JVM Memory Management - Garbage Collection, GC Tools, Java References

Azat Satklichov
azats@seznam.cz,
https://github.com/azatsatklichov/Java-Features/tree/master/src/main/java/features/jvm/memory/management

Agenda

- Internal Java Memory Model
- Hardware Memory Architecture All Types of Memory areas Allocated by the JVM
- Java Memory Model for Non Heap Memory
- Java Memory Model for Heap and Perm Spaces
- Memory Modifications and Issues
- Where Storage Lives
- Differences Between Stack and Heap Memory
- Types of Garbage Collections
- How Garbage Collection Works – Minor, Major and Full GC
- Garbage Collectors Implementations
- Garbage Collection Tuning
- Garbage Collection Tools - MXBeans, jstat, visualvm, visualgc, jcmd, jmc,
- Monitoring Tools - jcmd, FlightRecorder, Java Mission Controller
- Java Reference Classes – Strong -> Soft -> Weak -> Phantom

Internal Java Memory Model

Java memory model specifies how the JVM corporates with computer’s RAM.
JVM divides memory between thread stacks and heap.
Each thread running in the Java virtual machine has its own thread stack.

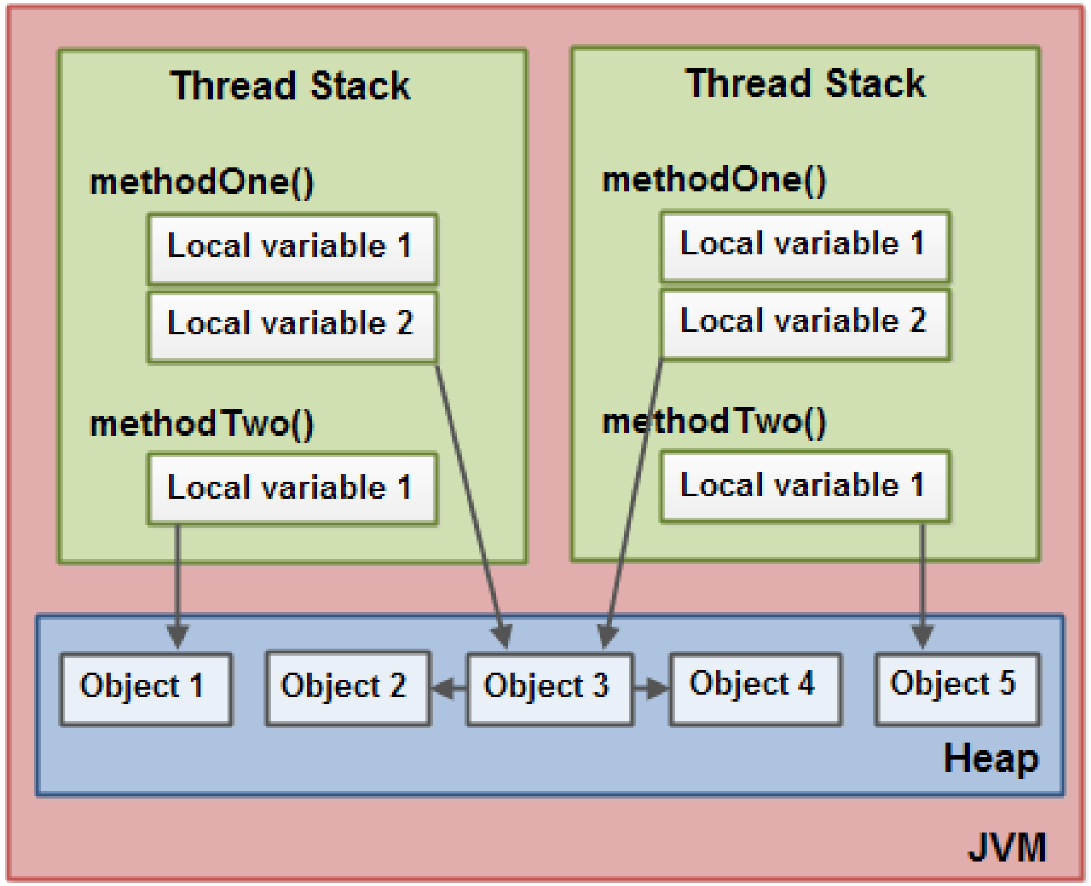
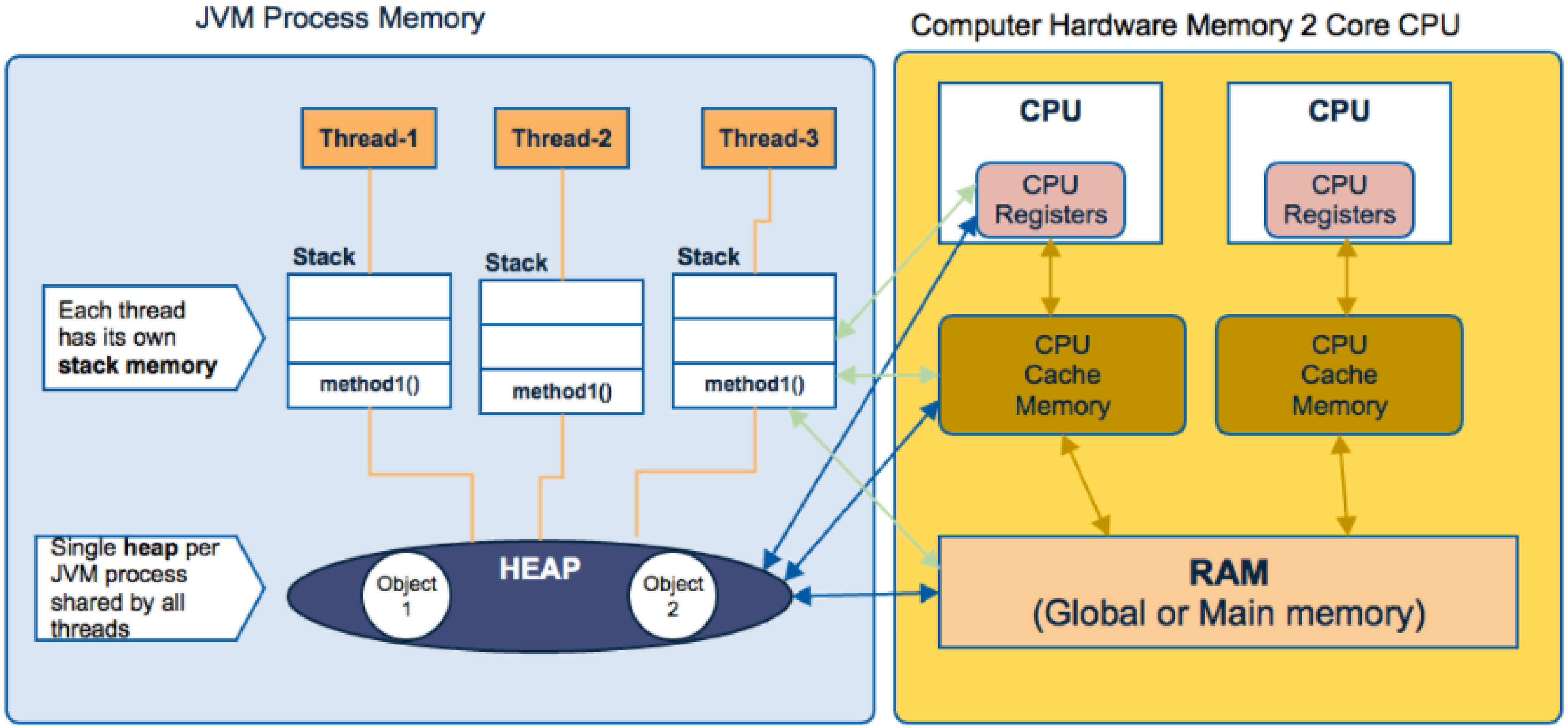


