, 2006). However, the CMS garbage collector is also executed when the percentage of live objects in the old generation is greater than the value of **–XX:CMSInitiatingOccupancyFraction=n** whose default value is 68% (Sun Microsystems, 2006).

It is important to remark that because of the concurrence more hardware threads are needed to execute the application and the garbage collector at the same time.

Nevertheless, if the old generation becomes depleted, the CMS collector cannot be executed so the mark and sweep algorithm is executed as a stop the world event. In order to use the CMS collector this parameter must be set -XX:+UseConcMarkSweepGC (Sun Microsystems, 2006).

e. Garbage first garbage collector

According to Detlefs, D et al. (2004) this collector is suitable for machines that can afford large heap sizes and multiple CPUs. Within the paper written in 2004, there are some concepts such as regions, remembered sets, collection sets, etc. Thus, it is necessary to understand these structures and techniques first.

- **Regions and allocation,** in garbage first the heap is divided in contiguous regions of virtual memory where each region has the same size. The goal of this regionalization is to avoid tracing the whole graph of references that was depicted in Figure 3-1. Thus, this is a space-incremental collector that operates incrementally considering small portions of the heap (Detlefs, D et al, 2004). The following figure shows a heap divided into regions.

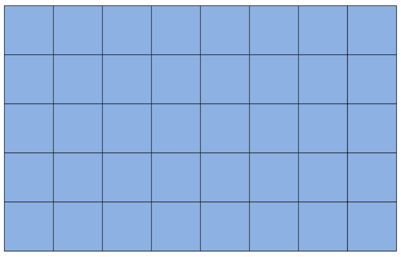


Figure e-1 A heap divided into regions of the same size represented by squares

Using regions and internal structures called remembered sets the collector knows whether objects that live in a given region are referred from other regions without tracing the whole heap (Detlefs, D et al, 2008).

To avoid contention during the allocation process each region is divided into thread-local allocation buffers. Thus, mutator threads allocate objects on these buffers using a compare-and-swap operation and after filling a region, a new one is chosen (Detlefs, D et al, 2004). Therefore, if we take a region (one square) from Figure e-1, we can see the following internal structure.

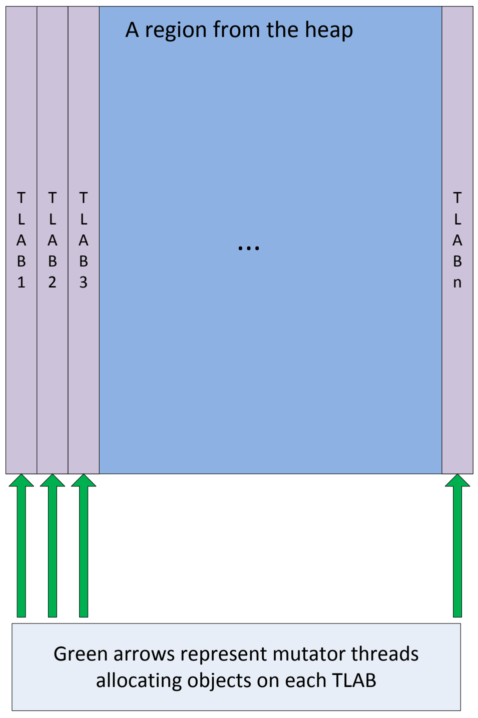

A region from the heap
TLAB1
TLAB2
TLAB3
...
TLABn
Green arrows represent mutator threads allocating objects on each TLAB

Figure e-2 The set of TLABs inside a given region

When a region is filled, a new region is selected to allocate objects, but in order to optimize this operation, empty regions are organized into a linked list (Detlefs, D et al, 2004) as is shown in the following figure.

