

Java Performance

Java Virtual Machine

- **The Java Virtual Machine (JVM) is an abstract computing machine.**
 - The JVM is a program that looks like a machine to the programs written to execute in it.
 - This way, Java programs are written to the same set of interfaces and libraries.
 - Each JVM implementation for a specific operating system, translates the Java programming instructions into instructions and commands that run on the local operating system.
 - This way, Java programs achieve platform independence.

Java Virtual Machine

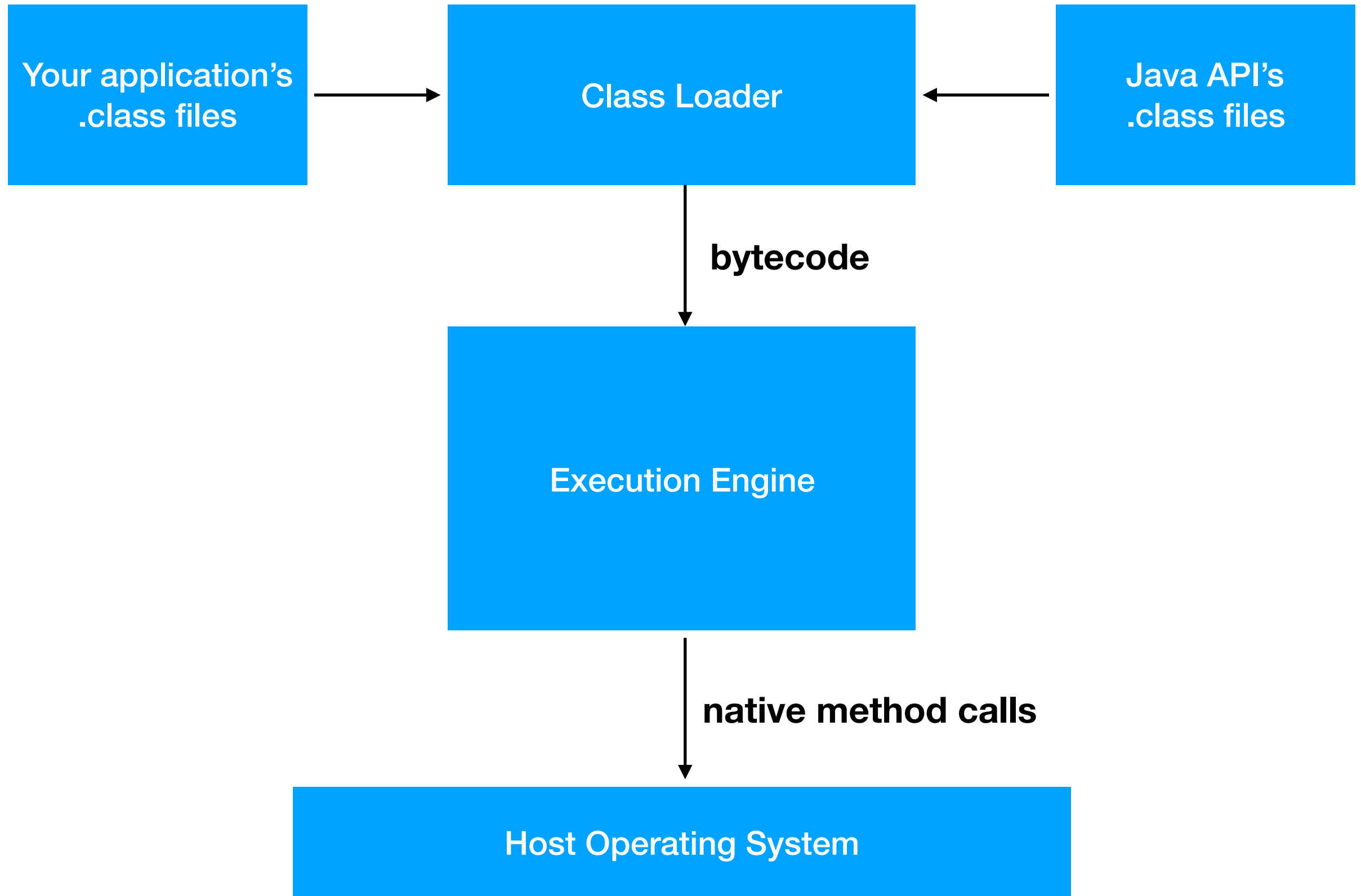
- **The Java virtual machine knows nothing of the Java programming language, only of a particular binary format, the class file format.**
- **A class file contains Java virtual machine instructions (or bytecodes) and a symbol table, as well as other ancillary information.**

Java Virtual Machine

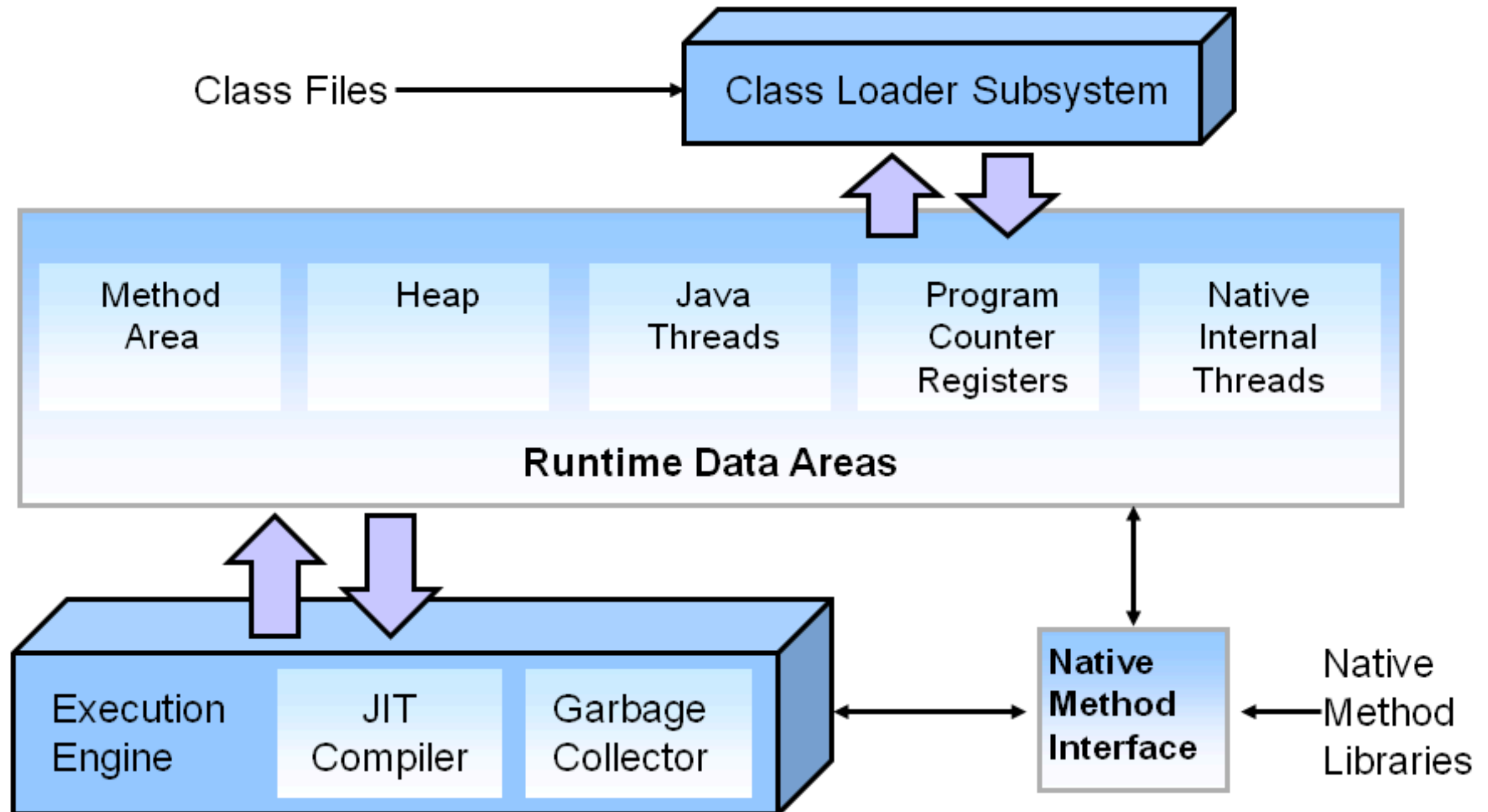
- **For the sake of security, the Java virtual machine imposes strong syntactic and structural constraints on the code in a class file.**
- However, any language with functionality that can be expressed in terms of a valid class file can be hosted by the Java virtual machine.
- Attracted by a generally available, machine-independent platform, implementors of other languages can turn to the Java virtual machine as a delivery vehicle for their languages.

Java Virtual Machine

- **The HotSpot JVM possesses an architecture that supports a strong foundation of features and capabilities and supports the ability to realize high performance and massive scalability.**
- For example, the HotSpot JVM JIT compilers generate dynamic optimizations.
 - In other words, they make optimization decisions while the Java application is running and generate high-performing native machine instructions targeted for the underlying system architecture.
- In addition, through the maturing evolution and continuous engineering of its runtime environment and multithreaded garbage collector, the HotSpot JVM yields high scalability on even the largest available computer systems.

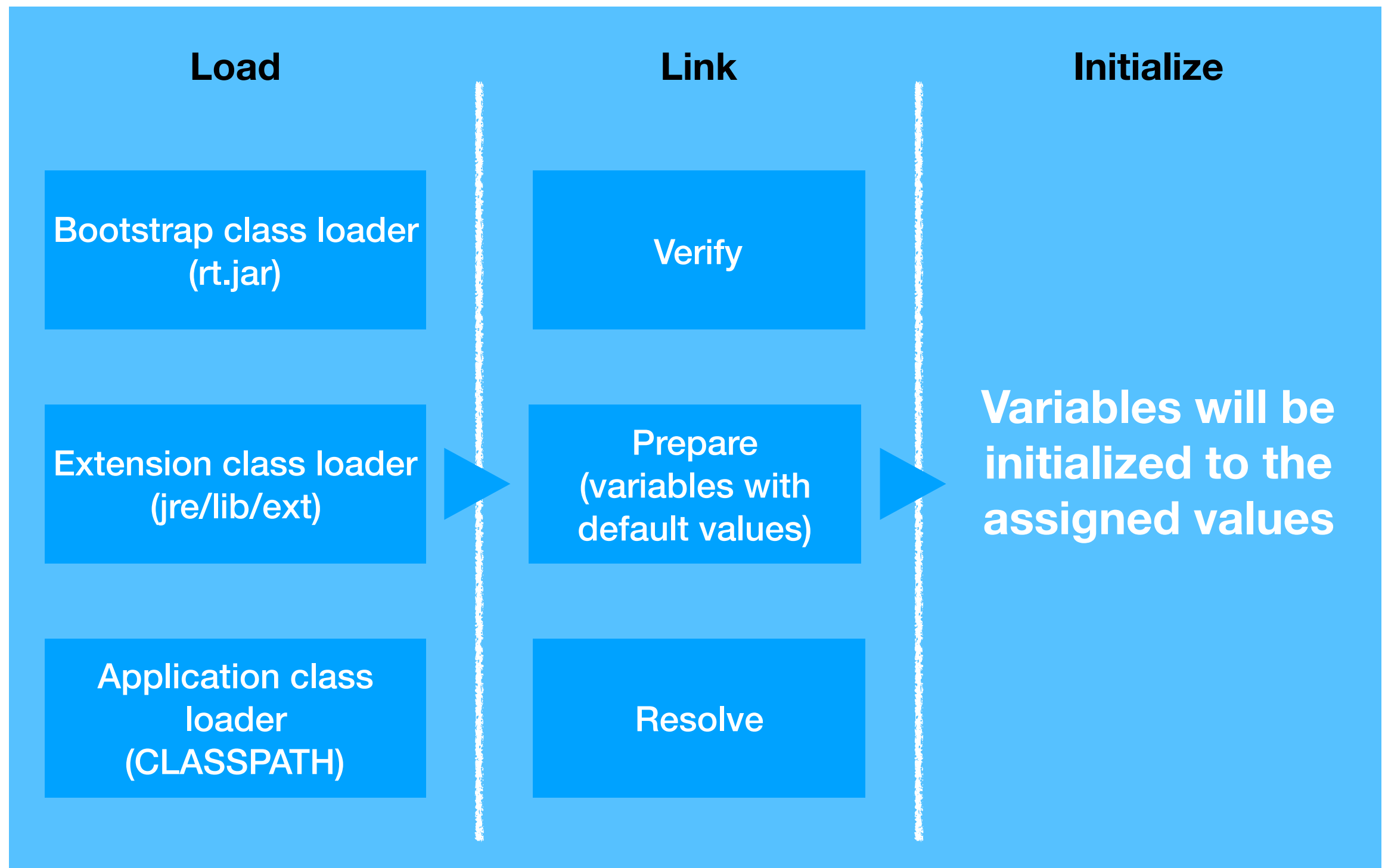


HotSpot JVM: Architecture

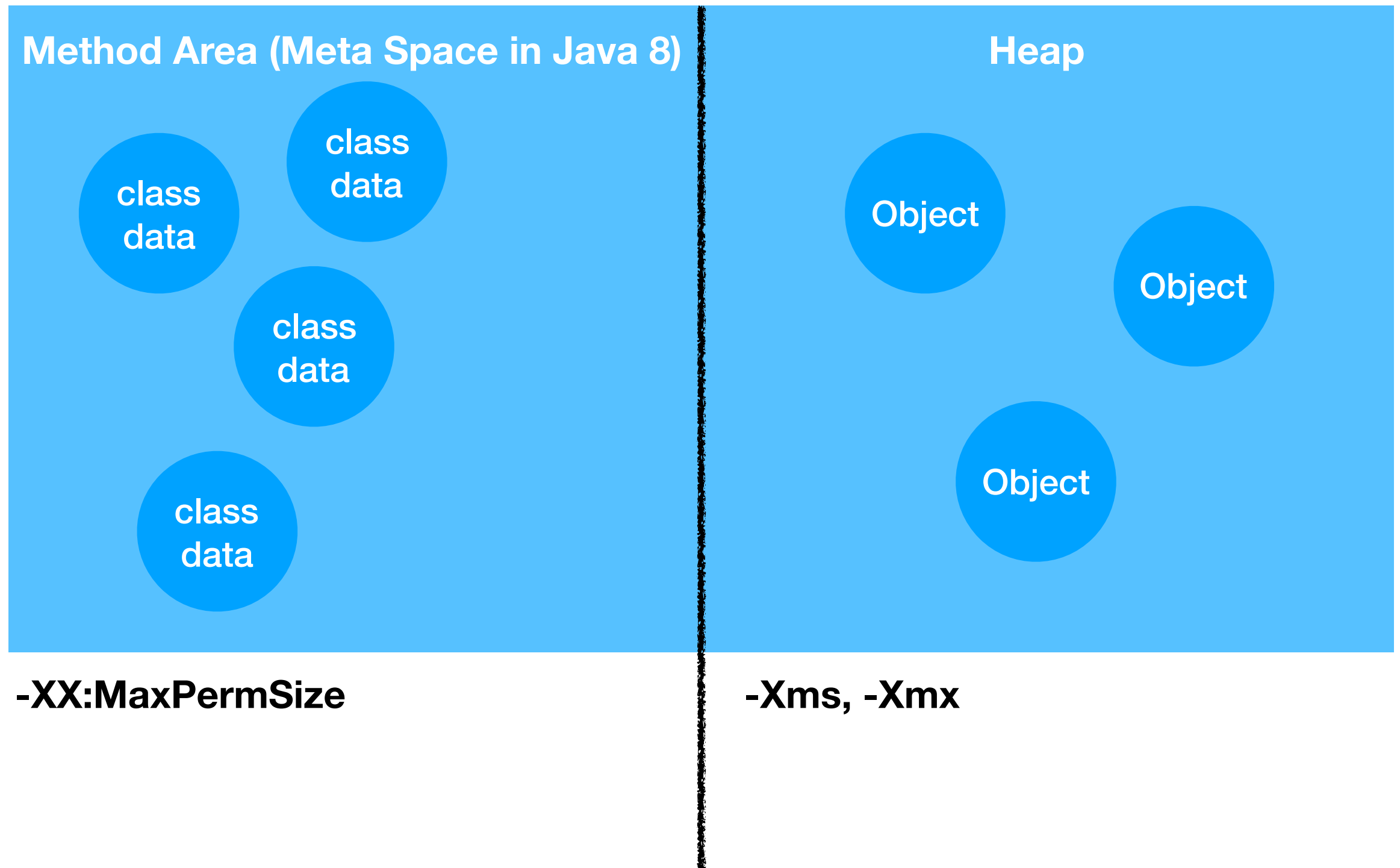


The main components of the JVM include the class loader, the runtime data areas, and the execution engine.

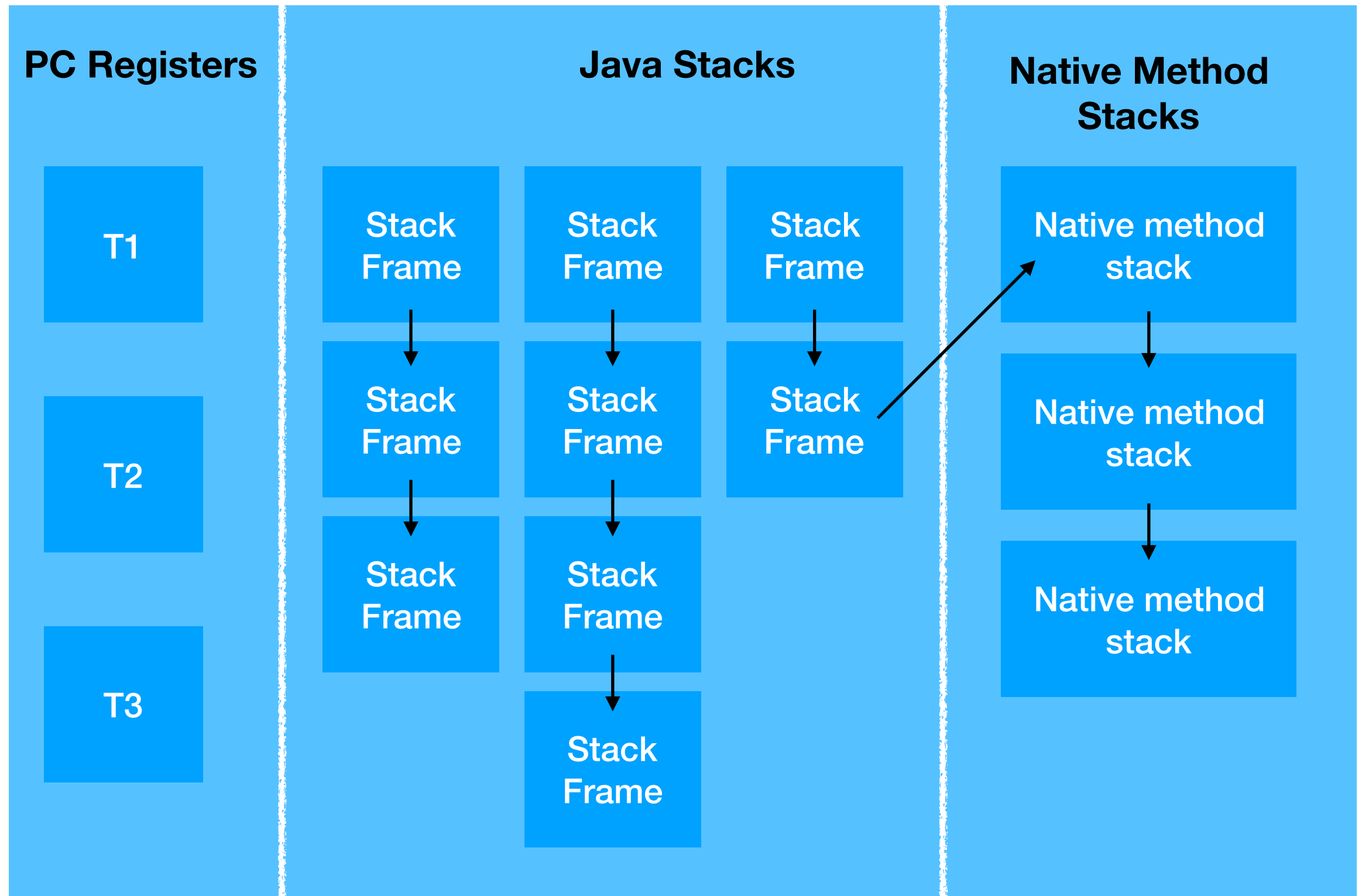
Class loader sub system



Runtime Data Areas (1 of 2)



Runtime Data Areas (2 of 2)



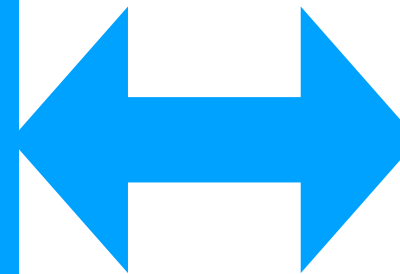
Execution Engine

Interpreter

HotSpot
Profiler

JIT
Compiler

Garbage
Collector

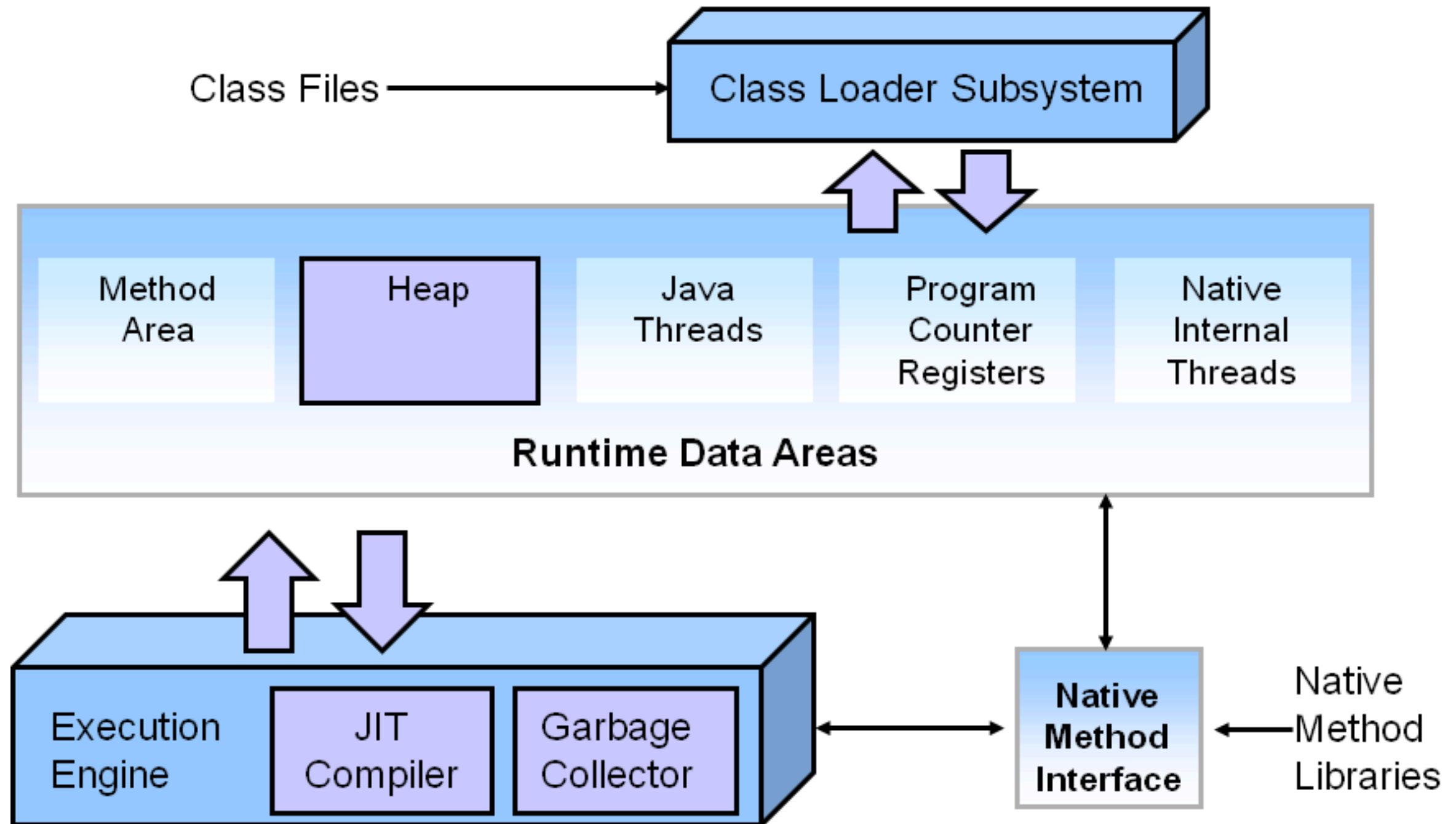


Native
Method
Interface
(JNI)

Native
method
libraries

(.dll
.so)

Key HotSpot JVM Components



Java Virtual Machine

- **There are three components of the JVM that are focused on when tuning performance.**
 - Heap, JIT Compiler and Garbage Collector
 - The heap is where your object data is stored.
 - This area is then managed by the garbage collector selected at startup.
 - Most tuning options relate to sizing the heap and choosing the most appropriate garbage collector for your situation.
 - The JIT compiler also has a big impact on performance but rarely requires tuning with the newer versions of the JVM.