Diagram illustrating the **call stack** and local variables stored on the thread stacks, and objects stored on **the heap**.

Objects on the heap can be accessed by all threads that have a reference to the object. When a thread has access to an object, it can also get access to that object's member variables. If two threads call a method on the same object at the same time, they will both have access to the object's member variables, but each thread will **have its own copy of the local variables**. Only local variables are stored on the thread stack.

Hardware Memory Architecture



Hardware memory architecture:

- Registers – fastest storage
- The Stack – in RAM
- Static Storage - (fixed) in RAM
- Constant Storage - ROM
- Non-RAM Storage (persistence)

The CPU can access its **cache memory (L(**Level) 1 and L2 , or even L3**) much faster than main memory**, but not as fast as it can access its **registers.** Hardware memory architecture **does not distinguish between thread stacks and heap.** On the hardware, both located in main memory. Parts of the <u>thread stacks</u> and <u>heap</u> may sometimes be present in CPU caches and in internal CPU registers, see in this diagram. When objects and variables can be stored in various different memory areas in the computer, **certain problems may occur**. See next slide:

Corporation of Java and Hardware Memory Models

The two main problems are:

- **Visibility** of thread updates (writes) to shared. If two or more threads are sharing an object, **without** using **volatile** declarations or **synchronization**, updates to the shared object made by one thread **may not be visible** to other threads.
- **Race conditions** when reading, checking and writing shared variables. See multithreading blog