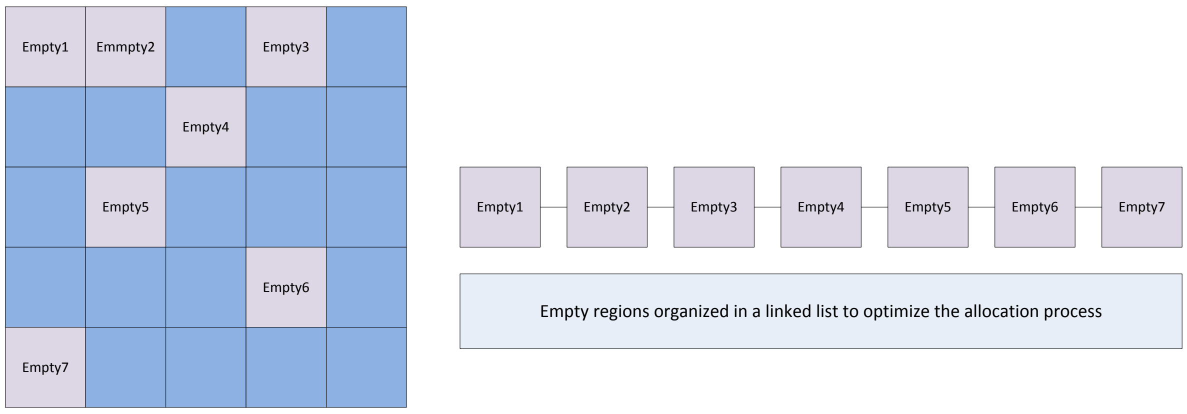


Figure e-3 The organization of empty regions that belong to the heap

- **Humongous objects**, one of the challenges faced by this collector is the allocation of large objects, these are called humongous objects and there are two types of allocations. First, when an object is greater than ¾ of a region's size and smaller than a region, the object is allocated into its own region, outside TLABs and no other object can be allocated on this region. Second, when an object is greater than a region's size, in this case the object is allocated using an integer number of regions, which work together as a region for that object. These objects are allocated outside TLABs and no other object can be allocated on these regions (Detlefs, D et al, 2008).

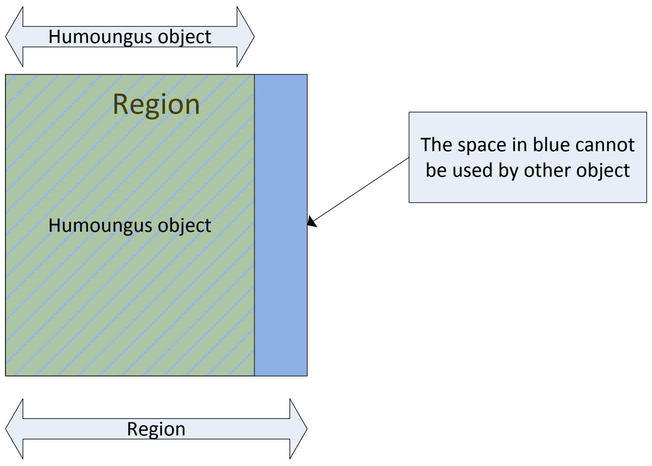


Figure e-4 A humongous object greater than ¾ of the region and smaller than the region