Performance Tuning

- **Typically, when tuning a Java application, the focus is on one of two main goals:**
 - Responsiveness
 - Throughput.

Responsiveness

- **Responsiveness refers to how quickly an application or system responds with a requested piece of data. Examples include:**
 - How quickly a desktop UI responds to an event
 - How fast a website returns a page
 - How fast a database query is returned
- **For applications that focus on responsiveness, large pause times are not acceptable. The focus is on responding in short periods of time.**

Throughput

- **Throughput focuses on maximizing the amount of work by an application in a specific period of time. Examples of how throughput might be measured include:**
 - The number of transactions completed in a given time.
 - The number of jobs that a batch program can complete in an hour.
 - The number of database queries that can be completed in an hour.
- **High pause times are acceptable for applications that focus on throughput. Since high throughput applications focus on benchmarks over longer periods of time, quick response time is not a consideration.**

Garbage Collection and its importance

Before Java

- **Popular languages:**
 - C, C++
- **Memory allocation and de-allocation was program's responsibility**
 - malloc(), realloc(), calloc()
 - free()
 - new, delete, constructors, destructors

Garbage collection

- **Java provides automatic memory management through a program called “Garbage Collector”.**
 - Removes objects that are not used anymore
 - live object = reachable (referenced by variables somewhere)
 - dead object = unreachable (no references anywhere)

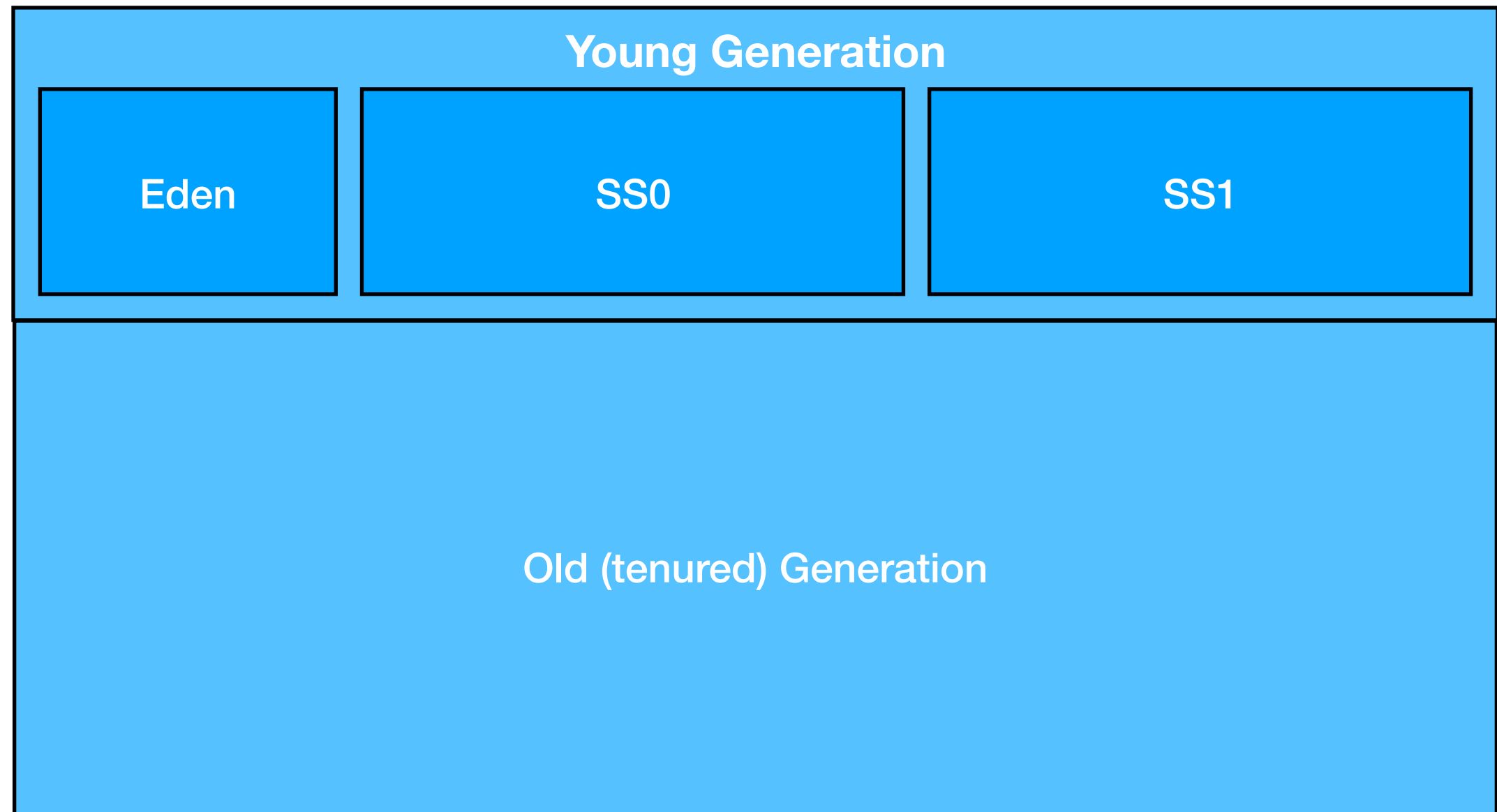
Garbage collection

- **Objects are allocated in the heap**
- **Static members, class definitions (metadata) etc are stored in method area (perm-gen, meta space)**
- **Garbage collection is carried out by a daemon thread called “Garbage Collector”**
- **System.gc()**
- **java.lang.OutOfMemoryError**

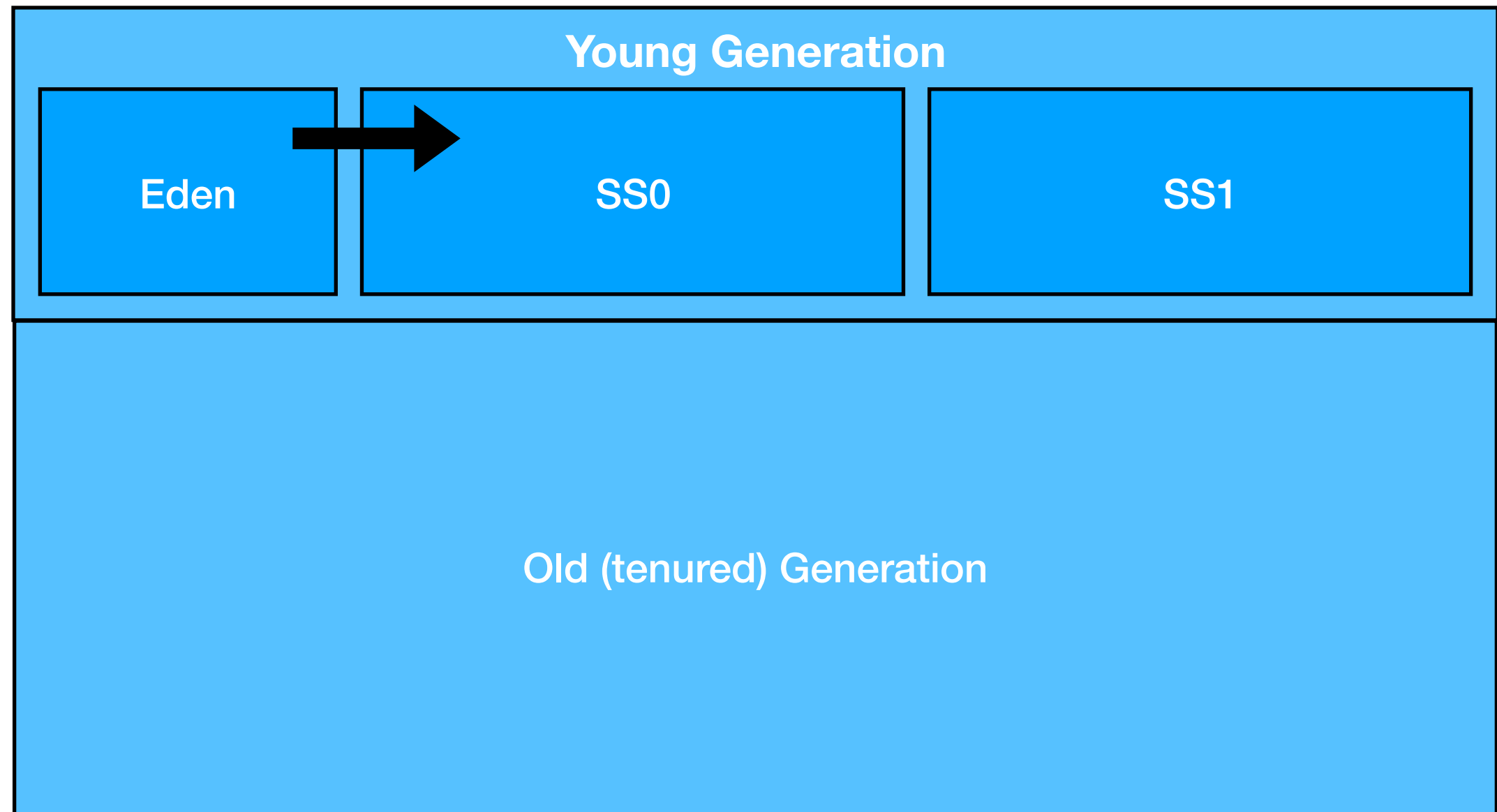
Garbage collection

- **Involves:**
 - Mark
 - Starts from a root node of your application, walks the object graph, marks objects that are reachable (live)
 - Sweep
 - Deletes unreachable objects
 - Compaction
 - Compact the memory by moving the live objects around, such that all the objects are placed contiguously (rather than fragmented)

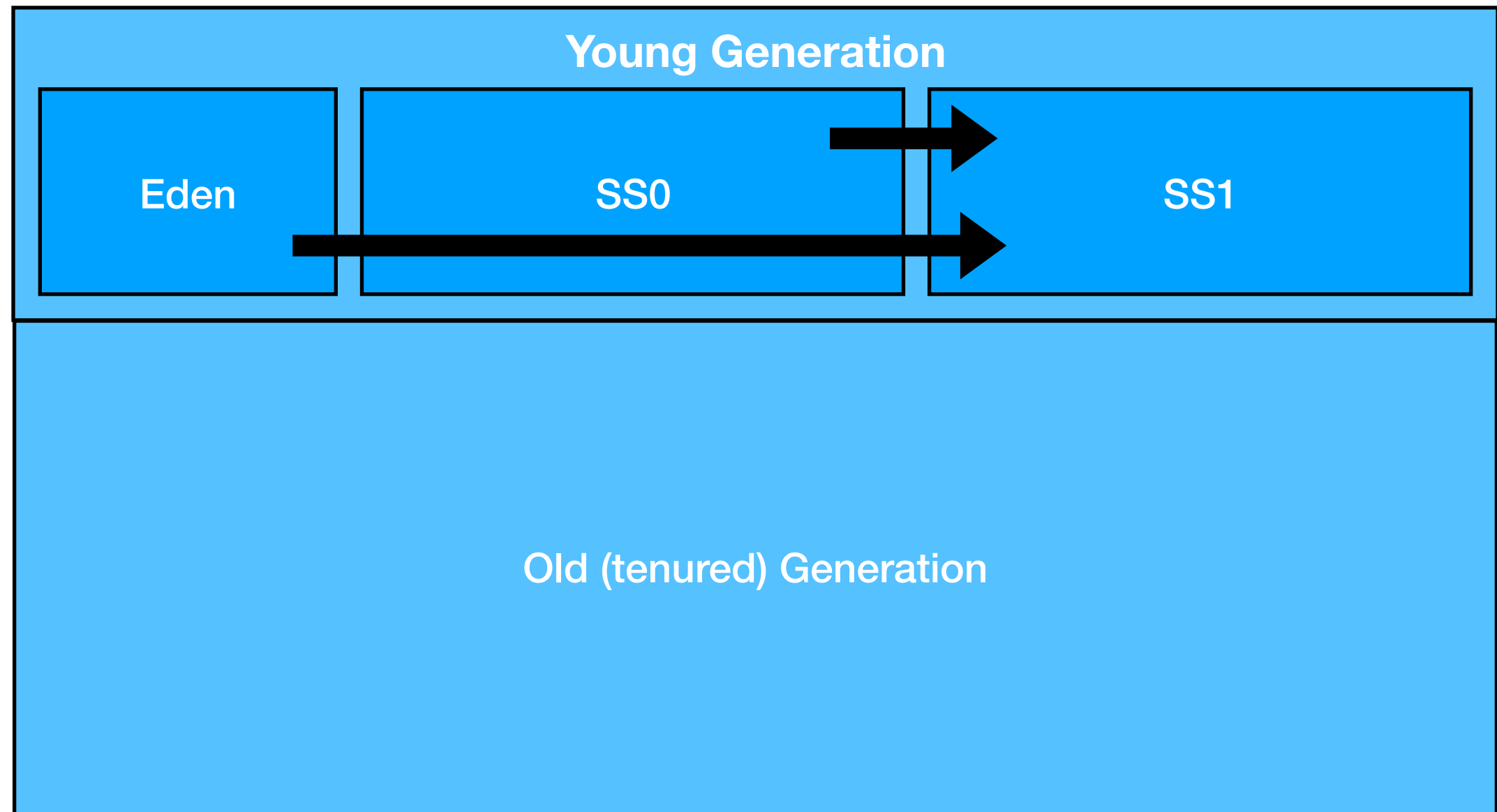
Generational collectors



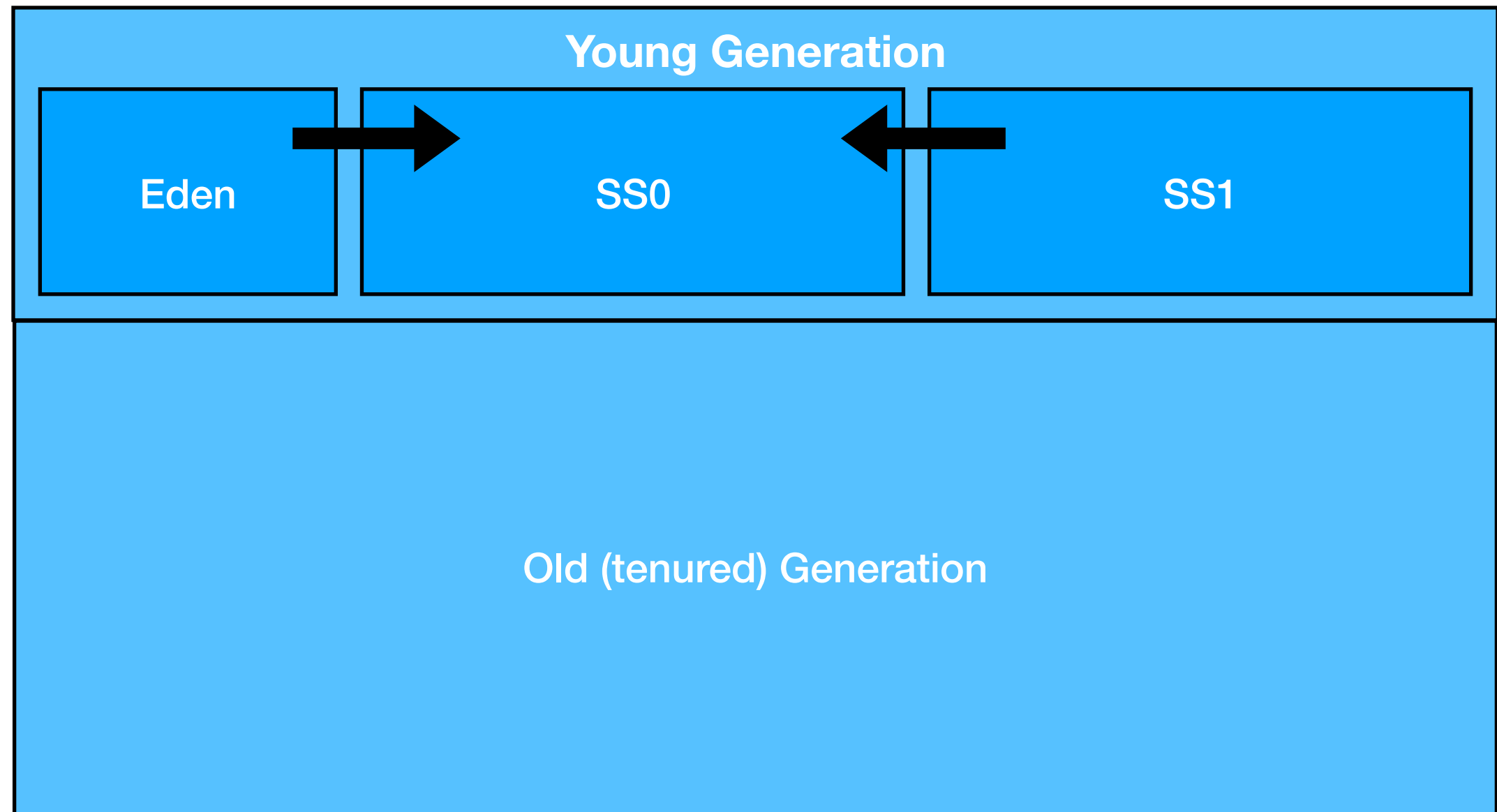
Generational collectors



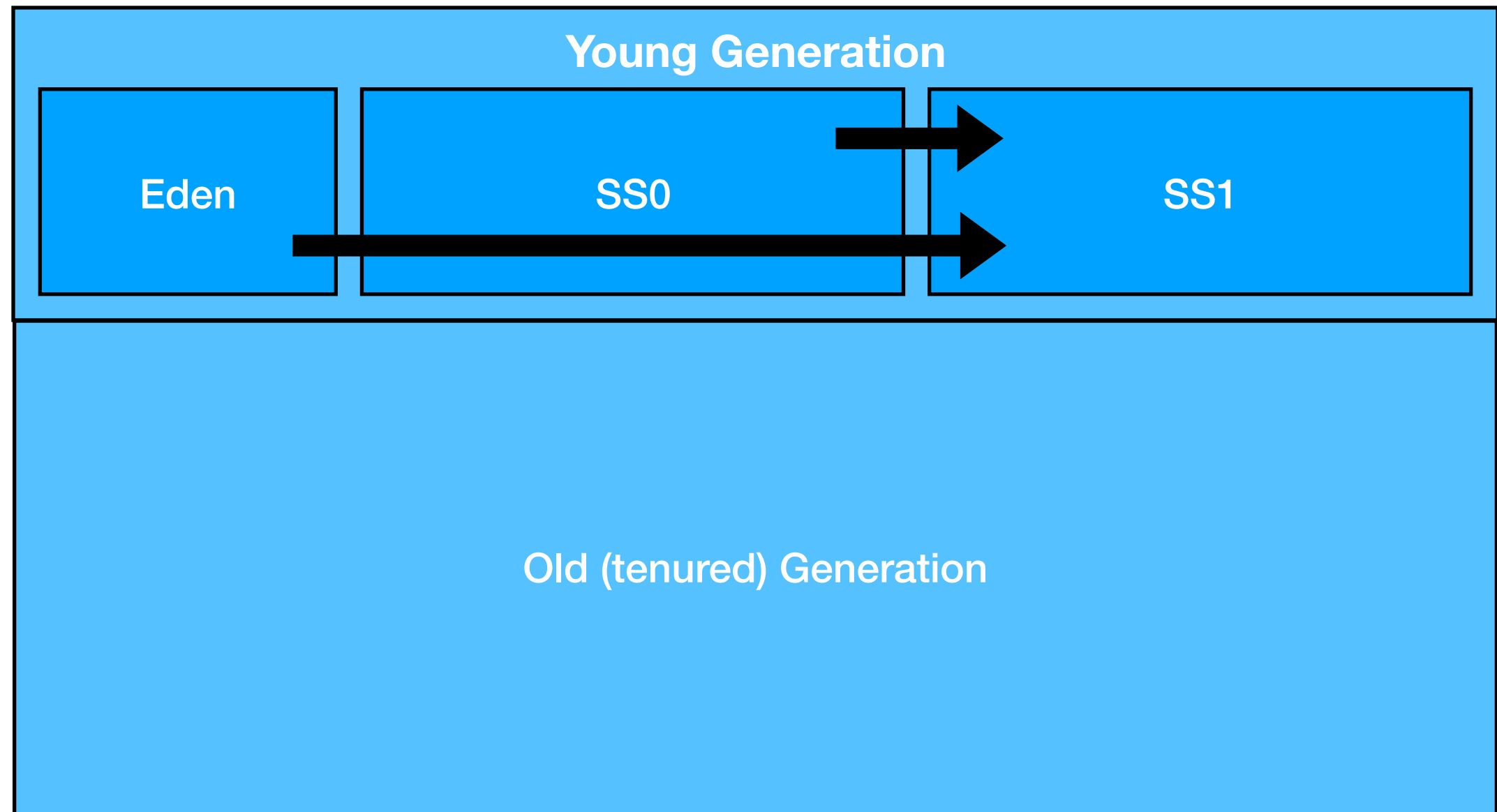
Generational collectors



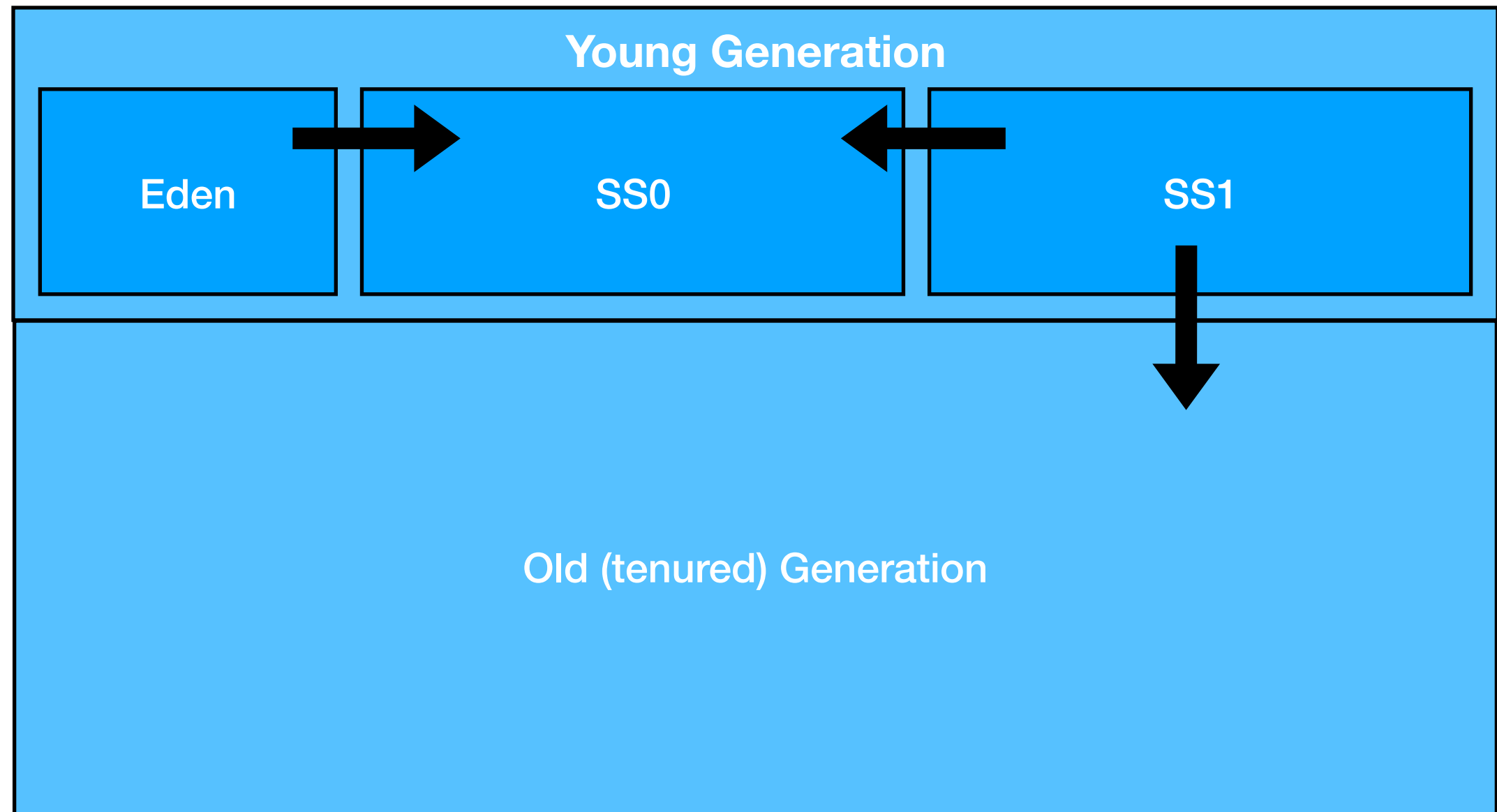
Generational collectors



Generational collectors



Generational collectors



Concurrent Mark Sweep collector

- **The Concurrent Mark Sweep (CMS) collector (also referred to as the concurrent low pause collector) collects the tenured generation.**
 - It attempts to minimize the pauses due to garbage collection by doing most of the garbage collection work concurrently with the application threads.
 - Normally the concurrent low pause collector does not copy or compact the live objects.
 - A garbage collection is done without moving the live objects.
 - If fragmentation becomes a problem, allocate a larger heap.