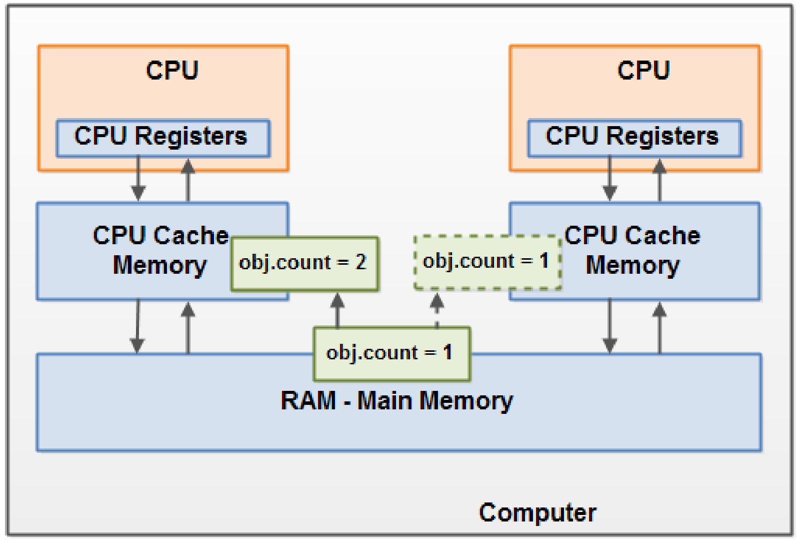
If two or more threads share an object, and more than one thread updates variables in that shared object, **race conditions** may occur.

Solution: use of synchronized block and volatile variables solve both.

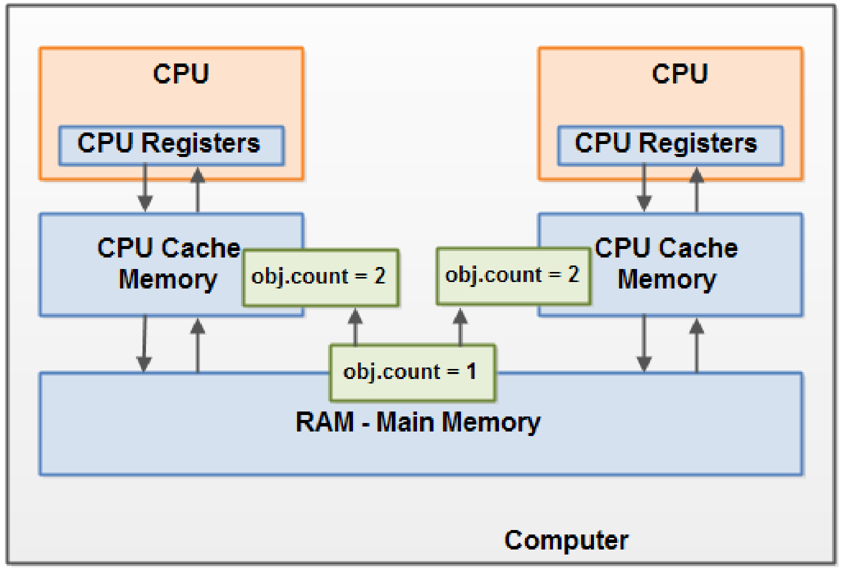
Java Happens Before Guarantee:

Rules how the Java VM and CPU is allowed to **reorder instructions** for performance gains. So in a multi-threading environment the reordering of the surrounding instructions does not result in a code that produces incorrect output.

- Instruction reordering
- Volatile visibility guarantee
- Constant folding and dead-code elimination (found by JMH)