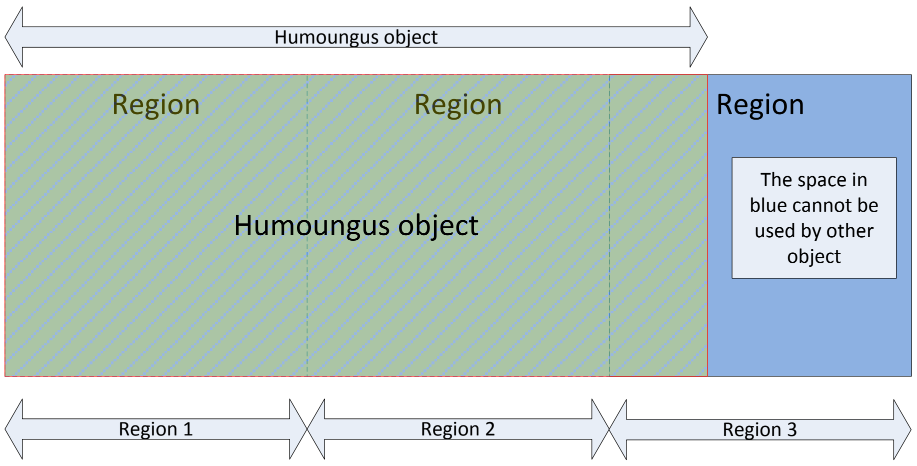


Figure e-5 A humongous object greater than a region allocated on three contiguous regions

- **Remembered sets,** each region has a data structure that saves information about the objects from other regions that point objects inside this region. Thus, each time an interregional relation is created, mutator threads should inform collector threads about this to update the remembered sets data. The goal of using remembered sets is to avoid looking for information about references within the whole heap. Thus, collector threads just need to query these remembered sets to get this information (Detlefs, D et al, 2004). The following figure shows a remembered set.

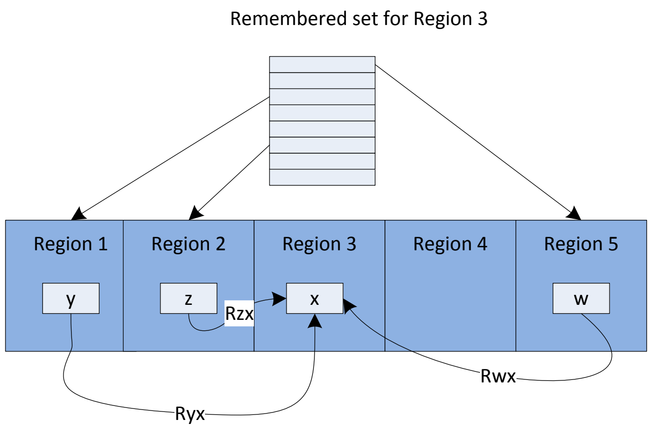
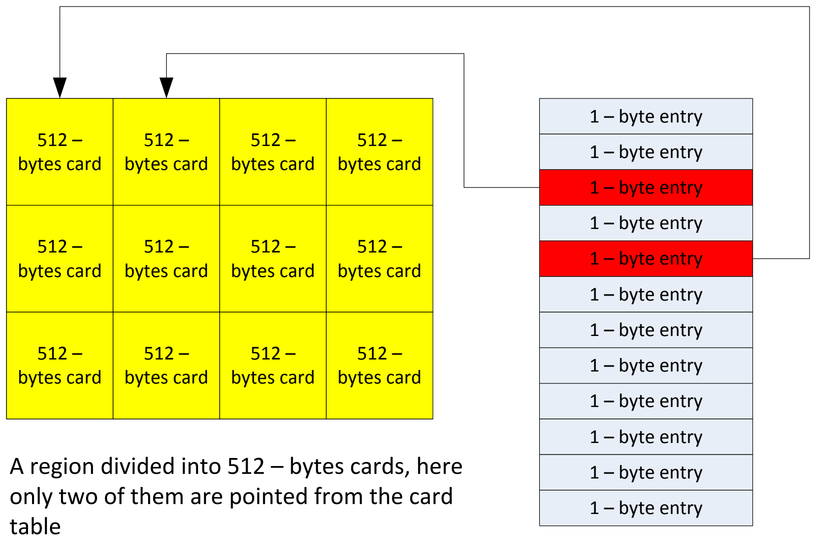


Figure e-6 A remembered set, source: Detlefs, D et al (2004)

With the remembered set shown in the previous figure, the collector avoids looking for references within the graph composed by these edges: Ryx, Rzx and Rwx. Thus, the collector should look for these relationships using a remembered set data structure, which helps to optimize the process (Detlefs, D et al, 2004).