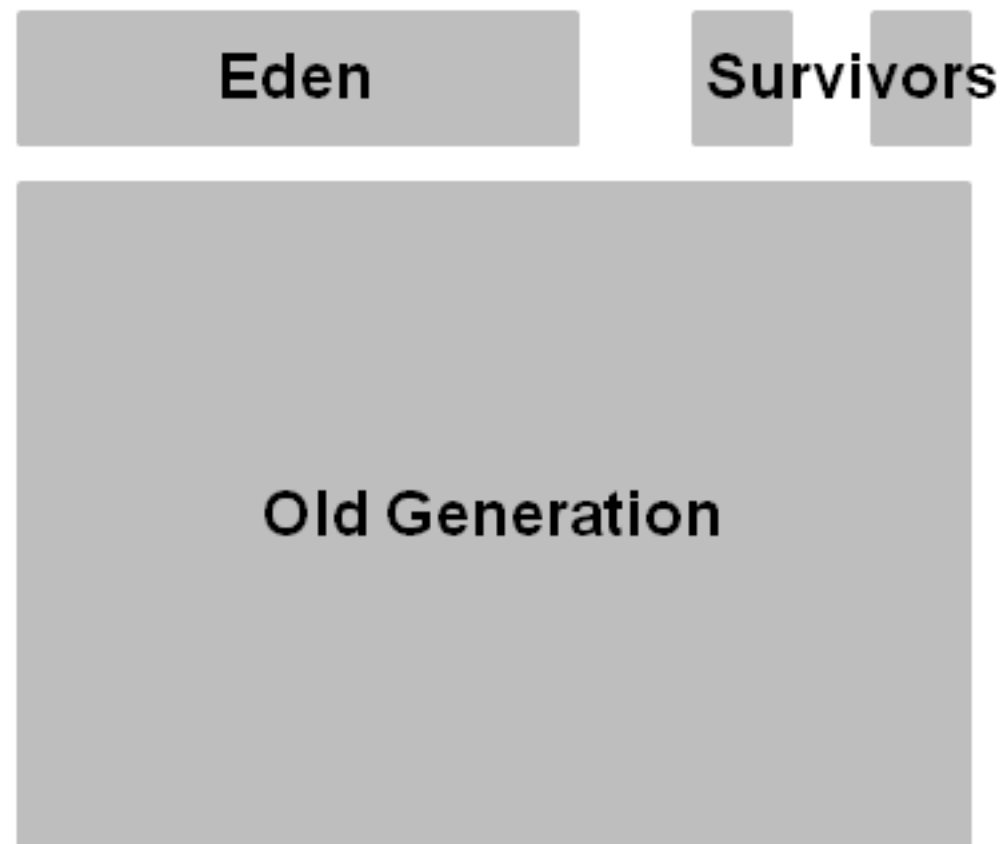
CMS collection phases (1 of 2)

- **The CMS collector performs the following phases on the old generation of the heap:**
 1. Initial Mark (Stop the World Event)
 - Objects in old generation are "marked" as reachable including those objects which may be reachable from young generation. Pause times are typically short in duration relative to minor collection pause times.
 2. Concurrent Marking
 - Traverse the tenured generation object graph for reachable objects concurrently while Java application threads are executing. Starts scanning from marked objects and transitively marks all objects reachable from the roots. The mutators are executing during the concurrent phases 2, 3, and 5 and any objects allocated in the CMS generation during these phases (including promoted objects) are immediately marked as live.

CMS collection phases (2 of 2)

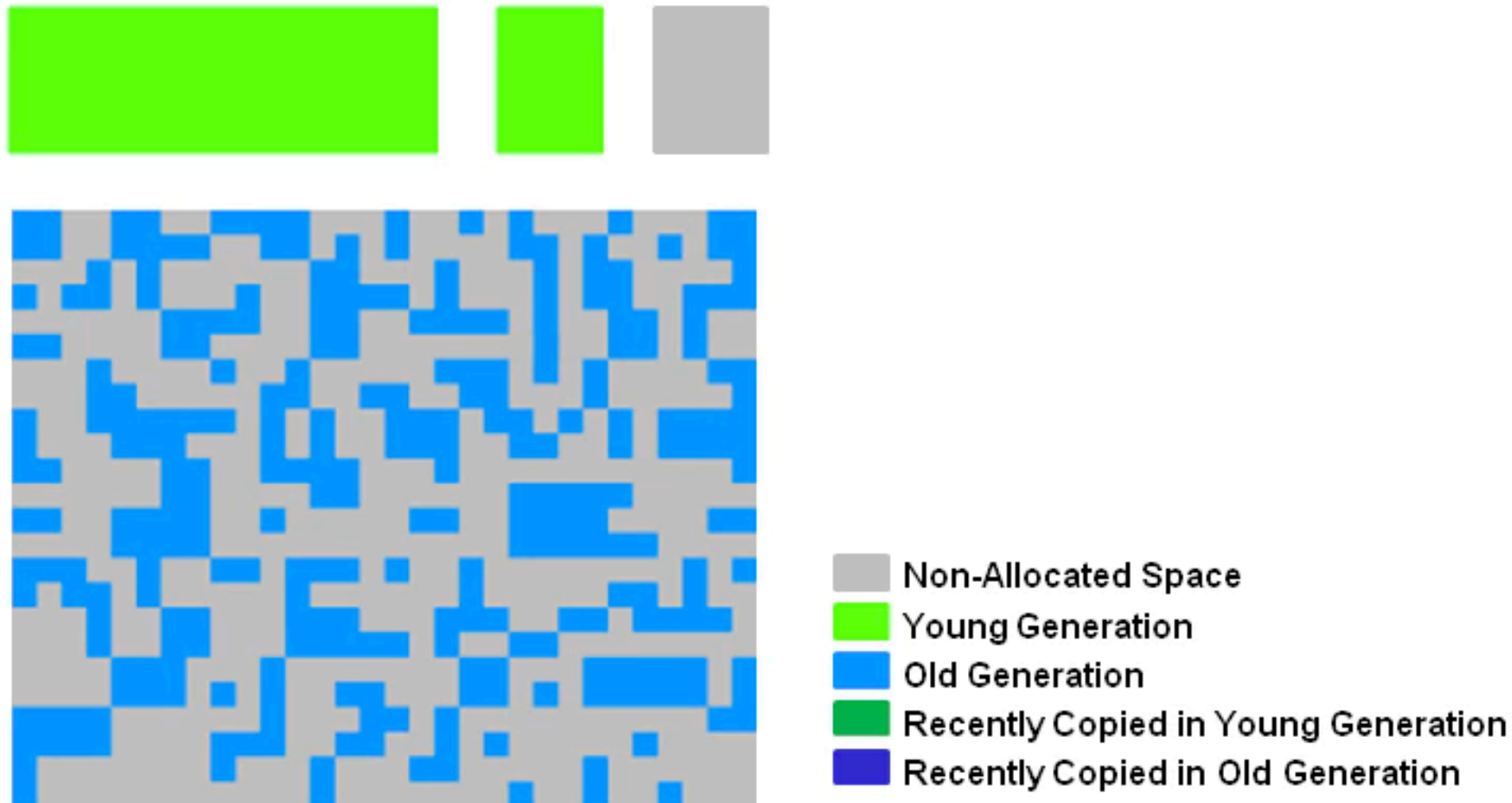
- **The CMS collector performs the following phases on the old generation of the heap:**
 3. Re-mark (Stop the World Event)
 - Finds objects that were missed by the concurrent mark phase due to updates by Java application threads to objects after the concurrent collector had finished tracing that object.
 4. Concurrent Sweep
 - Collects the objects identified as unreachable during marking phases. The collection of a dead object adds the space for the object to a free list for later allocation. Coalescing of dead objects may occur at this point. Note that live objects are not moved.
 5. Resetting
 - Prepare for next concurrent collection by clearing data structures.

CMS Heap Structure



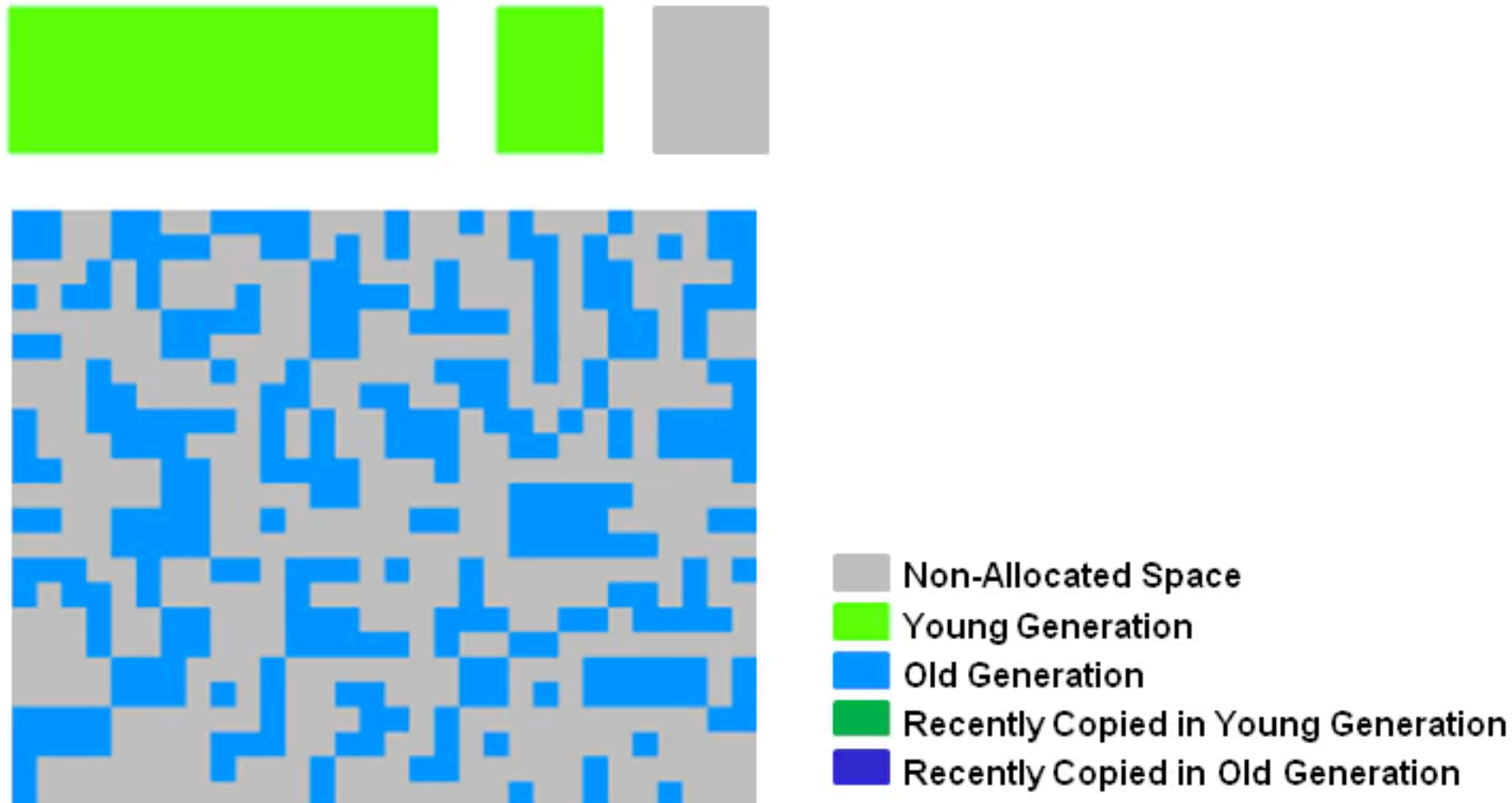
- **Heap Structure for CMS Collector**
 - The heap is split into three spaces.
 - Young generation is split into Eden and two survivor spaces.
 - Old generation is one contiguous space. Object collection is done in place.
 - No compaction is done unless there is a full GC.

How young GC Works



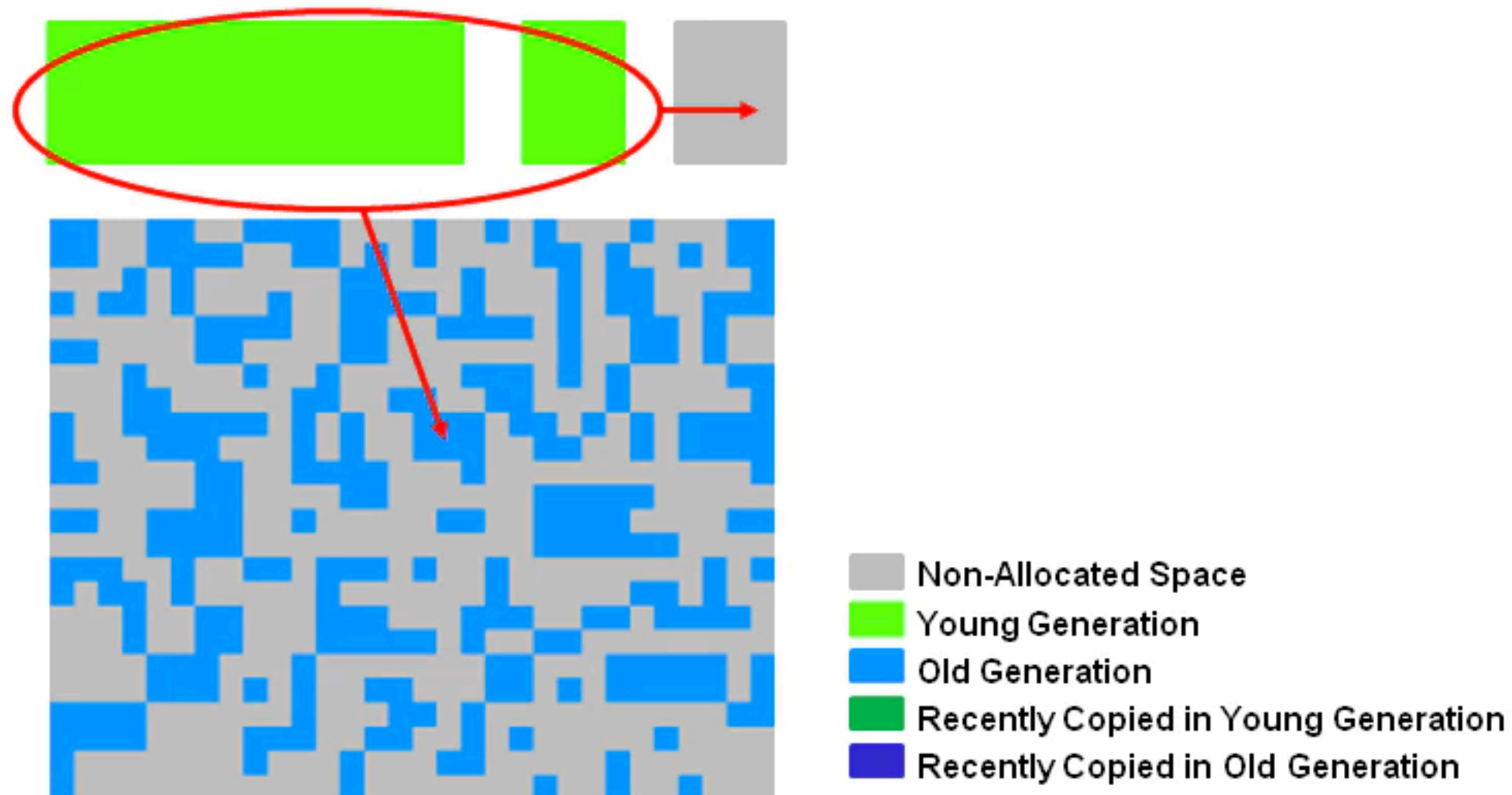
- The young generation is colored light green and the old generation in blue.
- This is what the CMS might look like if your application has been running for a while.
- Objects are scattered around the old generation area.

How young GC Works



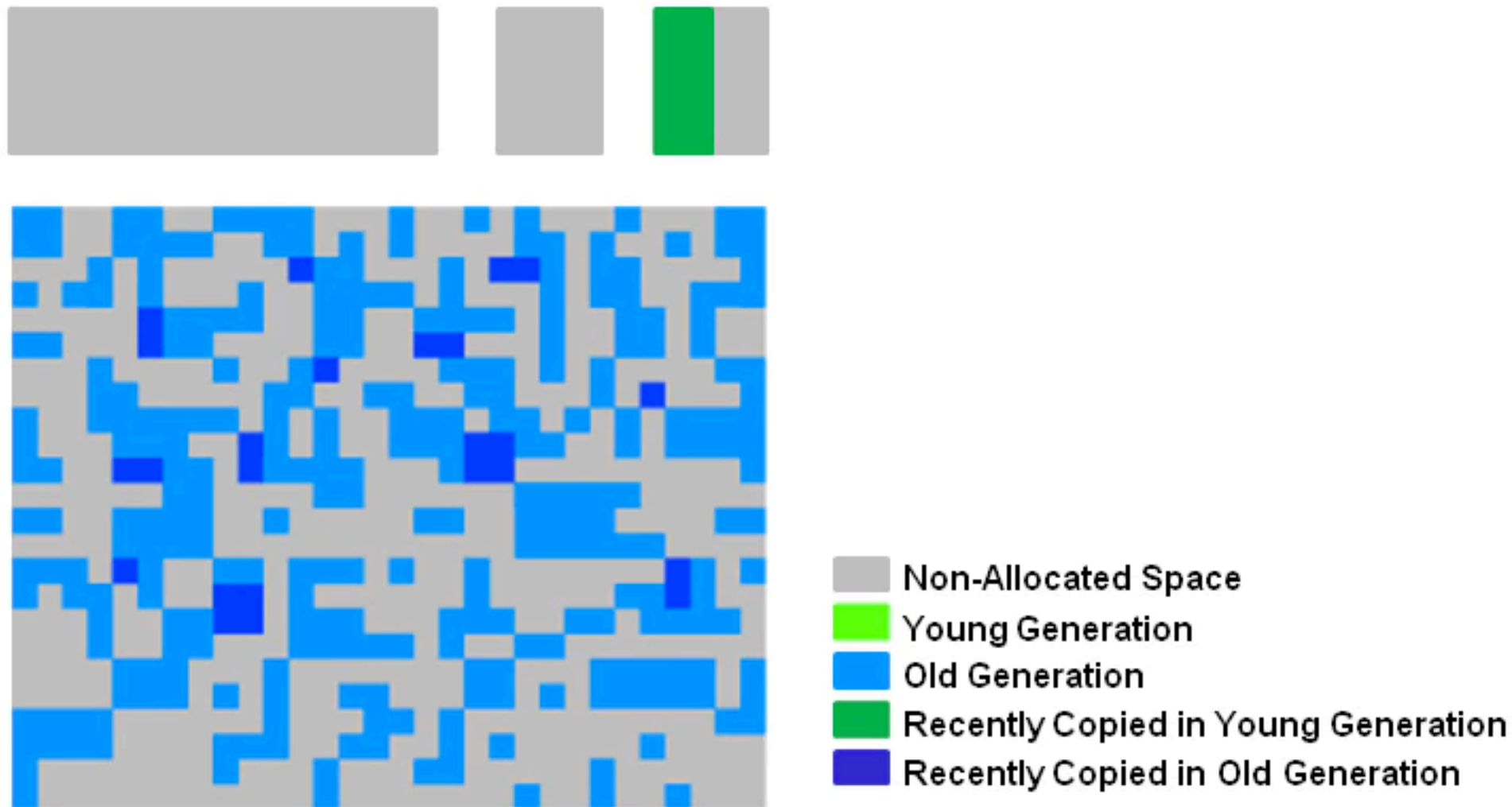
- **With CMS, old generation objects are deallocated in place.**
- **They are not moved around.**
- **The space is not compacted unless there is a full GC.**

Young Generation Collection



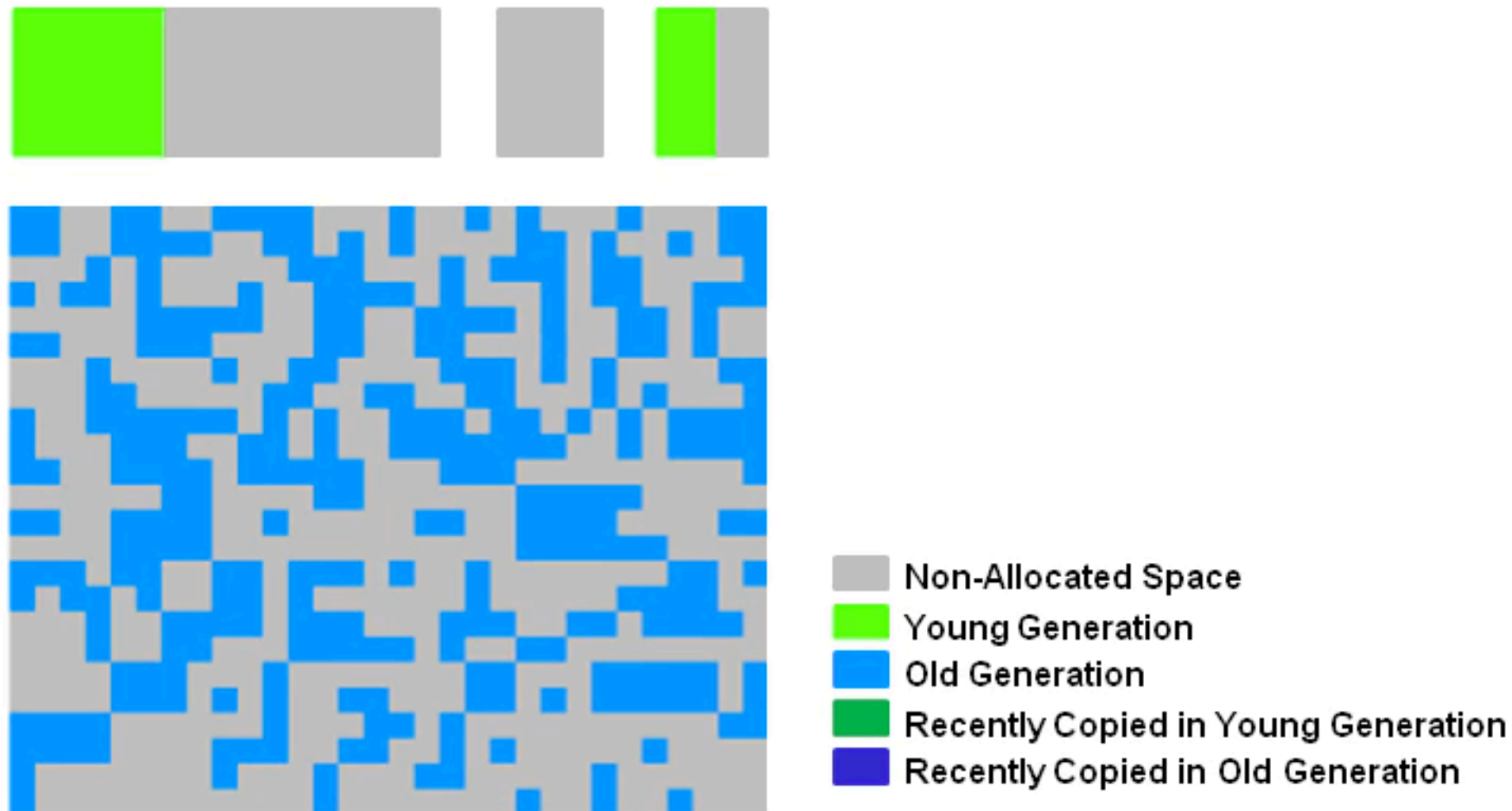
- Live objects are copied from the Eden space and survivor space to the other survivor space.
- Any older objects that have reached their aging threshold are promoted to old generation.

After Young GC



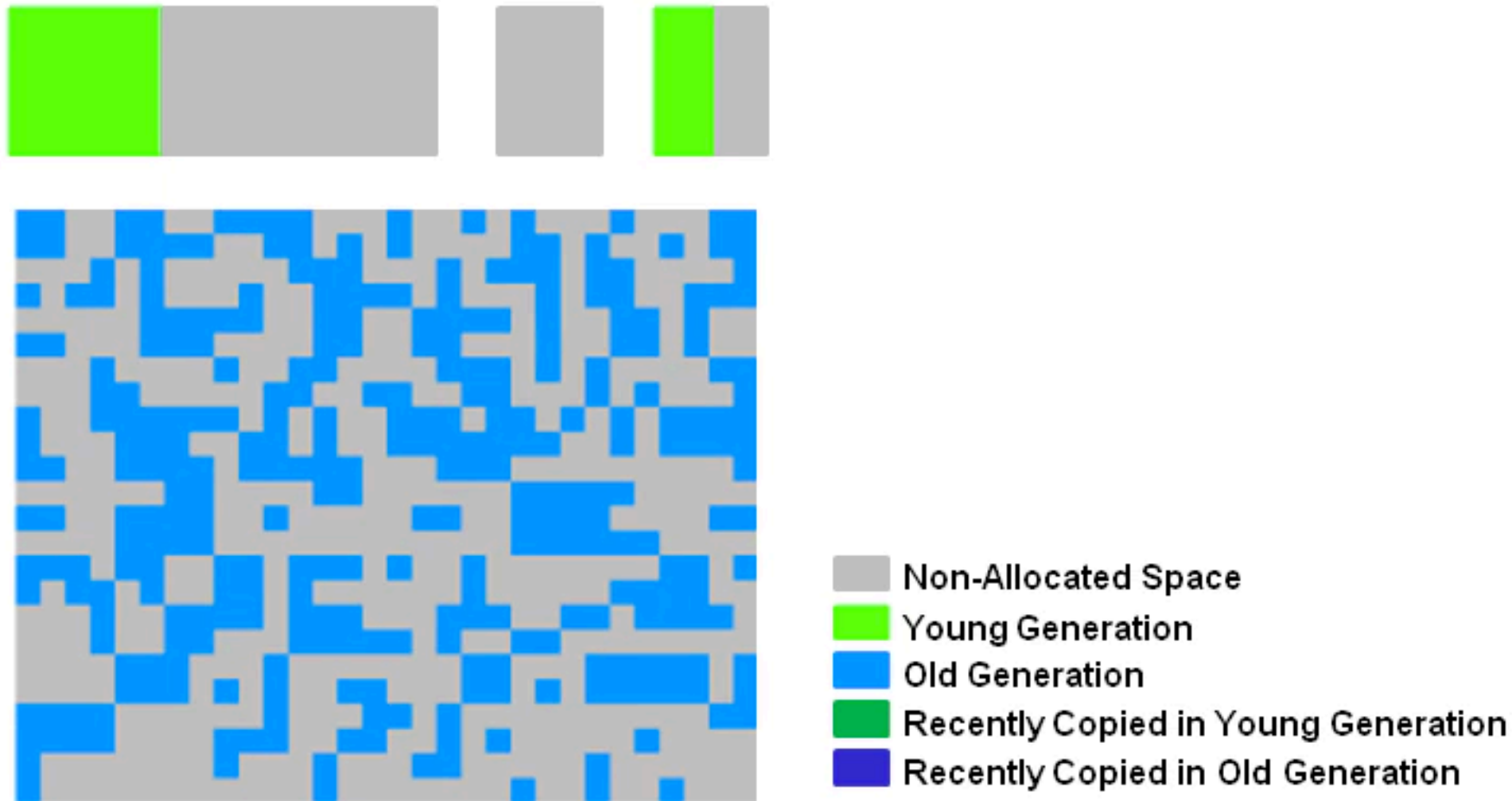
- After a young GC, the Eden space is cleared and one of the survivor spaces is cleared.
- Newly promoted objects are shown in dark blue on the diagram.
- The green objects are surviving young generation objects that have not yet been promoted to old generation.