- **Cards and tables of cards**, remembered sets are hash tables of cards; each card is a 512 bytes portion of the heap, which is mapped using a one-byte entry from the card table (Detlefs, D et al, 2004). The following figure shows this.



A region divided into 512 – bytes cards, here only two of them are pointed from the card table

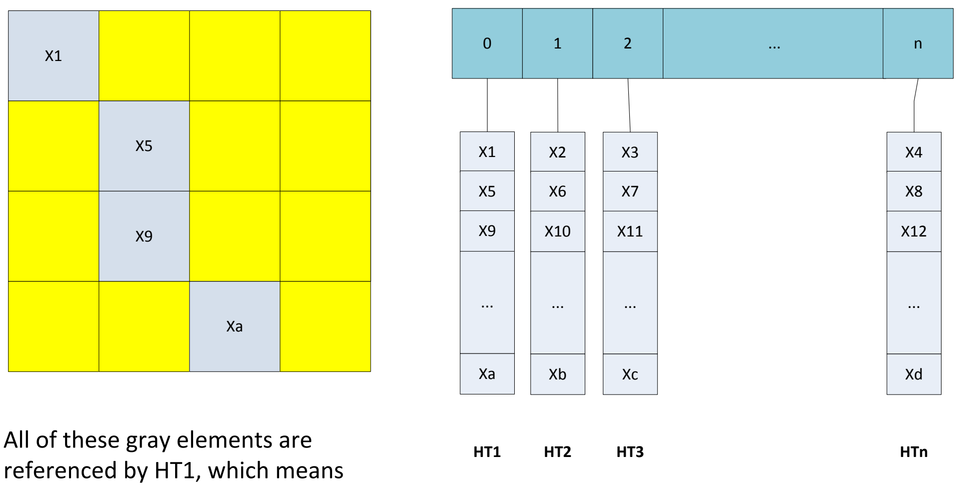
A card table (remembered set) that uses 8 bits to map a card in the region

Figure e-7 A region divided into cards and the card table or remembered set used to map them, source: Detlefs, D et al (2004)

Importantly, each region has an **array of several card tables** to avoid conflicts between the threads used by the garbage collector on the same region; which means a card table per GC thread. Therefore, a remembered set is an array of hash tables of cards (Detlefs, D et al, 2004).

A region divided into cards

This array of hash tables of cards is the remembered set



All of these gray elements are referenced by HT1, which means they are assigned to one garbage collector thread

HT: Hash table
There are “n” hash tables of cards where “n” is the number of parallel garbage collector threads

Figure e-8. A remembered set as an array of hash tables of cards, source: Detlefs, D et al (2004)

- **Generations**, garbage first is a generational garbage collector where each region can be part of the new generation (composed by the Eden and the Survivor space) or the old generation. However, generations are not a contiguous block as in previous collectors; the following figure shows this situation.

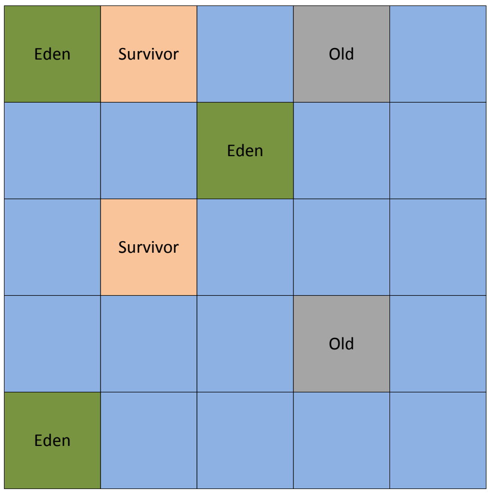
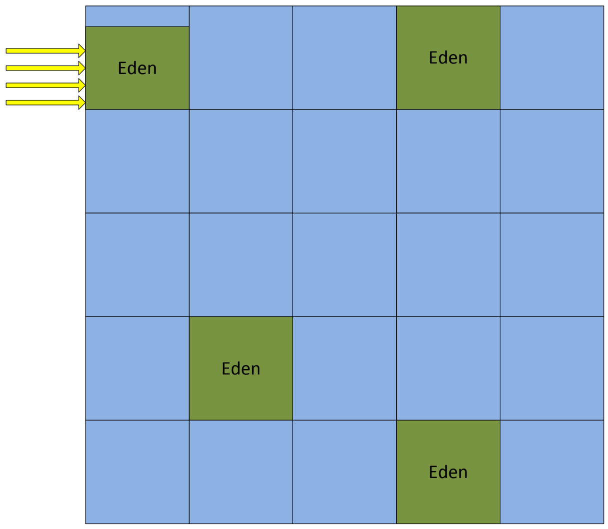


Figure e-9 Young generation and old generation distributed between different regions

In addition, young generation is designated dynamically, which means when mutator threads need another region to allocate objects, this region is automatically classified as Eden space and as a result they are considered as part of the next collection set (Detlefs, D et al, 2004). What is more Detlefs, D et al (2004) state *"...we gain an important benefit: remembered set processing is not required to consider modifications in young regions. Reachable young objects will be scanned after they are evacuated as a normal part of the next evacuation pause."*

1. Mutator threads, represented by arrows, are close to fill a region. The green color represents the space filled with objects, as can be seen some regions haven been filled before



2. After filling the region shown in the previous picture, mutator threads, now are filling other region that now is part of the young generation and part of the following collection set

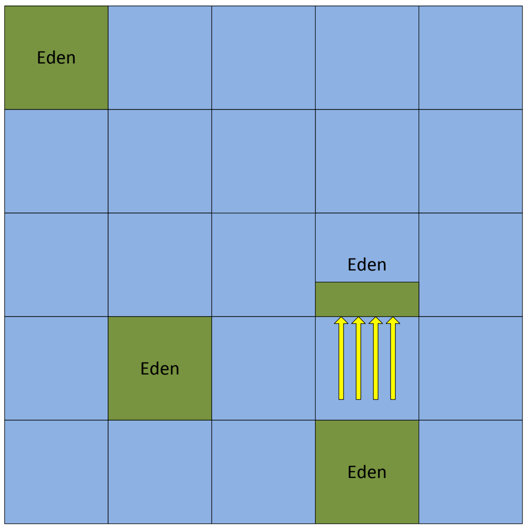


Figure e-10 The allocation of objects in the young generation, which is generated dynamically

- **Concurrent marking,** according to Detlefs, D et al (2004) concurrent marking is based on the following phases and data structures.

  - **Previous and next bitmap,** there are two bitmaps called previous and next, the first one is used to save information about the liveness as a result of the execution of the marking process. The second one is used to fill the information about liveness generated during the future execution of a new marking process; these bitmaps change roles at the end of the marking process. Therefore, the next bitmap that has been used during the process becomes the previous bitmap and the former previous bitmaps becomes available to be used as a next bitmap (Detlefs, D et al, 2004).

  - **Initial marking,** during this step the next bitmap is cleaned, mutator threads are stopped (stop-the-world pause) and all the objects that are directly reachable from the roots are marked (Detlefs, D et al, 2004).