Old gen collection in CMS



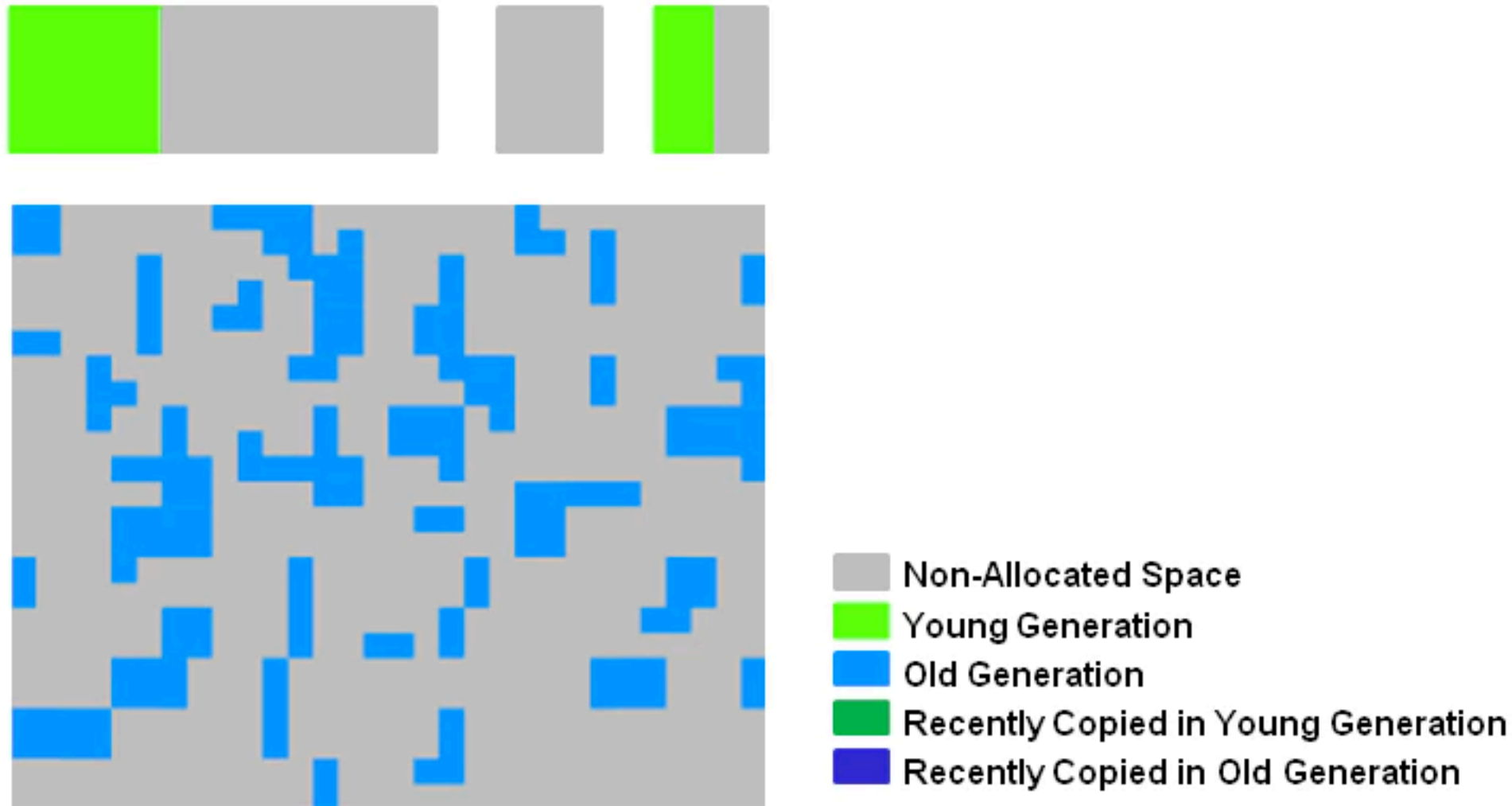
- **Two stop the world events take place: initial mark and remark. When the old generation reaches a certain occupancy rate, the CMS is kicked off.**
 1. **Initial mark is a short pause phase where live (reachable) objects are marked.**
 2. **Concurrent marking finds live objects while the application continues to execute.**
 3. **Finally, in the remark phase, objects are found that were missed during concurrent marking in the previous phase.**

Old Gen Collection – Concurrent Sweep



- **Old Generation Collection - Concurrent Sweep:**
 1. Objects that were not marked in the previous phase are deallocated in place. There is no compaction.
 2. Note: Unmarked objects == Dead Objects

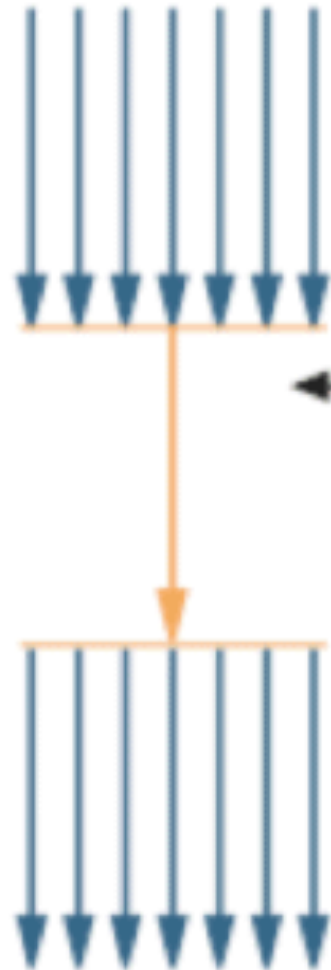
Old Gen Collection – After Sweeping



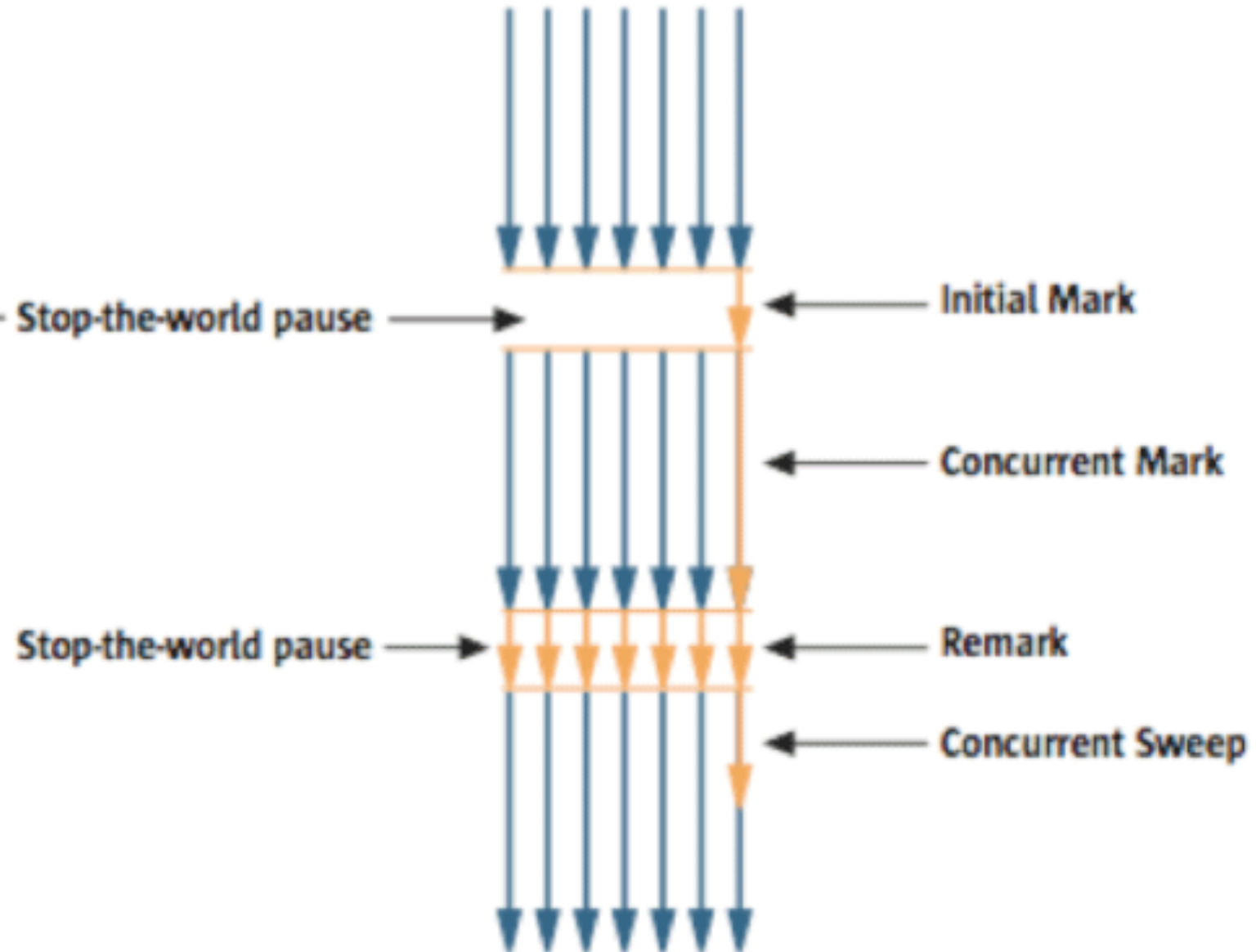
- **Old Generation Collection - After Sweeping:**
 1. After the Sweeping phase, you can see that a lot of memory has been freed up. You will also notice that no compaction has been done.
 2. Finally, the CMS collector will move through the resetting phase and wait for the next time the GC threshold is reached.

GC Pause times comparison

Serial Mark-Sweep-Compact Collector



Concurrent Mark-Sweep Collector



The Garbage-First (G1) collector

The Garbage-First (G1) collector

- **The Garbage-First (G1) collector is a server-style garbage collector, targeted for multi-processor machines with large memories.**
- **It meets garbage collection (GC) pause time goals with a high probability, while achieving high throughput.**
- **The G1 garbage collector is fully supported in Oracle JDK 7 update 4 and later releases.**

The Garbage-First (G1) collector

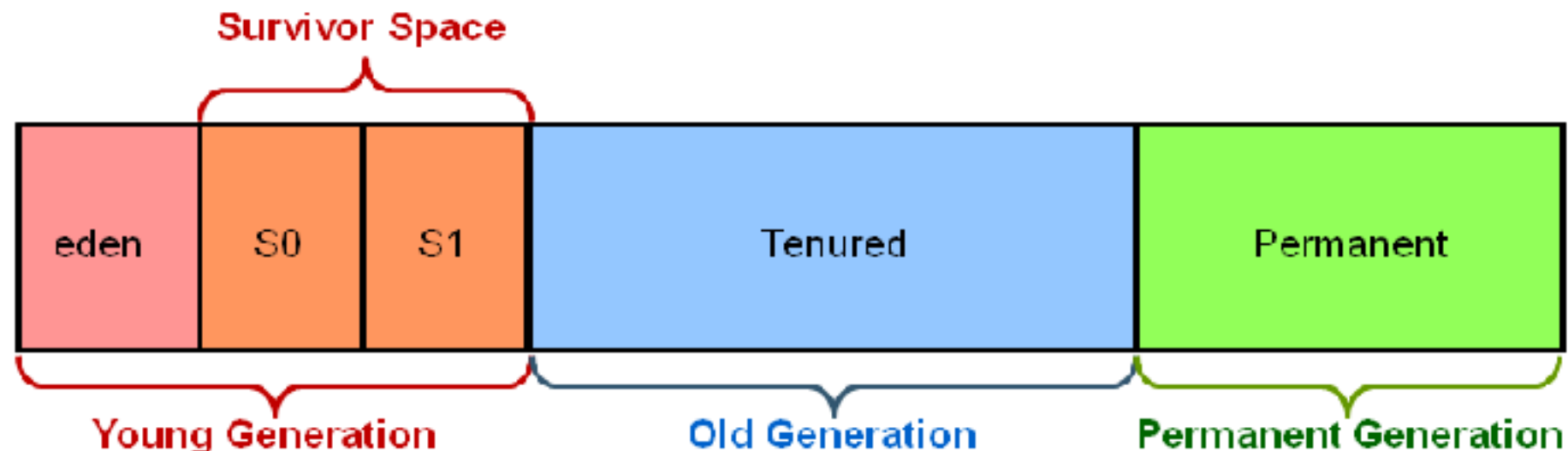
- **The G1 collector is designed for applications that:**
 - Can operate concurrently with applications threads (just like the CMS collector).
 - Compact free space without lengthy GC induced pause times.
 - Need more predictable GC pause durations.
 - Do not want to sacrifice a lot of throughput performance.
 - Do not require a much larger Java heap.

The Garbage-First (G1) collector

- **G1 is planned as the long term replacement for the Concurrent Mark-Sweep Collector (CMS).**
- **Comparing G1 with CMS, there are differences that make G1 a better solution.**
 - One difference is that G1 is a compacting collector.
 - G1 compacts sufficiently to completely avoid the use of fine-grained free lists for allocation, and instead relies on regions.
 - This considerably simplifies parts of the collector, and mostly eliminates potential fragmentation issues.
- **Also, G1 offers more predictable garbage collection pauses than the CMS collector, and allows users to specify desired pause targets.**

G1 Operational Overview

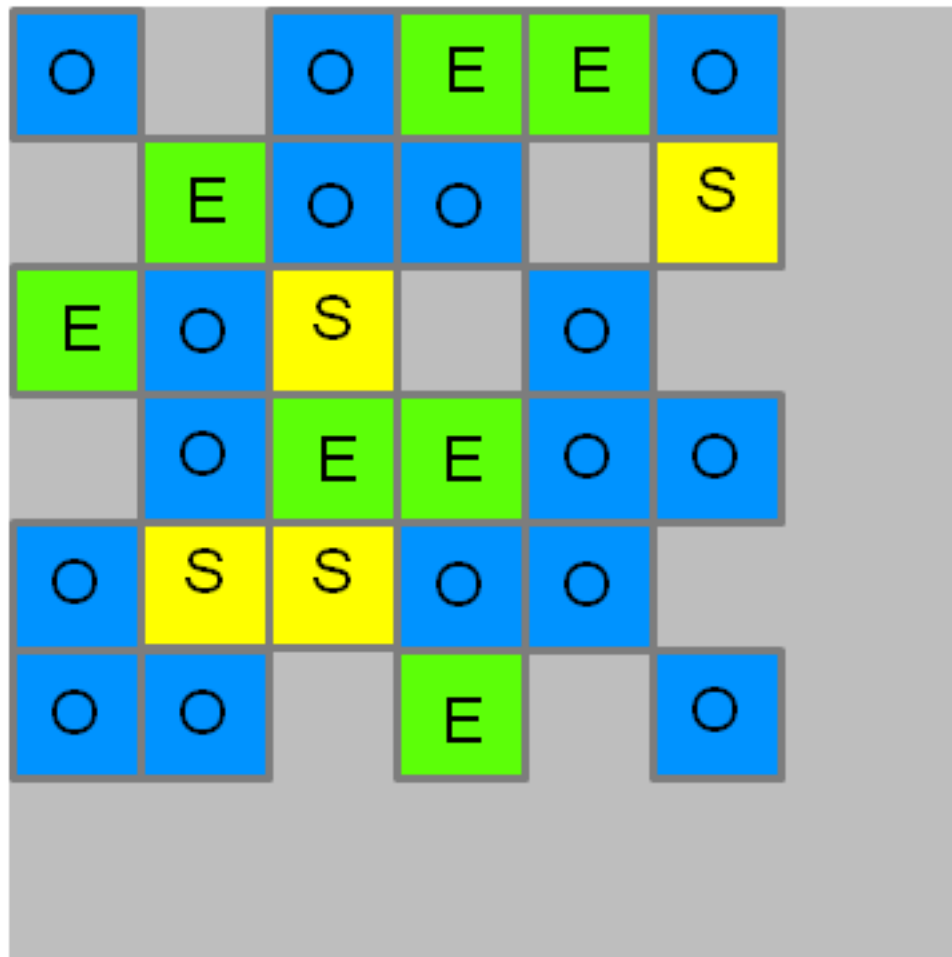
- The older garbage collectors (serial, parallel, CMS) all structure the heap into three sections of a fixed memory size:
 - Young generation
 - Old generation
 - Permanent generation
- All memory objects end up in one of these three sections.



G1 Operational Overview

- **The G1 collector takes a different approach.**
 - The heap is partitioned into a set of equal-sized heap regions, each a contiguous range of virtual memory.
 - Certain region sets are assigned the same roles (eden, survivor, old) as in the older collectors, but there is not a fixed size for them.
 - This provides greater flexibility in memory usage.

G1 Heap Allocation



Eden Space

Survivor Space

Old Generation

G1 Operational Overview

- **When performing garbage collections, G1 operates in a manner similar to the CMS collector.**
 - G1 performs a concurrent global marking phase to determine the liveness of objects throughout the heap.
 - After the mark phase completes, G1 knows which regions are mostly empty.
 - It collects in these regions first, which usually yields a large amount of free space. This is why this method of garbage collection is called Garbage-First.
 - As the name suggests, G1 concentrates its collection and compaction activity on the areas of the heap that are likely to be full of reclaimable objects, that is, garbage.
 - G1 uses a pause prediction model to meet a user-defined pause time target and selects the number of regions to collect based on the specified pause time target.

G1 Operational Overview

- **The regions identified by G1 as ripe for reclamation are garbage collected using evacuation.**
 - G1 copies objects from one or more regions of the heap to a single region on the heap, and in the process both compacts and frees up memory.
 - This evacuation is performed in parallel on multi-processors, to decrease pause times and increase throughput.
 - Thus, with each garbage collection, G1 continuously works to reduce fragmentation, working within the user defined pause times.
 - This is beyond the capability of both the previous methods.
 - CMS (Concurrent Mark Sweep) garbage collector does not do compaction.
 - ParallelOld garbage collection performs only whole-heap compaction, which results in considerable pause times.

G1 Operational Overview

- **It is important to note that G1 is not a real-time collector.**
 - It meets the set pause time target with high probability but not absolute certainty.
 - Based on data from previous collections, G1 does an estimate of how many regions can be collected within the user specified target time.
 - Thus, the collector has a reasonably accurate model of the cost of collecting the regions, and it uses this model to determine which and how many regions to collect while staying within the pause time target.

G1 Operational Overview

- **G1 has both concurrent (runs along with application threads, e.g., refinement, marking, cleanup) and parallel (multi-threaded, e.g., stop the world) phases.**
- Full garbage collections are still single threaded, but if tuned properly your applications should avoid full GCs.

G1 Footprint

- **If you migrate from the ParallelOldGC or CMS collector to G1, you will likely see a larger JVM process size.**
 - This is largely related to "accounting" data structures such as Remembered Sets and Collection Sets.

G1 Footprint

- **Remembered Sets or RSets track object references into a given region.**
 - There is one RSet per region in the heap.
 - The RSet enables the parallel and independent collection of a region. The overall footprint impact of RSets is less than 5%.
- **Collection Sets or CSets the set of regions that will be collected in a GC.**
 - All live data in a CSet is evacuated (copied/moved) during a GC.
 - Sets of regions can be Eden, survivor, and/or old generation.
 - CSets have a less than 1% impact on the size of the JVM.


G1 step by step

- **G1 Heap Structure**

- The heap is one memory area split into many fixed sized regions.
- Region size is chosen by the JVM at startup.
- The JVM generally targets around 2000 regions varying in size from 1 to 32Mb.

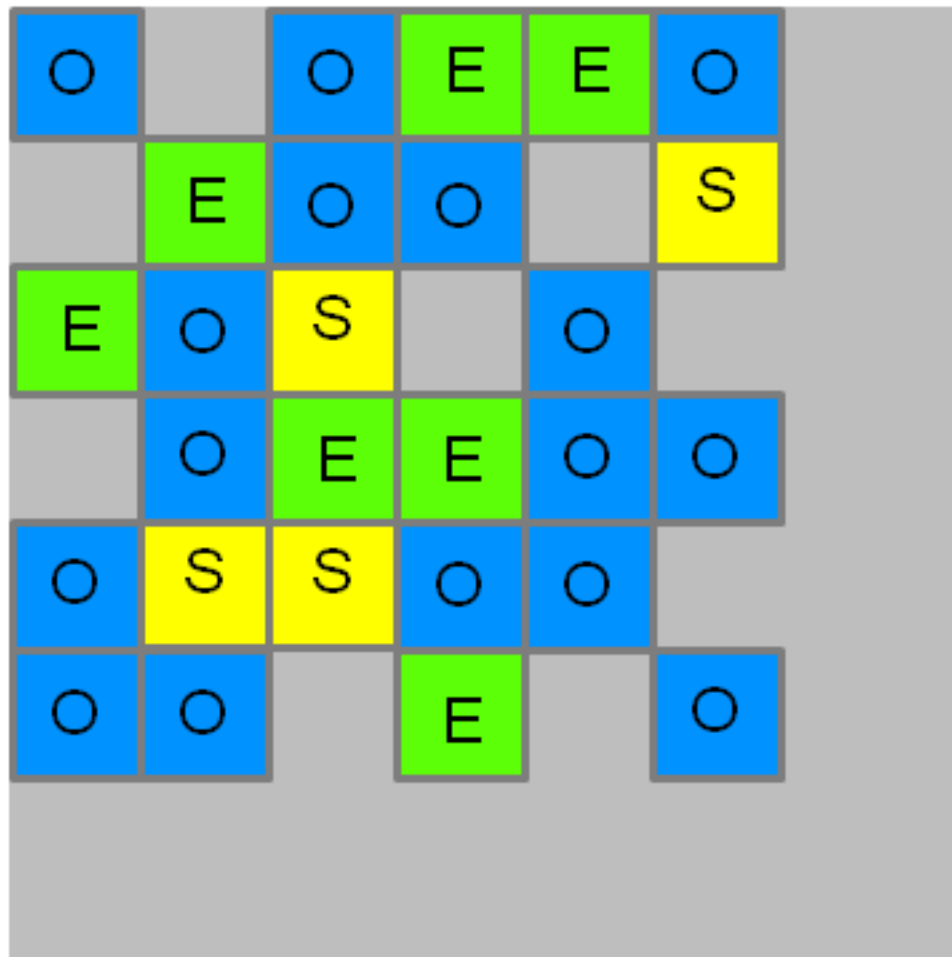
G1 step by step

G1 Heap Structure



One memory area
split into many fixed
sized regions

G1 Heap Allocation



Eden Space

Survivor Space

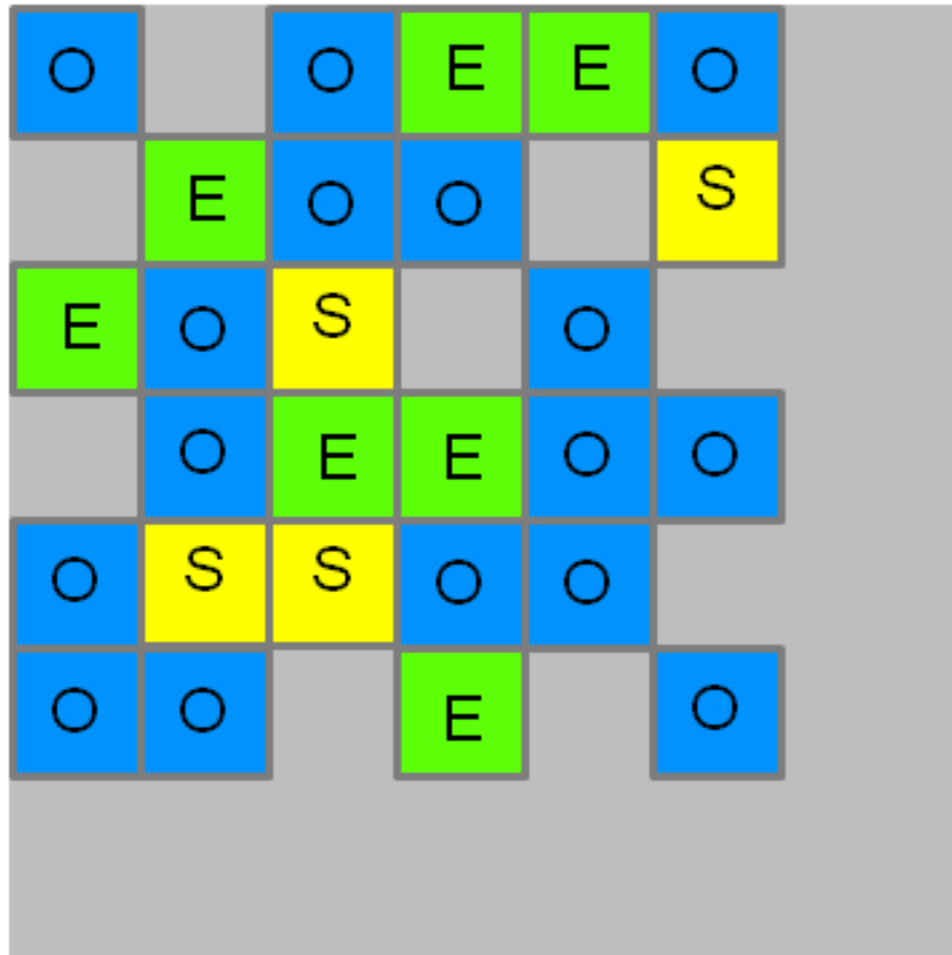
Old Generation

G1 step by step

- **G1 Heap Allocation**

- In reality, these regions are mapped into logical representations of Eden, Survivor, and old generation spaces.
- The colors in the picture shows which region is associated with which role. Live objects are evacuated (i.e., copied or moved) from one region to another.
- Regions are designed to be collected in parallel with or without stopping all other application threads.