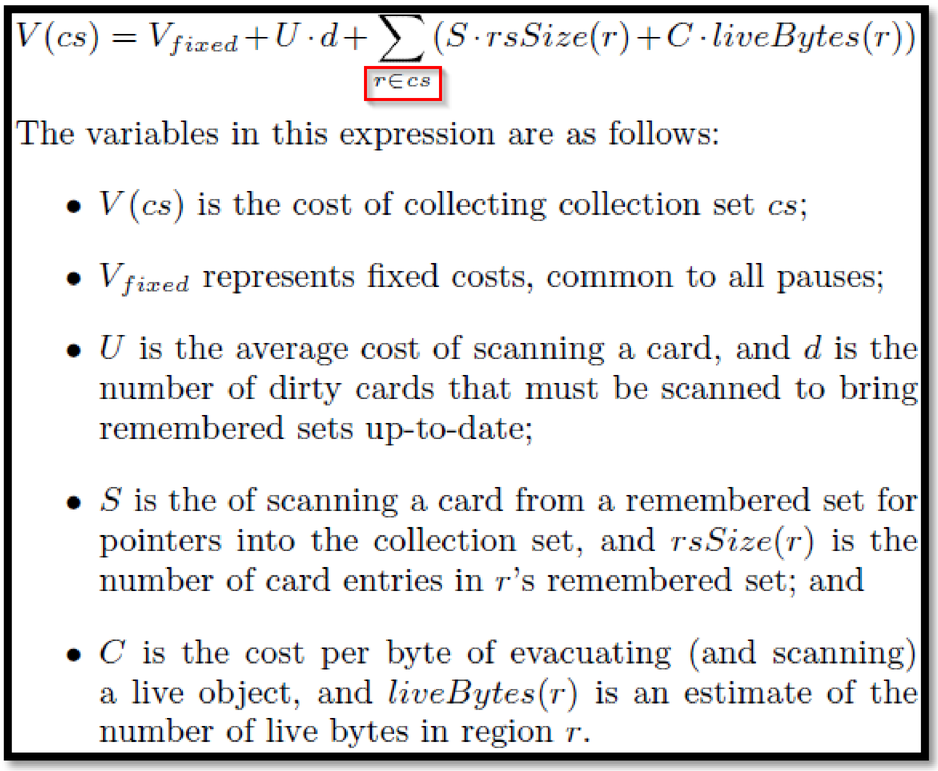
  - **Concurrent marking,** after finishing the previous step, mutator threads are enabled again and the concurrent making process mark alive objects taking as starting point the set of objects marked during the initial marking. As this process is executed while mutator threads are running, the graph can be modified while the collector is tracing it. Thus, a marking write barrier is used to save the pointer modifications that happen on the graph of references. These modifications are saved on a queue. The marking process is interrupted according to a regular interval to process the information saved on the queue (Detlefs, D et al, 2004).

  - **Final marking pause**, this a stop-the-world pause used to ensure that all the updates on the graph, made by mutator threads and recorded on the queues on the previous step, are processed. This process runs in parallel, which means using several threads (Detlefs, D et al, 2004).

  - **Counting alive data and cleaning,** during this step all the alive data marked is counted using the marking bitmap. However, as evacuation can happen during the marking process, it is necessary an additional stop-the-world cleaning process to finish the counting process properly. During this stop-the-world cleaning step, next and previous bitmaps change roles and data on the previous bitmap is used to calculate liveness. With this in mind, this process organize regions according to GC efficiency, which is metric calculated dividing the estimate of garbage within a region by the cost of collecting it. Importantly, this cost is calculated based on the formula, which will be presented in following sections. Finally, regions without any alive data are reclaimed immediately (Detlefs, D et al, 2004).

- **Soft real-time goal,** there are two types of collections within first garbage collector. One collection that is executed on a collection set composed by Eden and Survivor regions (young generation) and other collection, which is executed on a collection set composed by Eden, Survivor and Old regions. This garbage collector receives two user inputs, one input to set the desired time for stop-the-world events that run within collection sets and other inputs that sets the maximum amount of memory used to allocate objects (Detlefs, D et al, 2004).

- **How the soft real-time constraint is satisfied,** as was described above there are collections on the young generation and mixed collections, which collect garbage from the young and the old generation at the same time. In the former case, a forecast based on the average time used by previous collections is used to predict the duration of new collections. On the other hand, mixed executions use a model to forecast the duration of collections based on the following formula (Detlefs, D et al, 2004).



$$V(cs) = V_{fixed} + U \cdot d + \sum_{\boxed{r \in cs}} (S \cdot rsSize(r) + C \cdot liveBytes(r))$$

The variables in this expression are as follows:

- $V(cs)$ is the cost of collecting collection set cs;
- V_{fixed} represents fixed costs, common to all pauses;
- U is the average cost of scanning a card, and d is the number of dirty cards that must be scanned to bring remembered sets up-to-date;
- S is the of scanning a card from a remembered set for pointers into the collection set, and $rsSize(r)$ is the number of card entries in r's remembered set; and
- C is the cost per byte of evacuating (and scanning) a live object, and $liveBytes(r)$ is an estimate of the number of live bytes in region r.

Figure e-11 Formula used to forecast the duration of mix garbage collections. Source: Detlefs, D et al (2004)

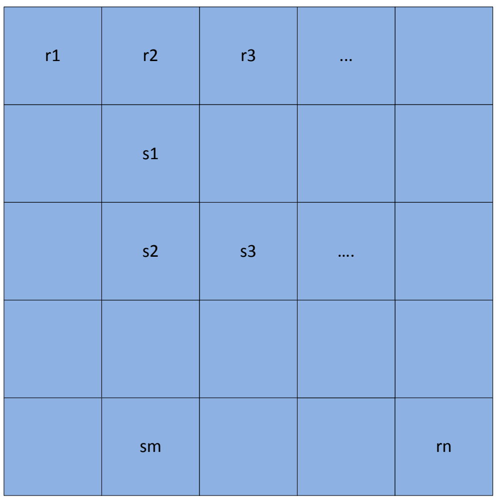
The above formula is used to calculate the pause time caused by the execution of the garbage first garbage collection on a set of regions that belong to a collection set (r Є cs). With this in mind, the algorithm can adapt the behaviour of the collector to adjust these parameters reaching the soft real-time constraint, given by the user, with high probability.

- **How the collection set is chosen,** as has been stated above, young regions are selected dynamically during the allocation process and they are part of the collection set by default so the question is how the regions from the old generation are select to be part of the collection set. At the end of the marking phase, all the regions are sorted according to their collection efficiency. However, this information cannot be used as is because remembered sets could have been updated since the finalization of the marking process. Thus, efficiency is recalculated and regions are sorted according to this new value. Regions from this new sorted set are picked to be part of the collection set taken into account two restrictions, the pause time must not be exceeded and the surviving data must not exceed the available space (Detlefs, D et al, 2004).

- **Triggering an evacuation pause,** according to Detlefs, D et al. (2004) evacuations are designed to move live objects from the collection set regions to release memory and to allow compaction. There are three possible ways to trigger an evacuation.

    - When a threshold called hard limit "H" is reached, this is triggered to ensure there is enough survivor space to execute evacuations. This hard limit "H" is based on the hard-margin "h", which represents the survivor space as a portion of the heap with size "M". The hard limit "H" is defined by H=(1-h)/M. When the paper was written its authors stated this value was a constant but would be dynamically adjusted in the future.

A heap divided in regions



R = {r1, r2, r3...rn} / set of regions that compose the heap

S = {s1, s2, s2...sm} / set of survivor regions

$S \subset R$ S is a sub set of R

$h = \sum_{i=1}^{i=m} Size(si)$ h: hard-marging equal to the total survivor space

$M = \sum_{i=1}^{i=n} Size(ri)$ M: heap size

$H = \frac{1-h}{M}$ H: hard limit, threshold used to trigger an evacuation pause

Figure e-12 The main concepts related to this threshold are summarized here

    - During fully young garbage collections, a number of young regions to be evacuated are calculated dynamically in a way that the evacuation on these regions generates a pause whose duration meets the soft real time constraint given by the user (Detlefs, D et al, 2004).