G1 Heap Allocation



Eden Space

Survivor Space

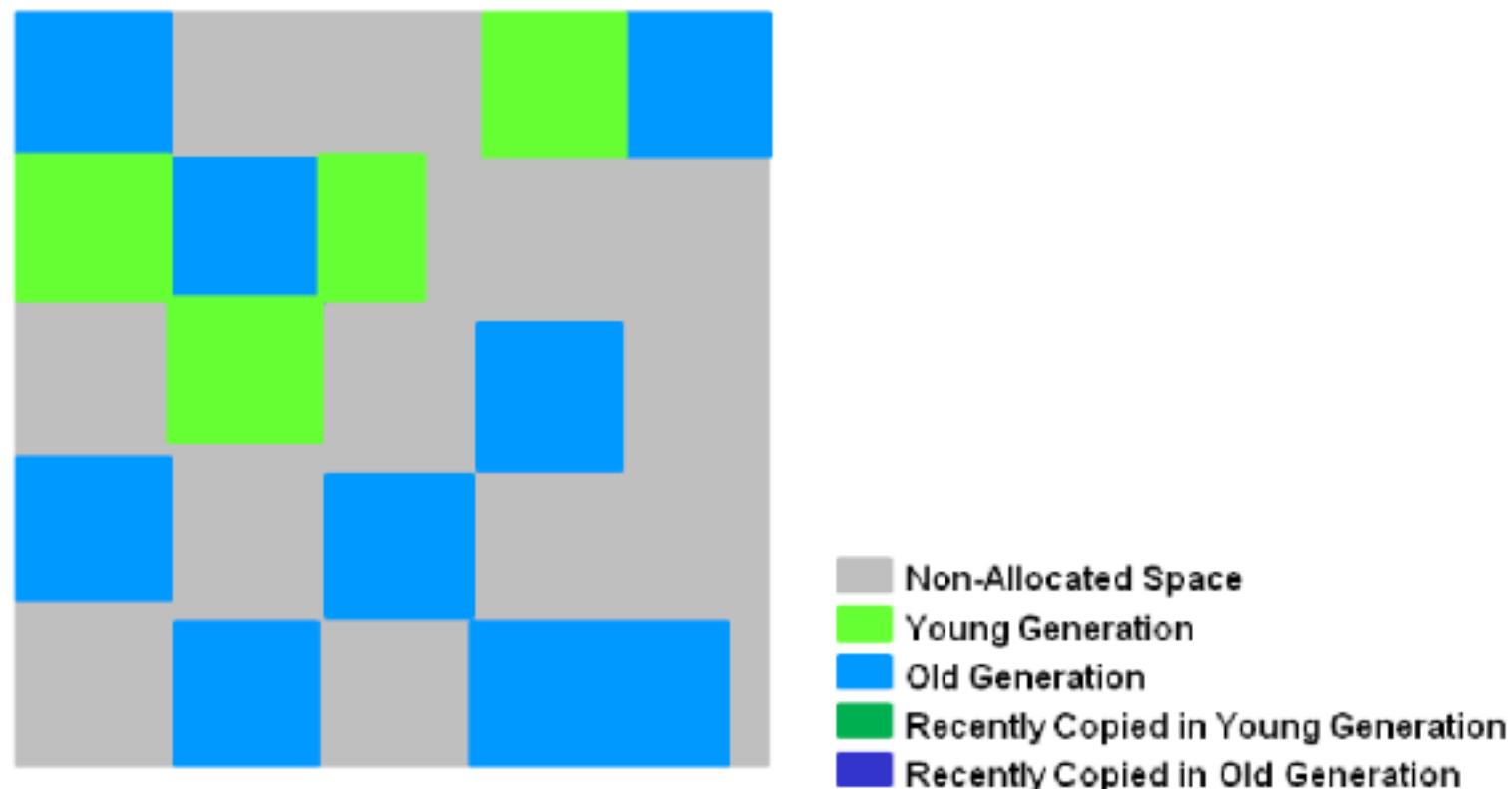
Old Generation

G1 step by step

- **As shown regions can be allocated into Eden, survivor, and old generation regions.**
 - In addition, there is a fourth type of object known as Humongous regions.
 - These regions are designed to hold objects that are 50% the size of a standard region or larger.
 - They are stored as a set of contiguous regions.
- Finally the last type of regions would be the unused areas of the heap.

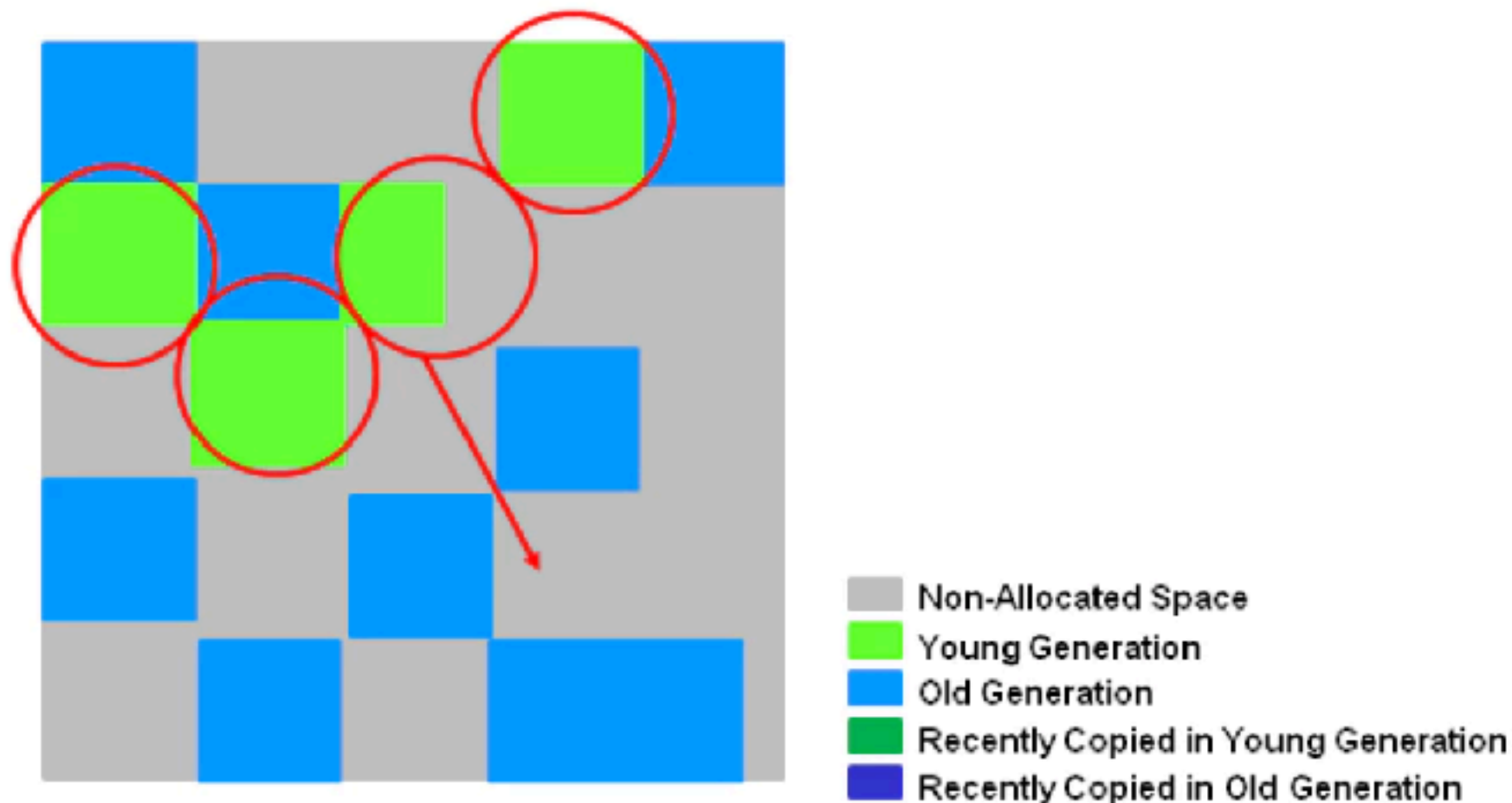
Young Generation in G1

- **The heap is split into approximately 2000 regions.**
 - Minimum size is 1Mb and maximum size is 32Mb.
 - Blue regions hold old generation objects and green regions hold young generation objects
 - Note that the regions are not required to be contiguous like the older garbage collectors.



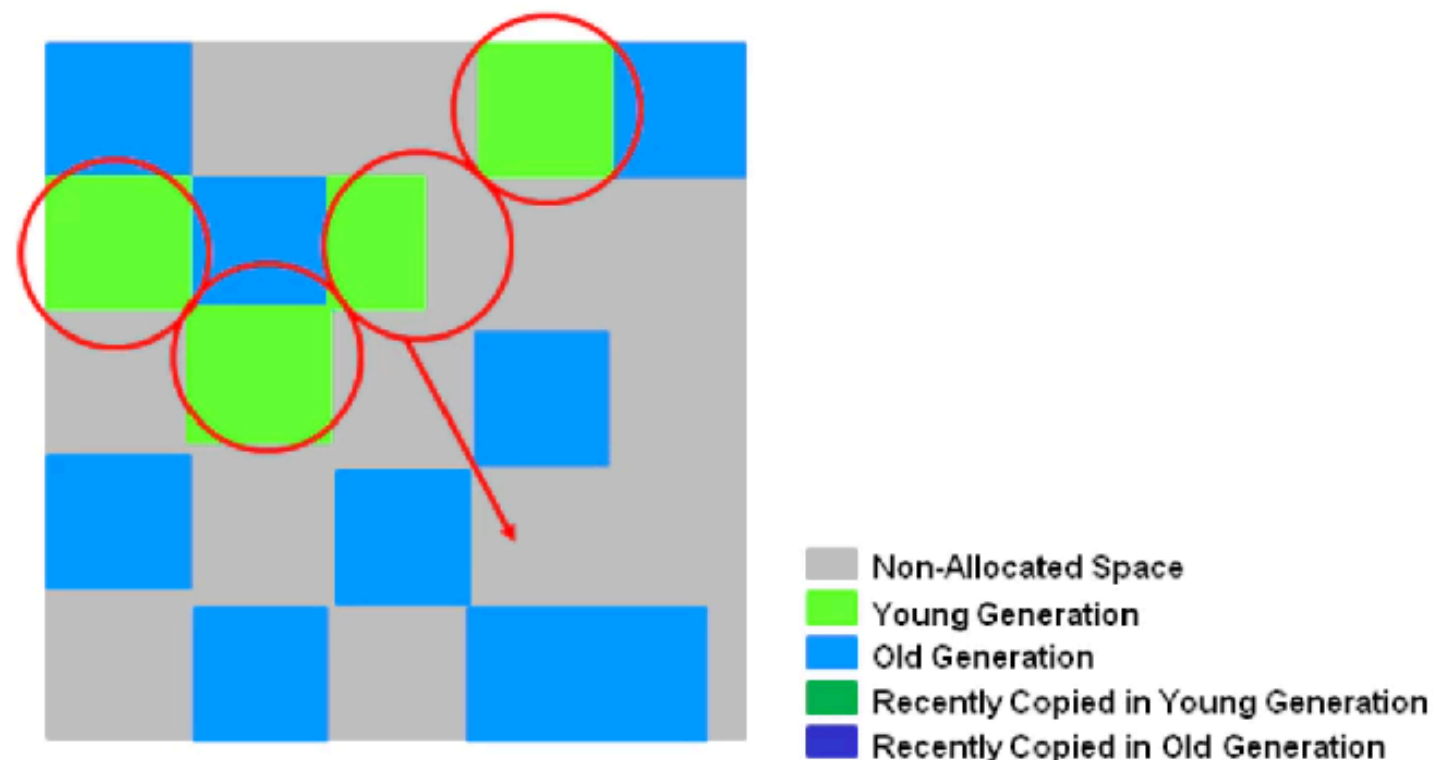
A Young GC in G1

- **Live objects are evacuated (i.e., copied or moved) to one or more survivor regions.**
- If the aging threshold is met, some of the objects are promoted to old generation regions.



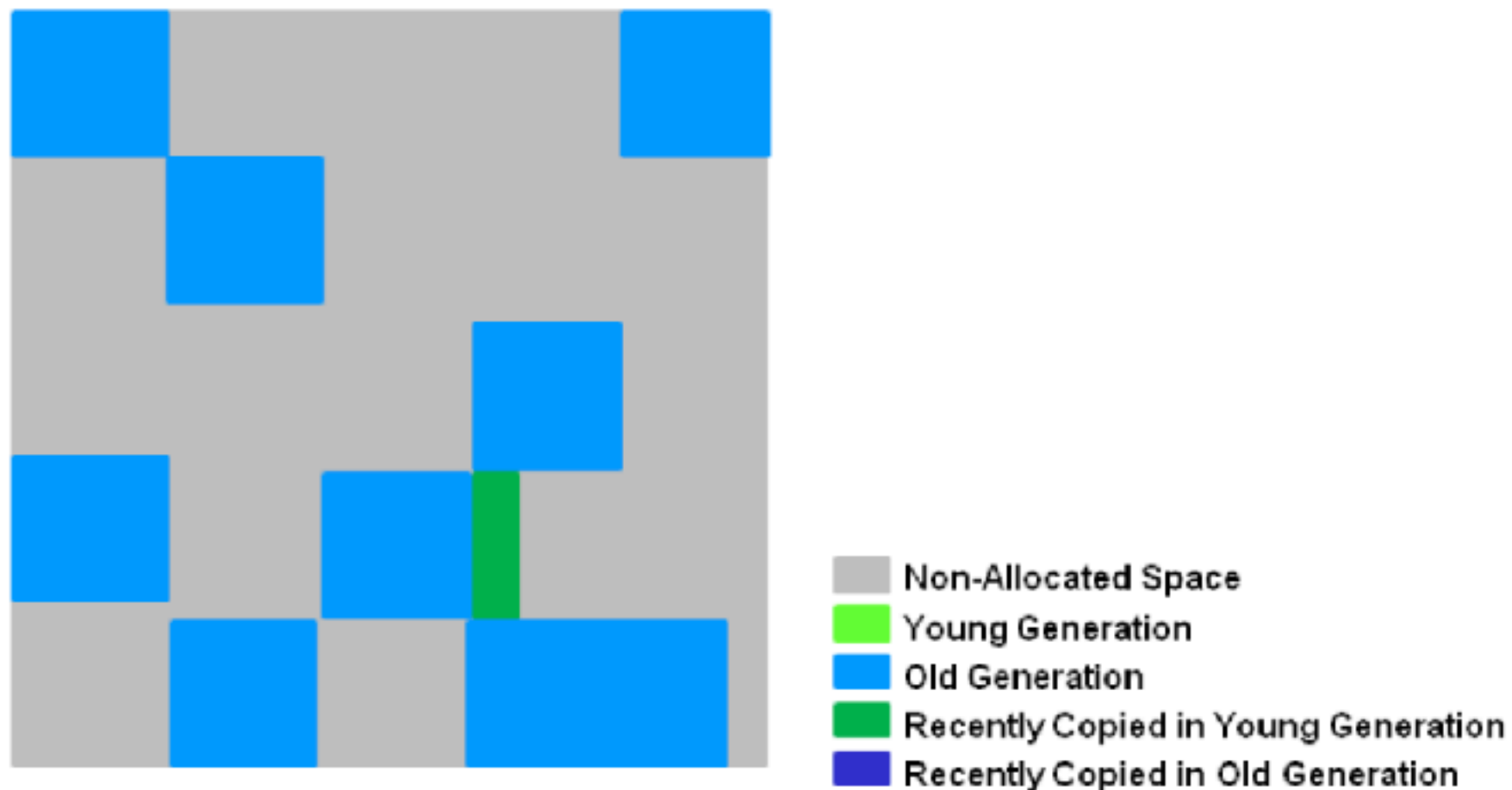
A Young GC in G1

- **This is a stop the world (STW) pause.**
 - Eden size and survivor size is calculated for the next young GC.
 - Accounting information is kept to help calculate the size.
 - Things like the pause time goal are taken into consideration.
 - This approach makes it very easy to resize regions, making them bigger or smaller as needed.



End of a Young GC with G1

- **Live objects have been evacuated to survivor regions or to old generation regions.**
 - Recently promoted objects are shown in dark blue.
 - Survivor regions in green.



G1 Young GC Summary

- **The heap is a single memory space split into regions.**
 - Young generation memory is composed of a set of non-contiguous regions.
 - This makes it easy to resize when needed.
 - Young generation garbage collections, or young GCs, are stop-the-world events.
 - All application threads are stopped for the operation.
 - The young GC is done in parallel using multiple threads.
 - Live objects are copied to new survivor or old generation regions.

Old Generation Collection with G1

Old Generation Collection with G1

- **Like the CMS collector, the G1 collector is designed to be a low pause collector for old generation objects.**
 - The G1 collector performs a bunch of phases on the old generation of the heap.
 - Note that some phases are part of a young generation collection.

G1 Collection Phases (1 of 3)

- **G1 Collection Phases - Concurrent Marking Cycle Phases**

1. Initial Mark (Stop the World Event)

- This is a stop the world event. With G1, it is piggybacked on a normal young GC. Mark survivor regions (root regions) which may have references to objects in old generation.

2. Root Region Scanning

- Scan survivor regions for references into the old generation. This happens while the application continues to run. The phase must be completed before a young GC can occur.

3. Concurrent Marking

- Find live objects over the entire heap. This happens while the application is running. This phase can be interrupted by young generation garbage collections.

G1 Collection Phases (2 of 3)

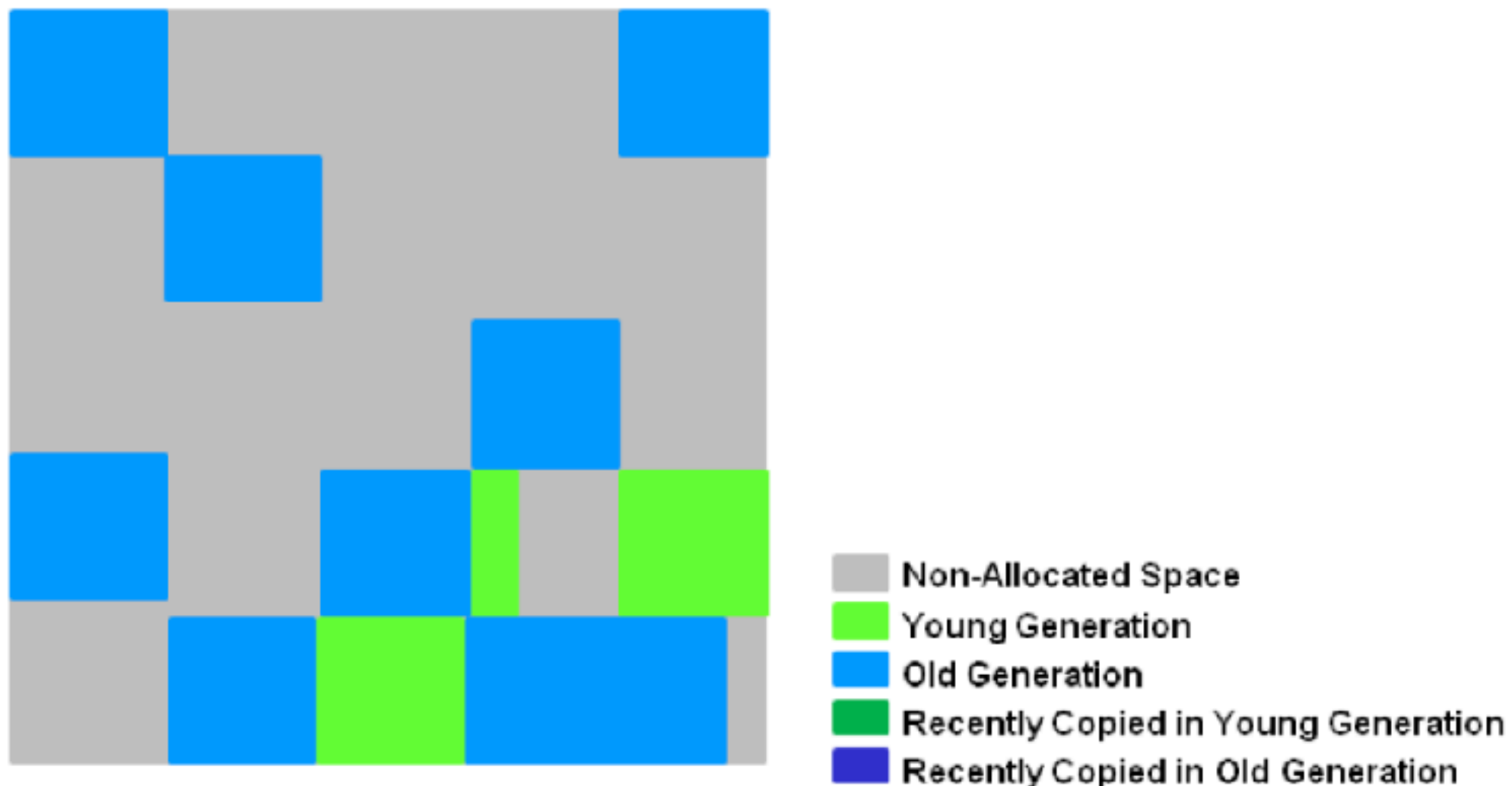
- **G1 Collection Phases - Concurrent Marking Cycle Phases**
 4. Remark (Stop the World Event)
 - Completes the marking of live object in the heap. Uses an algorithm called snapshot-at-the-beginning (SATB) which is much faster than what was used in the CMS collector.
 5. Cleanup (Stop the World Event and Concurrent)
 - Performs accounting on live objects and completely free regions. (Stop the world)
 - Updates the Remembered Sets. (Stop the world)
 - Reset the empty regions and return them to the free list. (Concurrent)

G1 Collection Phases (3 of 3)

- **G1 Collection Phases - Concurrent Marking Cycle Phases**
 6. Copying (Stop the World Event)
 - These are the stop the world pauses to evacuate or copy live objects to new unused regions.
 - This can be done with young generation regions which are logged as [GC pause (young)].
 - Or both young and old generation regions which are logged as [GC Pause (mixed)].

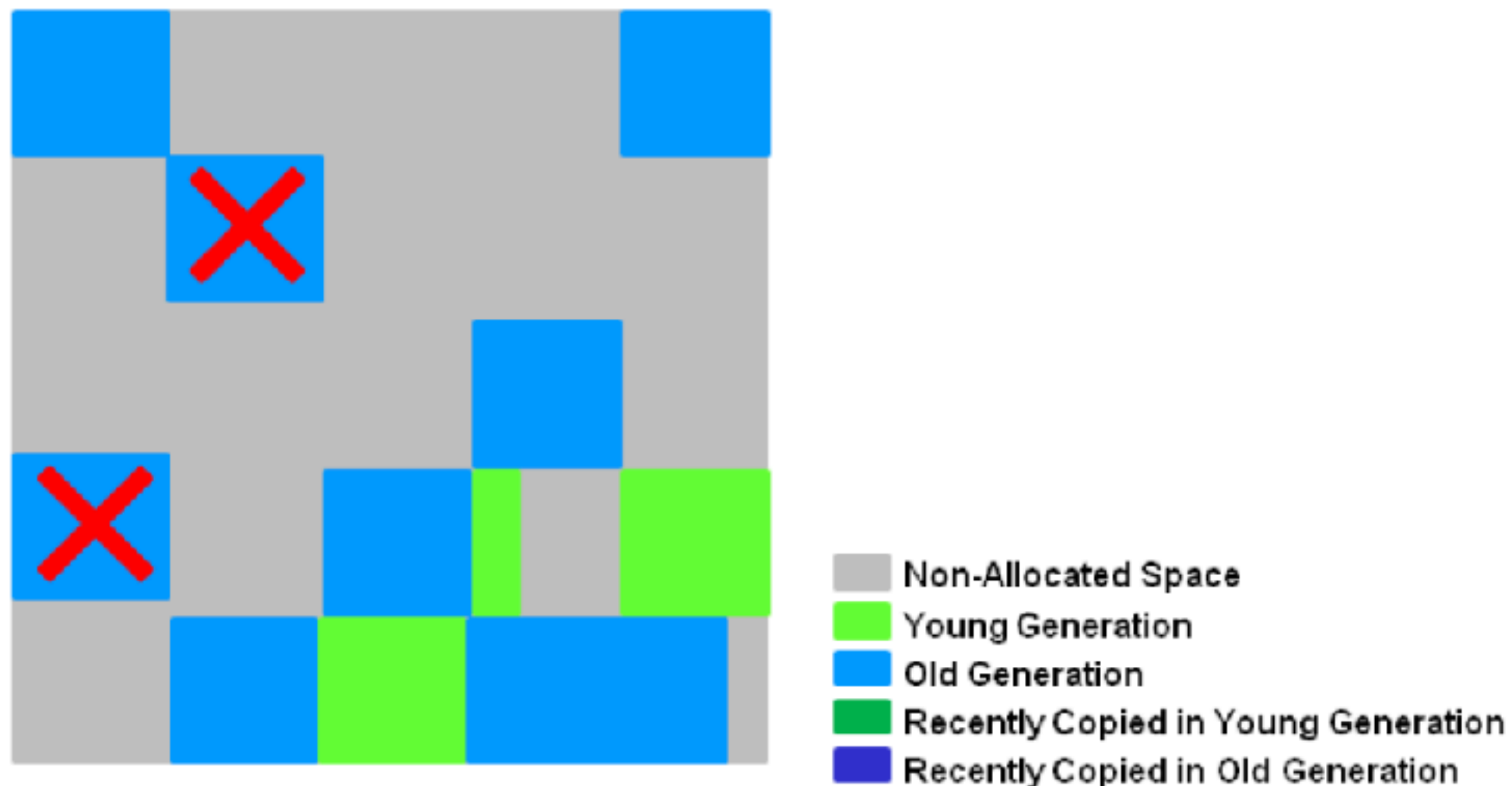
Initial Marking Phase

- Initial marking of live object is piggybacked on a young generation garbage collection.
- In the logs this is noted as GC pause (young)(inital-mark).



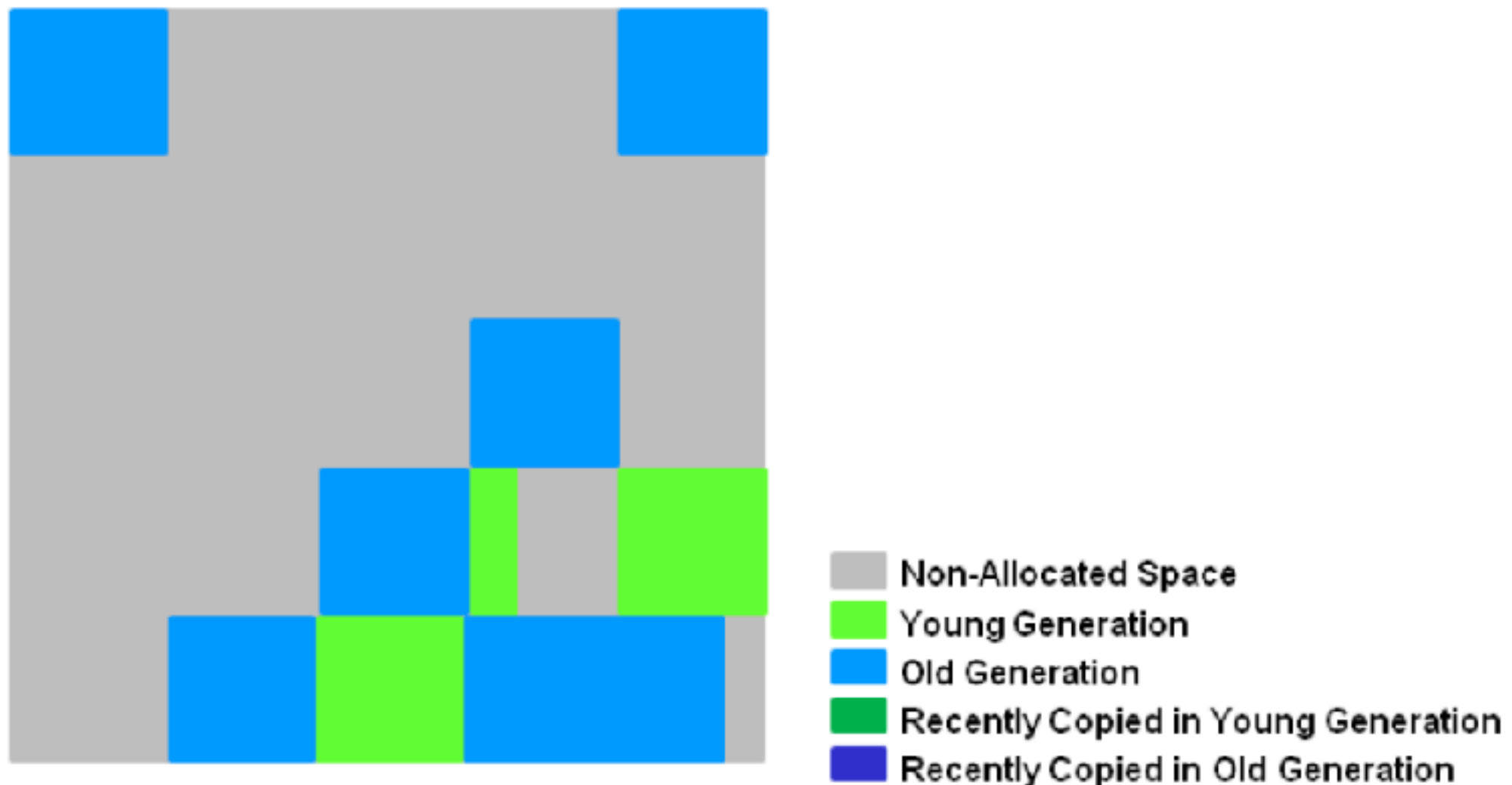
Concurrent Marking Phase

- If empty regions are found (as denoted by the "X"), they are removed immediately in the Remark phase.
- Also, "accounting" information that determines liveness is calculated.



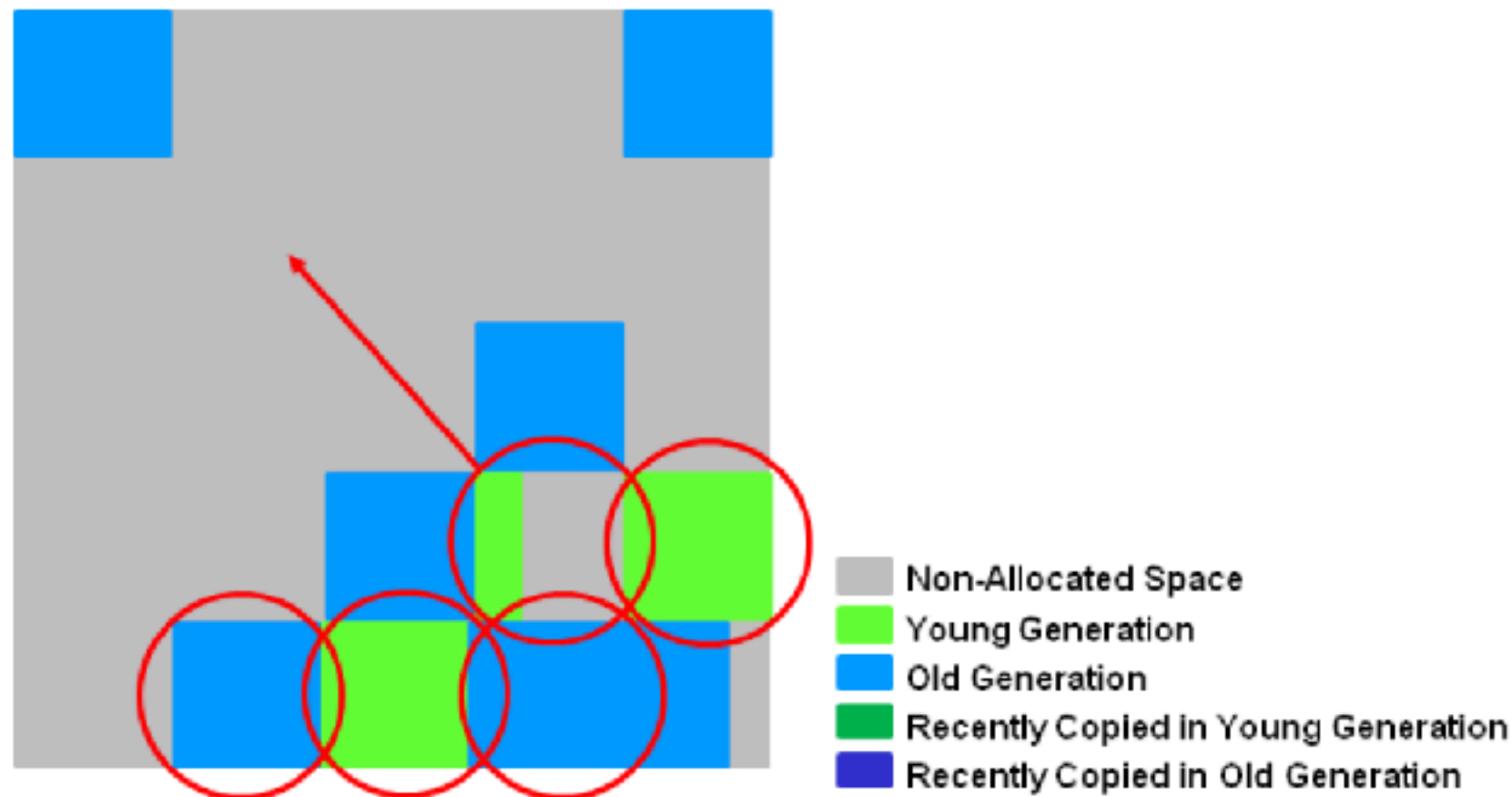
Re-mark Phase

- **Empty regions are removed and reclaimed.**
 - Region liveness is now calculated for all regions.



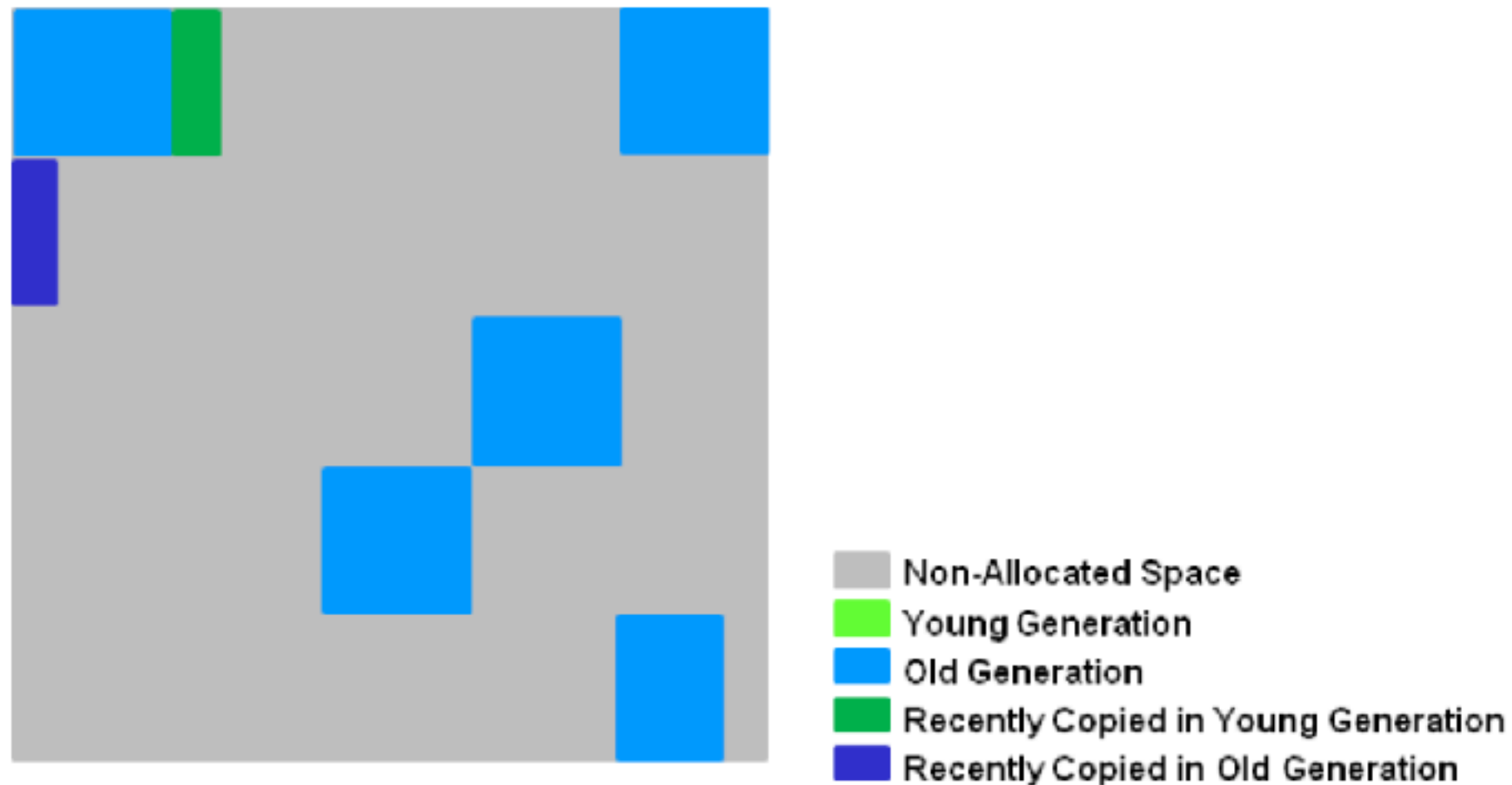
Copying/Cleanup Phase

- **G1 selects the regions with the lowest "liveness", those regions which can be collected the fastest.**
 - Then those regions are collected at the same time as a young GC.
 - This is denoted in the logs as [GC pause (mixed)].
 - So both young and old generations are collected at the same time.



After Copying/Cleanup Phase

- The regions selected have been collected and compacted into the dark blue region and the dark green region shown in the diagram.



Old Generation GC with G1 Summary

- **Concurrent Marking Phase**

- Liveness information is calculated concurrently while the application is running.
- This liveness information identifies which regions will be best to reclaim during an evacuation pause.
- There is no sweeping phase like in CMS.

- **Remark Phase**

- Uses the Snapshot-at-the-Beginning (SATB) algorithm which is much faster than what was used with CMS.
- Completely empty regions are reclaimed.

- **Copying/Cleanup Phase**

- Young generation and old generation are reclaimed at the same time.
- Old generation regions are selected based on their liveness.

G1 Command Line Options and Best Practices

Basic Command Line

- Is the default for Java 9
- In other Java versions, to enable the G1 Collector use:
-XX:+UseG1GC

- **Example:**

```
java -XX:+UseG1GC -XX:MaxGCPauseMillis=200 -jar MyJavaApp.jar
```

List of G1 GC Switches

- **-XX:+UseG1GC**
 - Use the Garbage First (G1) Collector
- **-XX:MaxGCPauseMillis=n**
 - Sets a target for the maximum GC pause time.
 - This is a soft goal, and the JVM will make its best effort to achieve it.
- **-XX:InitiatingHeapOccupancyPercent=n**
 - Percentage of the (entire) heap occupancy to start a concurrent GC cycle.
 - It is used by GCs that trigger a concurrent GC cycle based on the occupancy of the entire heap, not just one of the generations (e.g., G1).
 - A value of 0 denotes 'do constant GC cycles'.
 - The default value is 45.

List of G1 GC Switches

- **-XX:NewRatio=n**
 - Ratio of new/old generation sizes. The default value is 2.
- **-XX:SurvivorRatio=n**
 - Ratio of eden/survivor space size. The default value is 8.
- **-XX:MaxTenuringThreshold=n**
 - Maximum value for tenuring threshold. The default value is 15.
- **-XX:ParallelGCThreads=n**
 - Sets the number of threads used during parallel phases of the garbage collectors. The default value varies with the platform on which the JVM is running.

List of G1 GC Switches

- **-XX:ConcGCThreads=n**
 - Number of threads concurrent garbage collectors will use.
 - The default value varies with the platform on which the JVM is running.
- **-XX:G1ReservePercent=n**
 - Sets the amount of heap that is reserved as a false ceiling to reduce the possibility of promotion failure.
 - The default value is 10.
- **-XX:G1HeapRegionSize=n**
 - With G1 the Java heap is subdivided into uniformly sized regions.
 - This sets the size of the individual sub-divisions.
 - The default value of this parameter is determined ergonomically based upon heap size.
 - The minimum value is 1Mb and the maximum value is 32Mb.

Best Practices with G1 GC

Do not Set Young Generation Size

- **Explicitly setting young generation size via -Xmn meddles with the default behavior of the G1 collector.**
 - G1 will no longer respect the pause time target for collections. So in essence, setting the young generation size disables the pause time goal.
 - G1 is no longer able to expand and contract the young generation space as needed. Since the size is fixed, no changes can be made to the size.

Response Time Metrics

- **Instead of using average response time (ART) as a metric to set the `XX:MaxGCPauseMillis=<N>`, consider setting value that will meet the goal 90% of the time or more.**
 - This means 90% of users making a request will not experience a response time higher than the goal.
 - Remember, the pause time is a goal and is not guaranteed to always be met.

What is an Evacuation Failure?

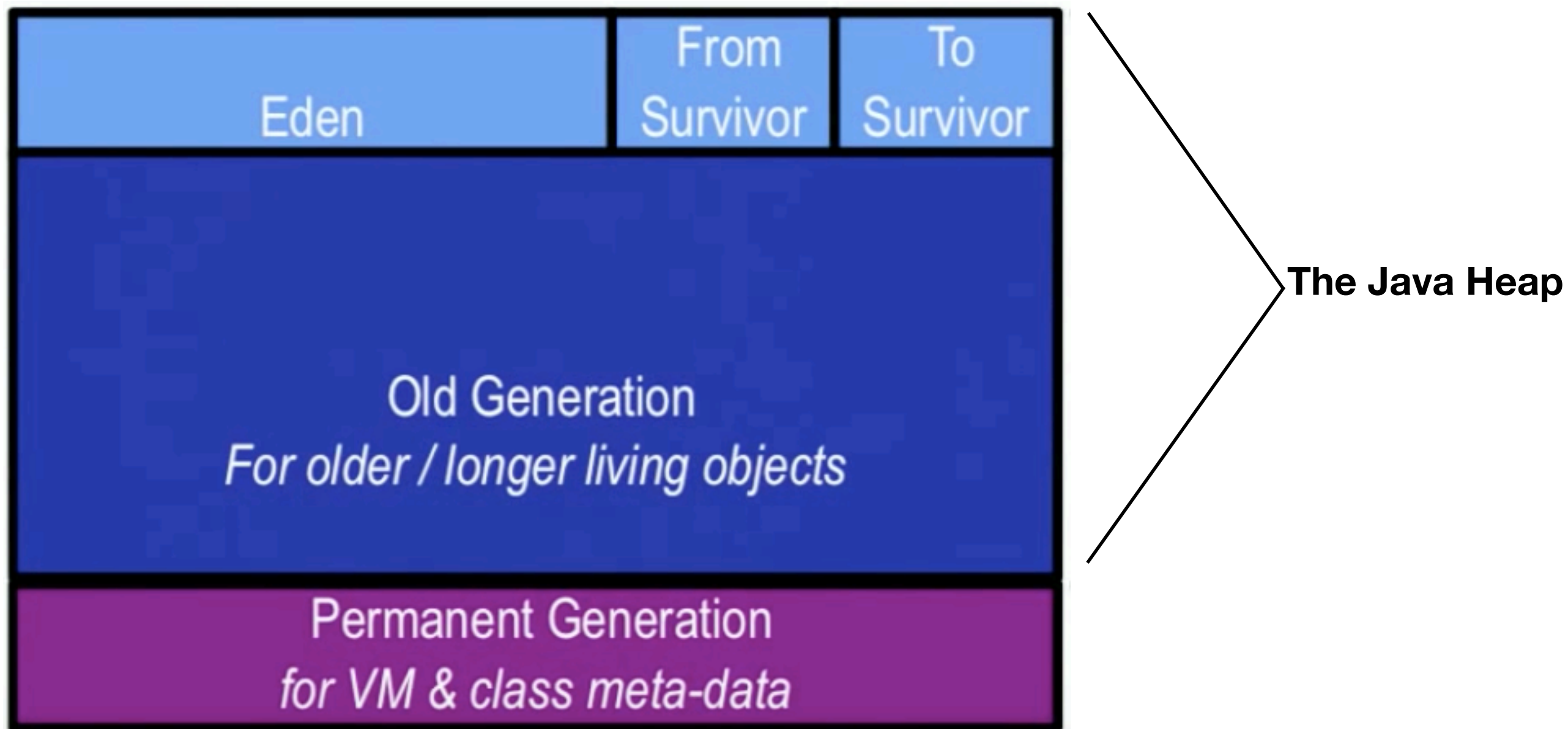
- **A promotion failure that happens when a JVM runs out of heap regions during the GC for either survivors and promoted objects.**
 - The heap can't expand because it is already at max.
 - This is indicated in the GC logs when using `-XX:+PrintGCDetails` by to-space overflow.
- **This is expensive!**
 - GC still has to continue so space has to be freed up.
 - Unsuccessfully copied objects have to be tenured in place.
 - Any updates to RSets of regions in the CSet have to be regenerated.
- **All of these steps are expensive.**

How to avoid Evacuation Failure

- **To avoid evacuation failure, consider the following options.**
 - Increase heap size
 - Increase the `-XX:G1ReservePercent=n`, the default is 10.
 - G1 creates a false ceiling by trying to leave the reserve memory free in case more 'to-space' is desired.
 - Start the marking cycle earlier
 - Increase the number of marking threads using the `-XX:ConcGCThreads=n` option.

**What you need to
know about GC**

Java HotSpot VM Heap layout



Important concepts (1 of 4)

- **Frequency of minor GC is dictated by**
 - Application object allocation rate
 - Size of the eden space
- **Frequency of object promotion into old generation is dictated by**
 - Frequency of minor GCs (how quickly objects age)
 - Size of the survivor spaces (large enough to age effectively)
 - Ideally promote as little as possible (more on this a bit later)

Important concepts (2 of 4)

- **Object retention can degrade performance more than object allocation**
 - In other words, the longer an object lives, the greater the impact on throughput, latency and footprint
 - Objects retained for a longer period of time
 - Occupy available space in survivor spaces
 - May get promoted to old generation sooner than desired
 - May cause other retained objects to get promoted earlier
 - GC only visits live objects
 - GC duration is a function of the number of live objects and object graph complexity

Important concepts (3 of 4)

- **Object allocation is very cheap**
 - 10 CPU instruction in common case
- **Reclamation of new object is also very cheap!**
 - Remember, only live objects are visited in a GC
- **Don't be afraid to allocate short lived objects**
 - ...especially for immediate results
- **GCs love small immutable objects and short-lived objects**
 - ...especially those that seldom survive a minor GC

Important concepts (4 of 4)

- **But, don't go overboard**
 - Don't do "needless" allocations
 - ... more frequent allocations means more frequent GCs
 - ... more frequent GCs imply faster object aging
 - ... faster promotions
 - ... more frequent needs for possibly either; concurrent old generation collection, or old generation compaction (i.e, full GC) ... or some kind of disruptive GC activity
- **It is better to use short-lived immutable objects than long-lived mutable objects**

Ideal situation

- **After application initialisation phase, only experience minor GCs and old generation growth is negligible**
 - Ideally, never experience need for old generation collection
 - Minor GCs are (generally) the fastest GC

Advice on choosing a GC

- **Start with Parallel GC (-XX:+UseParallelOldGC)**
 - Parallel GC offers the fastest minor GC times
 - If you can avoid full GCs, you'll likely achieve the best throughput and lowest latency
 - Move to CMS or G1 if needed (for old gen collections)
 - CMS minor GC times are slower due to promotion into “free lists”
 - CMS full GC avoided via old generation concurrent collection
 - G1 minor GC times are slower due to “Remembered Set” overhead
 - G1 full GC avoided via concurrent collection and fragmentation avoided by “Partial” old generation collection

GC Friendly Programming (1 of 4)

- **Large objects**
 - Expensive (in terms of time & CPU instructions) to allocate
 - Expensive to initialise (remember Java Spec ... object zeroing)
- **Large objects of different sizes can cause Java heap fragmentations**
 - A challenge for CMS, not so much so with ParallelGC or G1
- **Advice,**
 - Avoid large object allocations if you can
 - Especially frequent large object allocations during application “steady state”

GC Friendly Programming (2 of 4)

- **Data Structure Resizing**

- Avoid re-sizing of array backed collections / containers
- Use the constructor with an explicit size for the backing array

- **Re-sizing leads to unnecessary object allocation**

- Also contributes to Java heap fragmentation

- **Object pooling potential issues**

- Contributes to number of live objects visited during a GC
 - Remember GC duration is a function of live objects
- Access to the pool requires some kind of locking
 - Frequent pool access may become a scalability issue

GC Friendly Programming (3 of 4)

- **Finalizers**

- Please don't do it
- Requires at least 2 GCs cycle and GC cycles are slower
- If possible, add a method to explicitly free resources when done with an object
 - Can't explicitly free resources
 - Use Reference Objects as an alternative
 - See JDK's `DirectByteBuffer.java` implementation for an example use

GC Friendly Programming (4 of 4)

- **SoftReferences**

- Please don't do it!