    - During mixed garbage collections, a combination of young regions and old regions to be evacuated are chosen. The frequency of collections is tuned to accomplish the soft real time constraint given by the user. As the garbage collector uses the maximum frequency within the limits of the constraint, the number of young regions within the collection set is minimized and the number of old regions is maximized (Detlefs, D et al, 2004).

According to Detlefs, D et al (2004) G1 starts running fully-young garbage collections and it turns into a partially-young mode mode after the execution of the concurrent marking phase. These, partially-young collections are monitored to measure its efficiency and when it declines, the fully-young mode is executed again. In addition, G1 also takes into account the heap occupancy. For example, let us say the heap is almost full then G1 will continue executing partially-young collections beyond the declination of its efficiency to reduce the heap occupancy.

- **Triggering a concurrent marking,** as in the previous case where we had a hard-margin “h” and a hard-limit “H”, in this case there is a soft margin “u” and a soft limit defined by H – uM. This soft limit controls the level of occupancy in the heap so when this threshold is reached before an evacuation, the marking process is triggered according to the soft real-time constraint (Detlefs, D et al, 2004).

8. Conclusions and final thoughts

This essay has been done to put in just one document some of the references that I have read in order to deal with performance issues during my experience as a Java application server’s administrator. The main conclusion, of this document and my experience is that you should avoid using the configuration by default and you should select the most suitable collector according to the goals of your application and the hardware constraints. For example, even though, the new garbage first collector sounds like a good approach to reduce the time spent on garbage collections, you should keep in mind this collector needs big amounts of memory, more than 6 GB, and several hardware threads to execute its concurrent part. What is more, you should evaluate you application because it can be the case that your application is a batch process, which can afford a penalty on responsiveness without affecting the general throughput. However, this is not the case of a final user application where responsiveness is a relevant factor to be taken into account.

9. Future work

This essay has described a bunch of collectors available for Java applications. However, it has not described how to tune these collectors so in the short term, I am going to publish some documents about tuning the garbage collector with focus on the parallel collector and garbage first garbage collector.

10. References list

Cheney, C 1970, 'A Nonrecursive List Compacting Algorithm', Communications Of The ACM, 13, 11, pp. 677-678, Computers & Applied Sciences Complete, EBSCOhost, viewed 21 February 2017.

Jones, R., Hosking, A., Moss E. (2012) *The Garbage Collection Handbook*, Boca Raton, FL: CRC Press.

McCarthy J. (1960) Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I [Online document] Available from: http://www-formal.stanford.edu/jmc/recursive.pdf

Oracle (2012) Oracle AIA 11g Performance Tuning on Oracle’s SPARC T4 Servers A Guide for Tuning Pre-Built Integrations on Oracle AIA 11g [Online document] Available from: http://www.oracle.com/us/products/applications/aia-11g-performance-tuning-1915233.pdf

Oracle (2014) Oracle Performance Tuning Workshop

Created by: Raúl Castillo

Oracle (2016) Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide [Online document] Available from: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/generations.html#sthref16 (Accessed: 16/02/2017)

Sharan, K (2014), Beginning Java 8 Language Features. [Electronic Book] : Lambda Expressions, Inner Classes, Threads, I/O, Collections, And Streams, n.p.: Berkeley, CA : Apress, 2014., University of Liverpool Catalogue, EBSCOhost, viewed 5 June 2017.

Sun Microsystems (2006) Memory Management in the Java HotSpot™ Virtual Machine [Online document] Available from: http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf (Accessed: 13/02/2017)

Detlefs, D.,Flood, C., Heller, S., Printezis, T. (2004) Garbage-First Garbage Collection [Online document] Available from: https://pdfs.semanticscholar.org/cdae/3b09950a0df9fd893928782351d091070fe2.pdf (Accessed: 03/07/2017)

Detlefs, D, Heller, S, & Garthwaite, A 2008, 'Garbage-first garbage collection', USPTO Patent Grants, EBSCOhost, viewed 4 July 2017.