- **WeakReferences are cleared by GC**

- JVM GC's implementation determines how aggressive they are cleared
 - In other words, the JVM GC's implementation really dictates the degree of object retention
 - Remember the relationship to object retention
 - Higher object retention, longer GC pause times
 - Higher object retention, more frequent GC pauses

Subtle object retention (1 of 2)

```
class ClassWithFinalizer {  
    protected void finalize() { // do some cleanup }  
}
```

```
class MyClass extends ClassWithFinalizer {  
    private byte[] buffer = new byte[1024*1024*2];  
    ...  
}
```

- **Object retention consequences of MyClass?**
 - **At least 2 GC cycles to free the byte[] buffer**
- **How to lower the object retention?**

```
class MyClass {  
    private ClassWithFinalizer classWithFinalizer;  
    private byte[] buffer = new byte[1024*1024*2];  
    ...  
}
```

Subtle object retention (2 of 2)

- **What about inner classes?**
 - Remember that inner classes have an implicit reference to the outer instance
- **Potentially can increase object retention**
- **Again, increased object retention ... more live objects at GC time ... increased GC duration**

Fundamentals - Minor GCs

- **Minor GC Frequency - How often they occur**
 - Dictated by object allocation rate and size of eden space
 - Higher allocation rate or smaller eden => more frequent minor GC
 - Lower allocation rate or larger eden => less frequent minor GC
- **Minor GC Pause Time**
 - Dictated (mostly) by number of live objects
 - Some deviations of course, number of reference objects, object graph structure, number of promotions to old gen

Fundamentals - Full GC Frequency

- **Full GC Frequency - How often they occur**
 - For Parallel GC (and Serial GC)
 - Dictated by promotion rate and size of old generation space
 - Higher promotion rate or smaller old gen => more frequent full GC
 - Lower promotion rate or larger old gen => less frequent full GC
 - For CMS & G1 - a bit more complex!
 - Dictated by promotion rate, time to execute a concurrent cycle and when the concurrent cycle is initiated - potential for “losing the race”
 - Some differences between CMS & G1 concurrent cycles
 - Also for CMS, also dictated by frequency of old men fragmentation, a situation that requires old gen compaction via a full GC
 - G1 has shown to combat fragmentation very well

Fundamentals - Concurrent Cycle Frequency

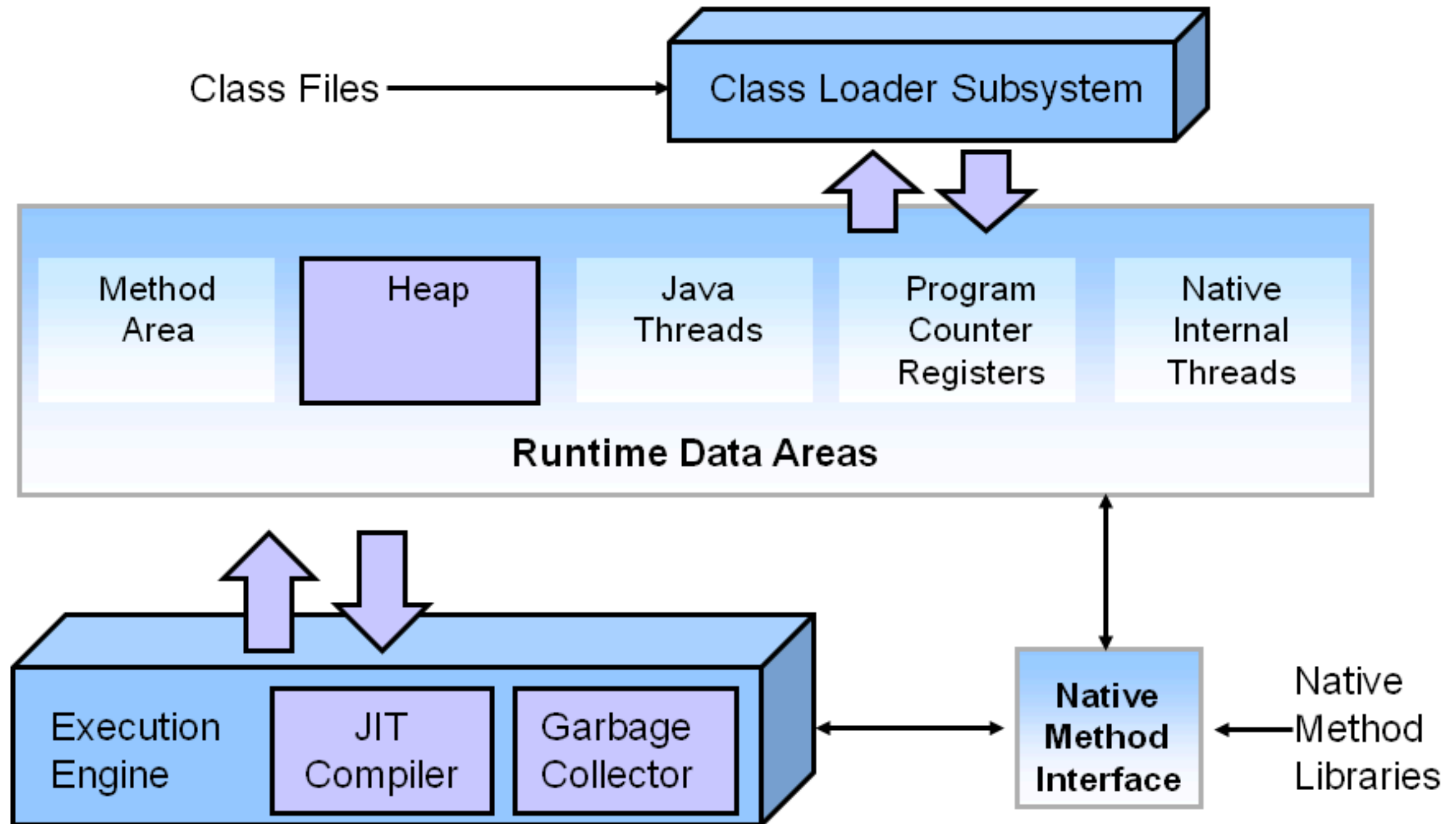
- **For CMS & G1, concurrent cycle frequency**
 - Dictated by the promotion rate, size of old gen and when concurrent cycle is initiated (a heap occupancy threshold)
 - CMS initiating threshold is a percent of old gen occupancy
 - G1 initiating threshold is a percent of entire Java heap, not just old gen occupancy
 - Remember concurrent cycles execute at the same time as your application taking CPU from your application

Fundamentals - Full GC Pause Time

- **For CMS or G1**
 - Almost always a very lengthy pause
 - Except a much longer pause than Parallel Old GC's full GC
 - Single threaded
 - CMS - in reaction to a promotion failure; “losing the race” (concurrent cycle did not finish in time) or fragmentation (old generation requires compaction)
 - G1 - in reaction to there not being enough space available to evaluate live objects to an available region “to-space-overflow”
- May have to “undo” reference updates due to promotion failure or to-space-overflow - a time consuming operation

**What you need to know
about JIT compilation**

Key HotSpot JVM Components



Important Concepts

- **Optimization decisions are made based on**
 - Classes that have been loaded and code paths executed
 - JIT compiler does not have full knowledge of entire program
 - Only knows what has been classloaded and code paths executed
 - Hence, optimization decisions makes assumptions about how a program has been executing - it knows nothing about what has not been classloaded or executed
 - Assumptions may turn out (later) to be wrong ... it must be able to “recover” which (may) limit the type(s) of optimization(s)
 - New classloading or code path ... possible de-opt/re-opt

Inlining and Virtualization, Completing Forces

- **Greatest optimization impact realised from “method inlining”**
- Virtualized methods are the biggest barrier to inlining
 - Good news ... JIT compiler can de-virtualize methods if it only sees 1 implementation of a virtualized method ... effectively makes it a mono-morphic call
 - Bad news ... if JIT compiler later discovers an additional implementation it must de-optimize, re-optimize for 2nd implementation ... no we have a bi-morphic call
 - This type of de-opt & re-opt will likely lead to lesser peak performance, especially true when / if you get to the 3rd implementation because now its a mega-morphia cal

Inlining and Virtualization, Completing Forces

- **Important points**

- Discovery of additional implementations of virtualized methods will slow down your application
- A mega-morphia call can limit or inhibit inlining capabilities

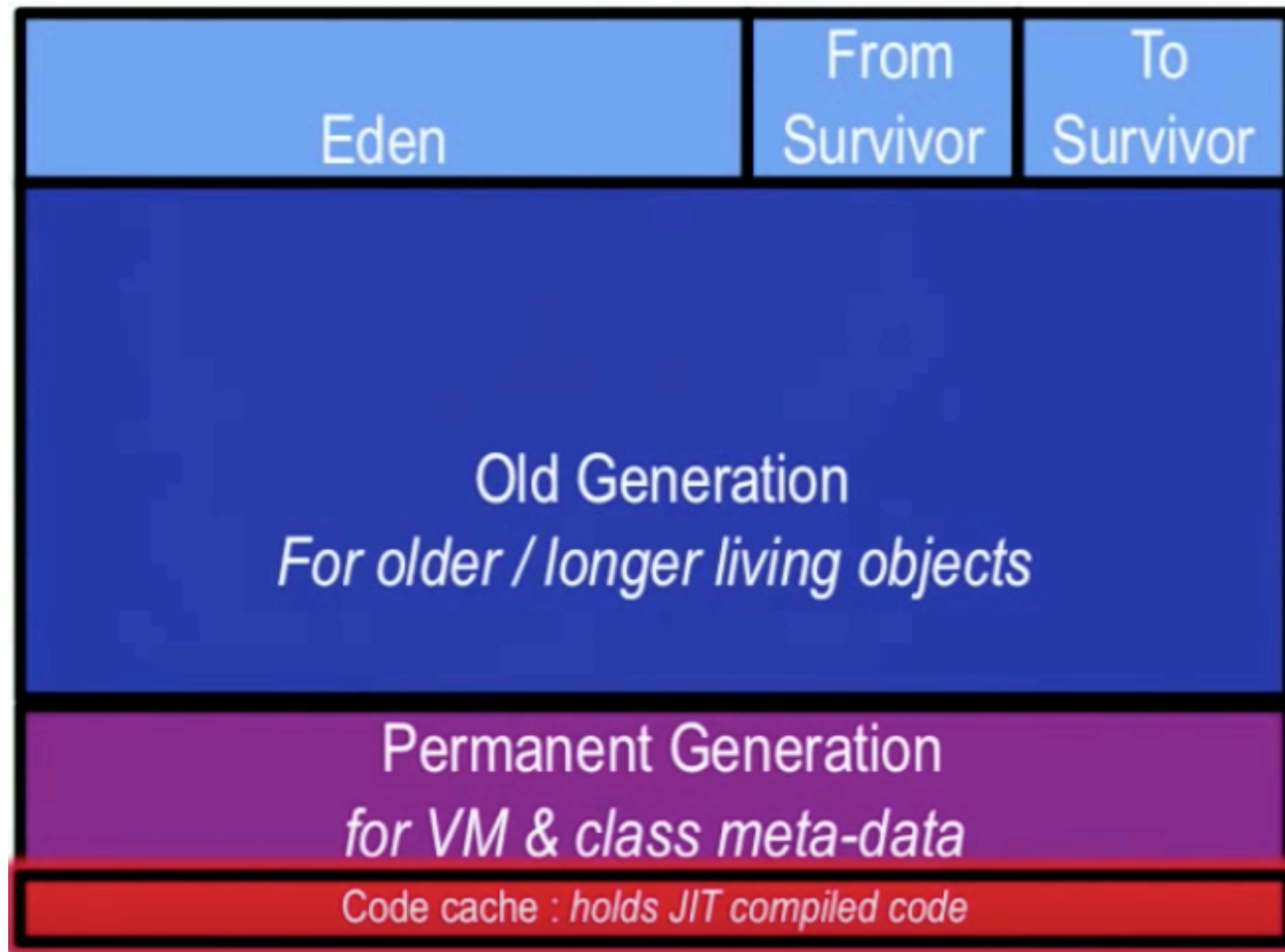
- **How about writing “JIT compiler friendly code”?**

- That’s a premature optimization!

- **Advice?**

- Write code in its most natural form, let the JIT compiler agonize over how to best optimize it
- Use tools to identify the problem areas and make code changes as necessary

Code Cache, the “hidden space”



Code cache

- **Default size is 48 megabytes for HotSpot Server JVM**
 - Increased to 96 megabytes for Java 8
 - 32 megabytes in HotSpot Client JVMs
- **Prior to Java 7, if you run out of code cache space**
 - JVM prints a warning message:
 - “CodeCache is full. Compiler has been disabled”
 - “Try increasing the code cache size using -
XX:ReservedCodeCacheSize=..”
 - Common symptom ... application mysteriously slows down after its been running for a lengthy period of time
 - Generally, more likely to see on (large) enterprise class apps

Code cache

- **How to monitor code cache space**
 - Can't periodically look at code cache space occupancy with monitoring tools such as Console
 - JIT compiler will continue to mark code that's no longer valid, but will not re-initiate new compilations, i.e, -XX:+PrintCompilation shows “made not entrant” and “made zombie”, but not new activations
 - So, code cache could look like it has available space via Console when in reality it is exhausted - can be very misleading!

Code cache

- **Advice**

- Profile app with profiler that also profiles the internals of the JVM
 - Look for high JVM interpreter CPU time
- Check log files for log message saying code cache is full
- Use `-XX:+UseCodeCacheFlushing` (Java 6u* releases & later)
 - Will evict least recently used code from code cache
 - Possible for compiler thread to cycle (optimize, throw away, optimize, throw away), but that's better than disabled compilation
- Best option, increase `-XX:ReservedCodeCacheSize`, or do both `+UseCodeCacheFlushing` & increase `ReservedCodeCacheSize`

Code cache

- **Java 7 +**
 - `-XX:+UseCodeCacheFlushing` is on by default
 - But, “flushing” may be an intrusive operation for the JIT compiler if there is a lot of additional demands made on it, i.e. new activations, code invalidations
 - May need to tune `-XX:CodeCacheMinimumFreeSpace` and `-XX:MinCodeCacheFlushingInterval`
- Advice
 - Profile with a profiler that also profiles JVM intervals and look for high amounts of CPU used in code cache flushing
 - Best option, increase `-XX:ReservedCodeCacheSize`, tune code cache flushing as a secondary activity

Tools to help you

GC Analysis Tools

- **Offline mode, after the fact**
 - GCHisto or GCViewer (search for “GCHisto” or “chewiebug GCViewer”) - both are GC log visualizers
 - Recommend -XX:+PrintGCDetails, -XX:+PrintGCTimeStamps or -XX:+PrintGCDateStamps JVM command line options
- **Online mode, while application is running**
 - VisualGC plugin for VisualVM (found in JDK’s bin directory, launched as “jvisualvm” - then install VisualGC plugin)
- **VisualVM or Eclipse MAT for unnecessary object allocation and object retention**

JIT Compilation Analysis Tools

- **Command line tools**

- `-XX:+PrintOptoAssembly`

- Requires “fastdebug JVM”, can be built from OpenJDK sources
 - Offers the ability to see generated assembly code with Java code
 - Lots of output to digest

- `-XX:+LogCompilation`

- Must add `-XX:+UnlockDiagnosticVMOptions`, but “fastdebug JVM” not required
 - Produces XML file that shows the path of JIT compiler optimisations
 - Non-trivial to read and understand
 - Search for “HotSpot JVM LogCompilation” for more details

JIT Compilation Analysis Tools

- **GUI Tools**

- Oracle Solaris Studio Performance Analyzer
 - Works with both Solaris and Linux (x86/x64 & SPARC)
 - Better experience on Solaris (more mature, ported to Linux a couple of years ago, and no CPU micro state info on Linux)
 - See generated JIT compiler code embedded with Java source
 - Free download (search for “Studio Performance Analyzer”)
 - Excellent method profiler, lock profiler and hardware counter profiler (i.e. CPU cache misses, TLB misses, instructions retired, etc)
- Similar tools
 - Intel VTune
 - AMD CodeAnalyst

Micro Benchmarking Java applications

What is a Micro Benchmark?

- **Benchmark is the process of recording the performance of a system.**
 - (Macro) benchmarks are done between different platforms to compare the efficiency between them.
 - Micro benchmarks are done within the same platform for a small snippet of code.

Micro Benchmark

- **Micro benchmarks are generally done for two reasons.**
 - To compare different approaches of code which implements the same logic and choose the best one to use.
 - To identify any bottlenecks in a suspected area of code during performance optimization.

How to do a Micro Benchmark?

- **Do not worry about the performance requirements.**
- Start with building functionality and concentrate on making things work and don't focus on how.
 - Once the software goes to production, we will be facing the inevitable.
- Have the performance objectives written down before writing the code.
- Weigh the importance of performance with respect to the functional requirements and come up with a balance between them.

Stages of a micro benchmark

- **Four important stages of a micro benchmark**
 - Benchmark design
 - Benchmark code implementation
 - Benchmark test execution
 - Results interpretation

Key Points to Remember

- **Always include a warmup phase which runs your test kernel all the way through, enough to trigger all initializations and compilations before timing phase(s).**
 - Fewer iterations is OK on the warmup phase.
 - The rule of thumb is several tens of thousands of inner loop iterations.
- **Always run with -XX:+PrintCompilation, -verbose:gc, etc., so you can verify that the compiler and other parts of the JVM are not doing unexpected work during your timing phase.**

Key Points to Remember

- **Print messages at the beginning and end of timing and warmup phases, so you can verify that there is no output from Rule 2 during the timing phase.**
- **Use a library for your benchmark as it is probably more efficient and was already debugged for this sole purpose.**
 - Such as JMH, Caliper

Java Micro Benchmark Tools

- **Writing your own micro or nano benchmark implementation code will face several problems:**
 - How to do a warm-up of the environment?
 - Managing the OS power balance process,
 - Equal priority for threads,
 - Repeating the task cycle,
 - Recording observations and reporting stats, etc;
 - All these steps should be done in an accurate and repeatable way.
 - Our stopwatch benchmarking methodology is surely not sufficient for this.

Java Micro Benchmark Tools

- **Most popular tools available for Micro Benchmarking Java applications:**
 - Caliper by Google
 - JMH by OpneJDK

JMH - Java Microbenchmark Harness

- **JMH is a Java harness for building, running, and analysing nano/micro/milli/macro benchmarks written in Java and other languages targeting the JVM.**
- **Annotation based**
- **OpenJDK**
- **The recommended way to run a JMH benchmark is to use Maven to setup a standalone project that depends on the jar files of your application.**
 - This approach is preferred to ensure that the benchmarks are correctly initialized and produce reliable results.
 - It is possible to run benchmarks from within an existing project, and even from within an IDE, however setup is more complex and the results are less reliable.

Most common Annotations

- **@Benchmark**
 - is the annotation that tells the JMH to benchmark the method.
- **@BenchmarkMode**
 - tells JMH what you want to measure.
- **@OutputTimeUnit**
 - enables you to specify what time units you want the benchmark results printed in.
- **@State**
 - Sometimes you may want to initialize some variables that your benchmark code needs, but which you do not want to be part of the code your benchmark measures. Such variables are called "state" variables.
 - State variables are declared in special state classes, and an instance of that state class can then be provided as parameter to the benchmark method.

Benchmark modes

- **org.openjdk.jmh.annotations.Mode**
 - Throughput
 - Measures the number of operations per second, meaning the number of times per second your benchmark method could be executed.
 - AverageTime
 - Measures the average time it takes for the benchmark method to execute (a single execution).
 - SampleTime
 - Measures how long time it takes for the benchmark method to execute, including max, min time etc.
 - SingleShotTime
 - Measures how long time a single benchmark method execution takes to run.
 - This is good to test how it performs under a cold start (no JVM warm up).
 - All

Benchmark Time Units

- **java.util.concurrent.TimeUnit**
 - NANoseconds
 - MICROseconds
 - MILLIseconds
 - SECONDS
 - MINUTES
 - HOURS
 - DAYS

State Scope

- **A state object can be reused across multiple calls to your benchmark method.**
 - JMH provides different "scopes" that the state object can be reused in.
 - The state scope is specified in the parameter of the @State annotation.
 - Scope.Thread
 - Each thread running the benchmark will create its own instance of the state object.
 - Scope.Group
 - Each thread group running the benchmark will create its own instance of the state object.
 - Scope.Benchmark
 - All threads running the benchmark share the same state object.

Getting Started With JMH

- The easiest way to get started with JMH is to generate a new JMH project using the JMH Maven archetype.
- The JMH Maven archetype will generate a new Java project with a single, example benchmark Java class, and a Maven pom.xml file.
- The Maven pom.xml file contains the correct dependencies to compile and build your JMH microbenchmark suite.

```
mvn archetype:generate
    -DinteractiveMode=false
    -DarchetypeGroupId=org.openjdk.jmh
    -DarchetypeArtifactId=jmh-java-benchmark-archetype
    -DgroupId=co.vinod
    -DartifactId=my-first-benchmark
    -Dversion=1.0
```

First JMH Benchmark code

```
package co.vinod;

import org.openjdk.jmh.annotations.Benchmark;

public class MyBenchmark {

    @Benchmark
    public void testMethod() {

        int a = 1;
        int b = 2;
        int sum = a + b;
    }

}
```

Building your JMH benchmark

```
mvn clean install
```

When you build your JMH benchmarks, Maven will always generate a JAR file named benchmarks.jar in the target directory

Running Your JMH Benchmarks

```
java -jar target/benchmarks.jar
```

This will start JMH on your benchmark classes. JMH will scan through your code and find all benchmarks and run them. JMH will print out the results to the command line.

**Coding practices to
follow for better
performance**

Coding practices to follow for better performance

- **Avoid object creation**
- **Knowing String better**
- **Exception and Casting**
- **The Rhythm of motion**
- **Use appropriate collection**

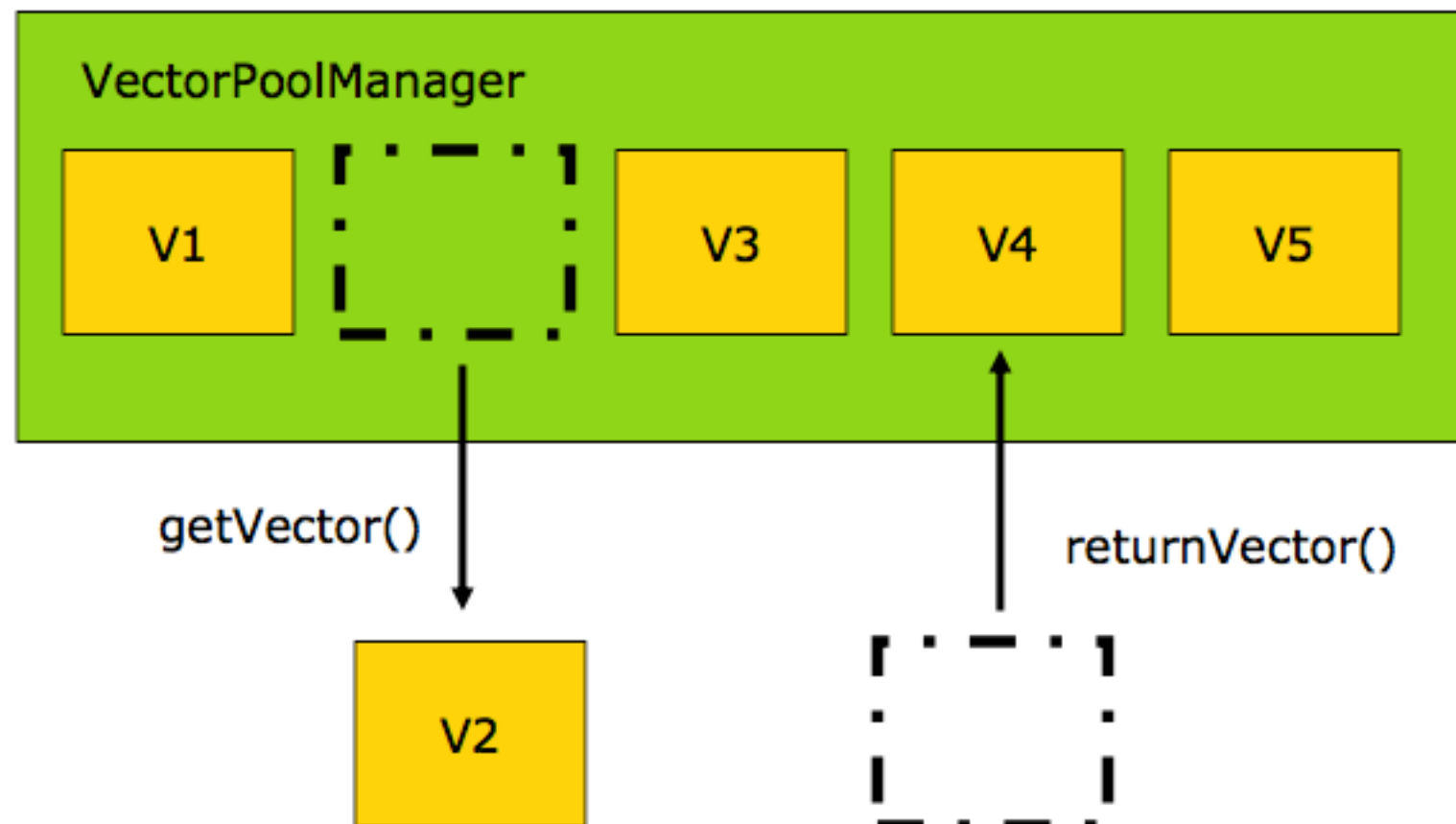
Avoid object creation

Avoid object creation

- **Object creation causes problem**
 - Lots of objects in memory means GC does lots of work
 - When GC starts, programs slow down
 - Creating object costs time and CPU effort for application
 - Reuse objects where possible

Pool Management

- Most container (e.g. Vector) objects could be reused rather than created and thrown away



Pool Management

```
public void doSome()
{
    for (int i=0; i < 10; i++) {
        Vector v = new Vector()
        ... do vector manipulation stuff
    }
}
```

```
public static VectorPoolManager vpm = new VectorPoolManager(25)
public void doSome()
{
    for (int i=0; i < 10; i++) {
        Vector v = vectorPoolManager.getVector( );
        ... do vector manipulation stuff
        vectorPoolManager.returnVector(v);
    }
}
```

Canonicalizing Objects

- Replace multiple object by a single object or just a few

```
public class VectorPoolManager
{
    private static final VectorPoolManager poolManager;
    private Vector[] pool;

    private VectorPoolManager(int size)
    {
        ....
    }
    public static Vector getVector()
    {
        if (poolManager == null)
            poolManager = new VectorPoolManager(20);

        ...
        return pool[pool.length-1];
    }
}
```

Singleton Pattern

Canonicalizing Objects

```
Boolean b1 = new Boolean(true);  
Boolean b2 = new Boolean(false);  
Boolean b3 = new Boolean(false);  
Boolean b4 = new Boolean(false);
```

4 objects in memory

```
Boolean b1 = Boolean.TRUE  
Boolean b2 = Boolean.FALSE  
Boolean b3 = Boolean.FALSE  
Boolean b4 = Boolean.FALSE
```

2 objects in memory

Canonicalizing Objects

```
String string = "55";  
Integer theInt = new Integer(string);
```

No Cache

```
String string = "55";  
Integer theInt = Integer.valueOf(string);
```

Object Cached

Canonicalizing Objects

```
private static class IntegerCache {
    private IntegerCache(){}

    static final Integer cache[] = new Integer[-(-128) + 127 + 1];
    static {
        for(int i = 0; i < cache.length; i++)
            cache[i] = new Integer(i - 128);
    }
}

public static Integer valueOf(int i) {
    final int offset = 128;
    if (i >= -128 && i <= 127) { // must cache
        return IntegerCache.cache[i + offset];
    }
    return new Integer(i);
}
```

Caching inside Integer.valueOf(...)

Keyword, 'final'

- Use the final modifier on variable to create immutable internally accessible object

```
public void doSome(Dimension width, Dimension height)
{
    //Re-assign allow
    width = new Dimension(5,5);
    ...
}
```

```
public void doSome(final Dimension width, final Dimension height)
{
    //Re-assign disallow
    width = new Dimension(5,5);
    ...
}
```

Auto-Boxing/Unboxing

- Object-Creation made every time we wrap primitive by boxing
- Use Auto-Boxing as need not as always

```
Integer i = 0;  
//Counting by 10M  
while (i < 1000000000)  
{  
    i++;  
}
```

Takes 2313 ms

Why it takes
 $2313/125 \approx 20$ times longer?

```
int p = 0;  
//Counting by 10M  
while (p < 1000000000)  
{  
    p++;  
}
```

Takes 125 ms

Knowing String better

Knowing String better

- **String is the Object mostly used in the application**
- **Overlook the String, the software may have the poor performance**

Compile-Time String Initialization

- Use the string concatenation (+) operator to create Strings at compile-time.

```
for (int i =0; i < loop; i++)  
{  
    //Looping 10M rounds  
    String x = "Hello" + "," + " "+ "World";  
}
```

Takes 16 ms

```
for (int i =0; i < loop; i++)  
{  
    //Looping 10M rounds  
    String x = new String("Hello" + "," + " "+ "World");  
}
```

Takes 672 ms

Runtime String Initialization

- Use **StringBuffers/StringBuilder** to create Strings at runtime.

```
String name = "Smith";
for (int i =0; i < loop; i++)
{
    //Looping 1M rounds
    String x = "Hello";
    x += ",";
    x += " Mr.";
    x += name;
}
```

Takes 10298 ms

```
String name = "Smith";
for (int i =0; i < loop; i++)
{
    //Looping 1M rounds
    String x = (new StringBuffer()).append("Hello")
        .append(",").append(" ")
        .append(name).toString();
}
```

Takes 6187 ms

String comparison

- Use appropriate method to compare the String

```
for (int i =0; i < loop; i++)  
{  
    //10m loops  
    if (a != null && a.equals(""))  
    {  
    }  
}
```

Takes 125 ms

```
for (int i =0; i < loop; i++)  
{  
    //10m loops  
    if (a != null && a.length() == 0)  
    {  
    }  
}
```

Takes 31 ms

String comparison

- Use appropriate method to compare the String

```
String a = "abc"  
String b = "cdf"  
for (int i = 0; i < loop; i++)  
{  
    if (a.equalsIgnoreCase(b))  
    {  
    }  
}
```

Takes 750 ms

```
String a = "abc"  
String b = "cdf"  
for (int i = 0; i < loop; i++)  
{  
    if (a.equals(b))  
    {  
    }  
}
```

Takes 125 ms

String comparison

- **String.equalsIgnoreCase() does only 2 steps**
- **It checks for identity and then for Strings being the same size**

Intern String

- **To compare String by identity**

- Normally, string can be created by two ways

- By new String(...)

- ```
String s = new String("This is a string literal.");
```

- By String Literals

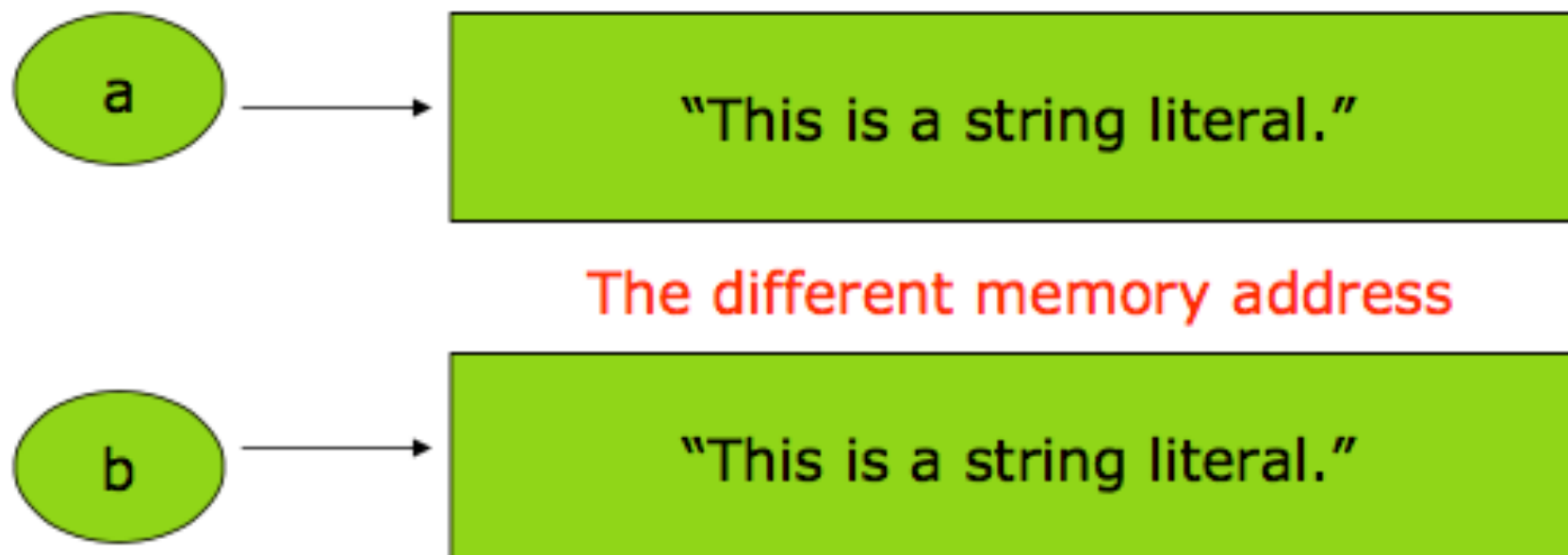
- ```
String s = "This is a string literal.";
```

- Create Strings by new String(...)

- JVM always allocate a new memory address for each new String created even if they are the same.

Intern String

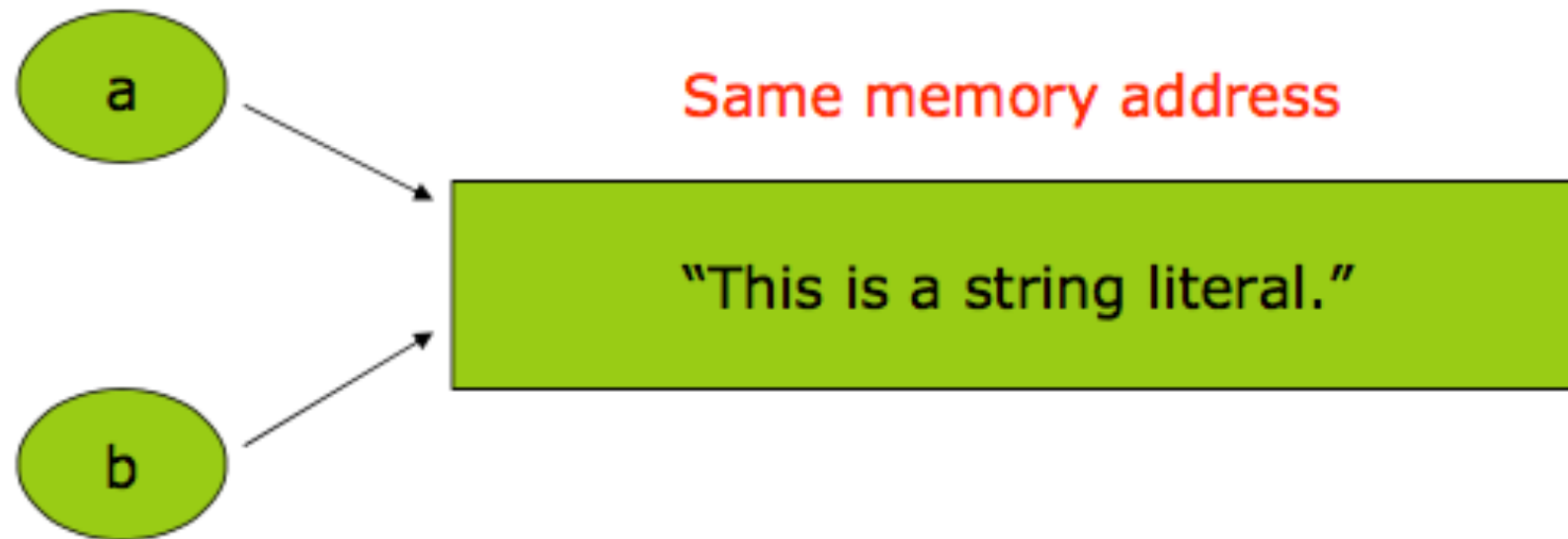
```
String a = new String("This is a string literal.");  
String b = new String("This is a string literal.");
```



Intern String

- Create Strings by Literals, Strings will be stored in Pool
- Double create Strings by laterals, they will share as a unique instances

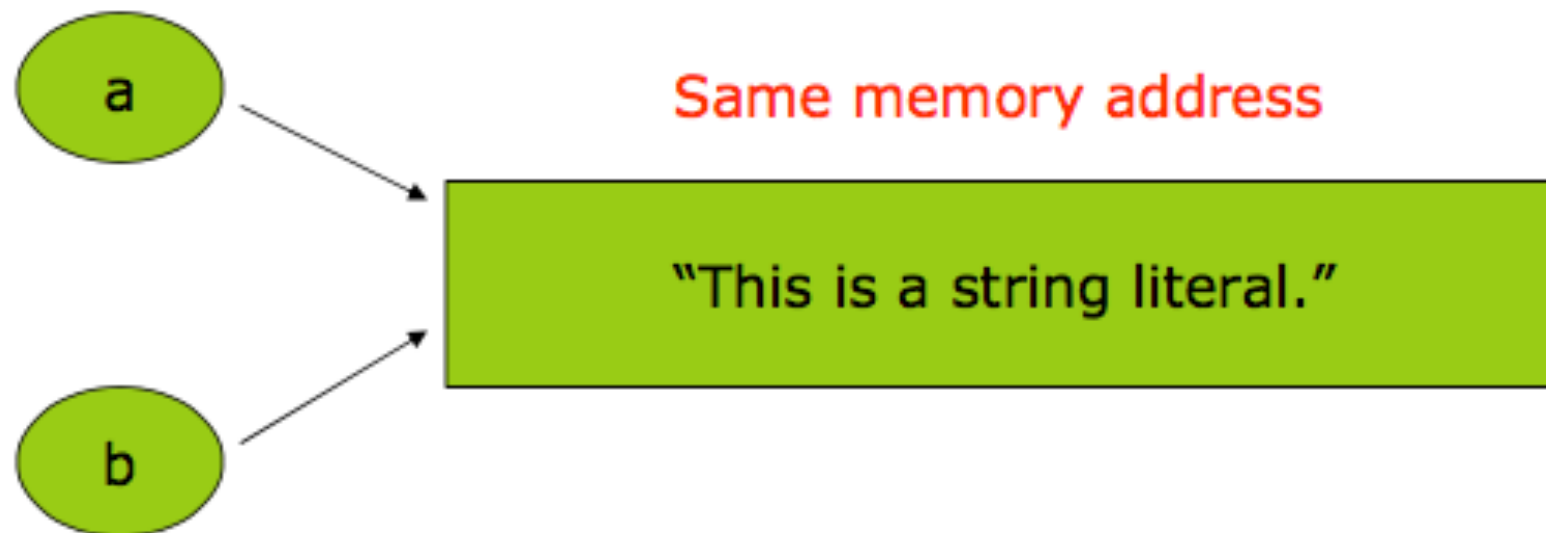
```
String a = "This is a string literal.";
String b = "This is a string literal.";
```



Intern String

- We can point two Strings variable to the same address if they are the same values, by using `String.intern()` method

```
String a = new String("This is a string literal.").intern();  
String b = new String("This is a string literal.").intern();
```



Intern String

- The idea is ... intern String could be used to compare String by identity
- What "compare by identity" means?

If (a == b)

Identity comparison
(by reference)

If (a.equals(b))

Value comparison

Intern String

- **By using reference, Identity comparison is fast**
- **In traditional style, String must be compare by equals() to avoid the negative result**
- **But Intern String...**
 - If Strings have different value they also have different address.
 - If Strings have same value they also have the same address.
 - So we can say that
 - `(a == b)` is equivalent to `(a.equals(b))`
 - For these string variables
 - `String a = "abc";`
 - `String b = "abc";`
 - `String c = new String("abc").intern();`
 - They are pointed to the same address with the same value

Intern String

```
for (int i =0; i < loop; i++)  
{  
    if (a.equals(b))  
    {  
    }  
}
```

Takes 312 ms

```
for (int i =0; i < loop; i++)  
{  
    if (a == b)  
    {  
    }  
}
```

Takes 32 ms

Intern String

- **String.intern() comes with overhead as there is a step to cache**
- **Use Intern String if they are planed to compare two or more times**
- **char array instead of String**
- **Avoid doing some stuffs by String object itself for optimal performance**

Intern String

```
String x = "abcdefghijklmn";  
for (int i = 0; i < loop; i++)  
{  
    if (x.charAt(5) == 'x')  
    {  
    }  
}
```

Takes 281 ms

```
String x = "abcdefghijklmn";  
char y[] = x.toCharArray();  
for (int i = 0; i < loop; i++)  
{  
    if ( (20 < y.length && 20 >= 0) && y[20] == 'x')  
    {  
    }  
}
```

Takes 156 ms

Exception and Casting

Stop exception to be thrown
if it is possible

Exception is really expensively to execute

Avoid Exception

```
Object obj = null;
for (int i = 0; i < loop; i++)
{
    try
    {
        obj.hashCode();
    } catch (Exception e) {}
}
```

Takes 18563 ms

```
Object obj = null;
for (int i = 0; i < loop; i++)
{
    if (obj != null)
    {
        obj.hashCode();
    }
}
```

Takes 16 ms

Cast as Less

We can reduce runtime cost by grouping cast object which is several used

Cast as Less

```
Integer io = new Integer(0);
Object obj = (Object)io;
for (int i = 0; i < loop; i++)
{
    if (obj instanceof Integer)
    {
        byte x = ((Integer) obj).byteValue();
        double d = ((Integer) obj).doubleValue();
        float f = ((Integer) obj).floatValue();
    }
}
```

Takes 31 ms

```
for (int i = 0; i < loop; i++)
{
    if (obj instanceof Integer)
    {
        Integer icast = (Integer)obj;
        byte x = icast.byteValue();
        double d = icast.doubleValue();
        float f = icast.floatValue();
    }
}
```

Takes 16 ms

The Rhythm of motion

Loop Optimization

- There are several ways to make a faster loop
- Don't terminate loop with method calls
 - Method Call generates some overhead in Object Oriented Paradigm

```
byte x[] = new byte[loop];
for (int i = 0; i < x.length; i++)
{
    for (int j = 0; j < x.length; j++)
    {
    }
}
```

Takes 109 ms

```
byte x[] = new byte[loop];
int length = x.length;
for (int i = 0; i < length; i++)
{
    for (int j = 0; j < length; j++)
    {
    }
}
```

Takes 62 ms

Loop Optimization

- **Use int to iterate over loop**
 - VM is optimized to use int for loop iteration not by byte, short, char

```
for (int i = 0; i < length; i++)  
{  
    for (int j = 0; j < length; j++)  
    {  
    }  
}
```

Takes 62 ms

```
for (short i = 0; i < length; i++)  
{  
    for (short j = 0; j < length; j++)  
    {  
    }  
}
```

Takes 125 ms

Loop Optimization

- Use `System.arraycopy(...)` for copying object instead of running over loop
- `System.arraycopy()` is native function It is efficiently to use

```
for (int i = 0; i < length; i++)  
{  
    x[i] = y[i];  
}
```

Takes 62 ms

```
System.arraycopy(x, 0, y, 0, x.length);
```

Takes 16 ms

Loop Optimization

- **Terminate loop by primitive use not by function or variable**
- Primitive comparison is more efficient than function or variable comparison

```
for(int i = 0; i < countArr.length; i++)  
{  
    for(int j = 0; j < countArr.length; j++)  
    {  
  
    }  
}
```

Takes 424 ms

```
for(int i = countArr.length-1; i >= 0; i--)  
{  
    for(int j = countArr.length-1; j >= 0; j--)  
    {  
  
    }  
}
```

Takes 298 ms

Loop Optimization

- **The average time of switch vs. if-else is about equally in random case**
- Switch is quite fast if the case falls into the middle but slower than if-else in case of falling at the beginning or default case
- ** Test against a contiguous range of case values eg, 1,2,3,4,..

```
for(int i = 0; i < loop; i++)  
{  
    if (i%10== 0)  
    {  
  
    } else if (i%10 == 1)  
    {  
  
    ...  
    } else if (i%10 == 8)  
    {  
    } else if (i%10 == 9)  
    {  
    }  
}
```

Takes 2623 ms

```
for(int i = 0; i < loop; i++)  
{  
    switch (i%10)  
    {  
        case 0: break;  
        case 1: break;  
        ...  
        case 7: break;  
        case 8: break;  
        default: break;  
    }  
}
```

Takes 2608 ms

Recursion

- **Recursive Algorithm**
 - Recursive function is easy to read but it costs for each recursion
- **Tail Recursion**
 - A recursive function for which each recursive call to itself is a reduction of the original call.

Recursive vs. Tail-Recursive

```
public static long factorial1(int n)
{
    if (n < 2) return 1L;
    else return n*factorial1(n-1);
}
```

Takes 172 ms

```
public static long factorial1a(int n)
{
    if (n < 2) return 1L;
    else return factorial1b(n, 1L);
}
public static long factorial1b(int n, long result)
{
    if (n == 2) return 2L*result;
    else return factorial1b(n-1, result*n);
}
```

Takes 125 ms

Dynamic Cached Recursive

- Do cache to gain more speed

```
public static long factorial1(int n)
{
    if (n < 2) return 1L;
    else return n*factorial1(n-1);
}
```

Takes 172 ms

```
public static final int CACHE_SIZE = 15;
public static final long[] factorial3Cache = new long[CACHE_SIZE];
public static long factorial3(int n)
{
    if (n < 2) return 1L;
    else if (n < CACHE_SIZE)
    {
        if (factorial3Cache[n] == 0)
            factorial3Cache[n] = n*factorial3(n-1);
        return factorial3Cache[n];
    }
    else return n*factorial3(n-1);
}
```

Takes 94 ms

Recursion Summary

Dynamic-Cached Tail Recursive

is better than

Tail Recursive

is better than

Recursive

**Use appropriate
collection**

Accession

ArrayList vs. LinkedList

Random Access

```
ArrayList al = new ArrayList();
```

```
for (int i = 0; i < loop; i++)  
{  
    al.get(i);  
}
```

Takes 281 ms

```
LinkedList ll = new LinkedList();
```

```
for (int i = 0; i < loop; i++)  
{  
    ll.get(i);  
}
```

Takes 5828 ms

Sequential Access

```
ArrayList al = new ArrayList();  
for (Iterator i = al.iterator(); i.hasNext();) {  
    i.next();  
}
```

Takes 1375 ms

```
LinkedList ll = new LinkedList();  
for (Iterator i = ll.iterator(); i.hasNext();) {  
    i.next();  
}
```

Takes 1047 ms

ArrayList is good for **random** access

LinkedList is good for **sequential** access

Random vs. Sequential Access

```
ArrayList al = new ArrayList();
```

```
for (int i = 0; i < loop; i++)  
{  
    al.get(i);  
}
```

Takes 281 ms

```
LinkedList ll = new LinkedList();
```

```
for (Iterator i = ll.iterator(); i.hasNext();)  
{  
    i.next();  
}
```

Takes 1047 ms

Random Access is better than
Sequential Access

Insertion

ArrayList vs. LinkedList

Insertion at zero index

```
ArrayList al = new ArrayList();  
  
for (int i = 0; i < loop; i++)  
{  
    al.add(0, Integer.valueOf(i));  
}
```

Takes 328 ms

```
LinkedList ll = new LinkedList();  
for (int i = 0; i < loop; i++)  
{  
    ll.add(0, Integer.valueOf(i));  
}
```

Takes 109 ms

LinkedList does insertion
better than ArrayList

Vector is like **ArrayList**
but it is **synchronized**
version

Accession and Insertion

Vector vs. ArrayList

Random Accession

```
ArrayList al = new ArrayList();  
for (int i = 0; i < loop; i++)  
{  
    al.get(i);  
}
```

Takes 281 ms

```
Vector vt = new Vector();  
for (int i = 0; i < loop; i++)  
{  
    vt.get(i);  
}
```

Takes 422 ms

Sequential Accession

```
ArrayList al = new ArrayList();  
for (Iterator i = al.iterator(); i.hasNext();) {  
    i.next();  
}
```

Takes 1375 ms

```
Vector vt = new Vector();  
for (Iterator i = vt.iterator(); i.hasNext();) {  
    i.next();  
}
```

Takes 1890 ms

Insertion

```
ArrayList al = new ArrayList();  
for (int i = 1; i < loop; i++)  
{  
    al.add(0, Integer.valueOf(i));  
}
```

Takes 328 ms

```
Vector vt = new Vector();  
for (int i = 0; i < loop; i++)  
{  
    vt.add(0, Integer.valueOf(i));  
}
```

Takes 360 ms

Vector is slower than ArrayList
in every method

Use Vector if only synchronize needed

Addition and Accession

Hashtable vs HashMap

Addition

```
Hashtable ht = new Hashtable();  
for (int i = 0; i < loop; i++)  
{  
    ht.put(Integer.valueOf(i), Integer.valueOf(i));  
}
```

Takes 453 ms

```
HashMap hm = new HashMap();  
for (int i = 0; i < loop; i++)  
{  
    hm.put(Integer.valueOf(i), Integer.valueOf(i));  
}
```

Takes 328 ms

Accession

```
Hashtable ht = new Hashtable();  
for (int i = 0; i < loop; i++)  
{  
    ht.get(Integer.valueOf(i));  
}
```

Takes 94 ms

```
HashMap hm = new HashMap();  
for (int i = 0; i < loop; i++)  
{  
    hm.get(Integer.valueOf(i));  
}
```

Takes 47 ms

Hashtable is synchronized
so it is slower than HashMap