Visibility

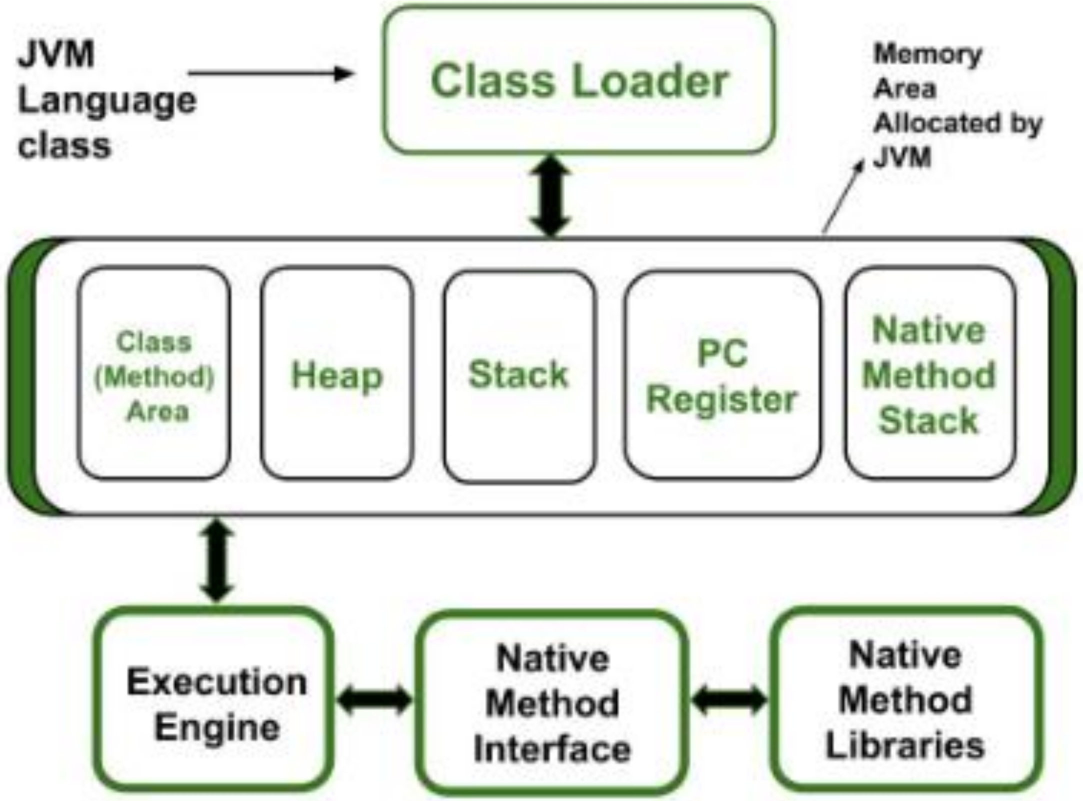


Race conditions

All Types of Memory areas Allocated by the JVM

JVM perform some particular **types of operations**:
1.Loading of code   2. Verification of code 3. Executing the code 4. Provides a run-time environment to the users

ClassLoader
It is a subsystem of JVM which is used to load class files. It is mainly responsible for three activities.
- Loading   - Linking   - Initialization



All these functions take different forms of memory structure, **in the JVM it is divided into 5 different parts**:

1. Class(Method) Area
2. Heap
3. Stack
4. Program Counter Register
5. Native Method Stack

Heap

The Heap area is the memory block where objects are created or objects are stored. Heap memory allocates memory for class interfaces and arrays (an array is an object). It is used to allocate memory to objects at run time

Stack

Each thread has a private JVM stack, created at the same time as the thread. It is used to store data and partial results which will be needed while returning value for method and performing dynamic linking.
Java Stack stores frames and a new frame is created each time at every invocation of the method. A frame is destroyed when its method invocation completes

Program Counter Register (PC)

PC registers mainly store the line number of the currently executing thread. Since it records the line number of each thread, PC registers are private to threads. As PC registers only record the address of current instructions, it is the only area that does **not define OutOfMemoryError** in JVM. PC register is capable of storing the return address or a native pointer on some specific platform..

Native method Stacks

Also called C stacks, native method stacks are **not written in Java language**. This memory is allocated for each thread when it's created and it can be of a fixed or dynamic nature. Native libraries are linked to a Java program through JNI or JNA(Java Native Access). Native methods are also associated with each thread.

Permanent Generation or 'Perm Gen' (Metaspace since Java 8) – not Heap Memory

- Contains the app. **metadata** required by the JVM to describe the classes and methods used in the app. It stores per-class structures such as runtime constant pool, field and method data, and the code for methods and constructors, as well as interned Strings
- Perm Gen is populated by JVM at runtime based on the classes used by the app.
- Perm Gen also contains Java SE library classes and methods.
- Perm Gen objects are garbage collected in a **full garbage collection.**
- Size tune: -XX:PermSize and -XX:MaxPermSize

Class (Method) Area

Method Area is part of space in the **Perm Gen** and used to store class structure (**runtime constants and static variables**) and code for methods and constructors.

Runtime Constant Pool

Runtime constant pool is per-class runtime representation of constant pool in a class. It **contains class runtime constants and static methods**. Runtime constant pool is part of the **method area**.

Memory Pool

Created by JVM memory managers to create a **pool of immutable objects (e.g.** String Pool **)** if the implementation supports it. Memory Pool **can belong to Heap or Perm Gen**, depending on the JVM memory manager implementation.

Java Memory Model for Non Heap Memory

Non-Heap Memory

Permanent Generation (since Java 8 it is replaced by **Metaspace**)

Cache Memory

Stores compiled code (i.e. native code) generated by **JIT compiler**, JVM internal structures, loaded profiler agent code and data, etc. To avoid *"CodeCache is full … tune it*

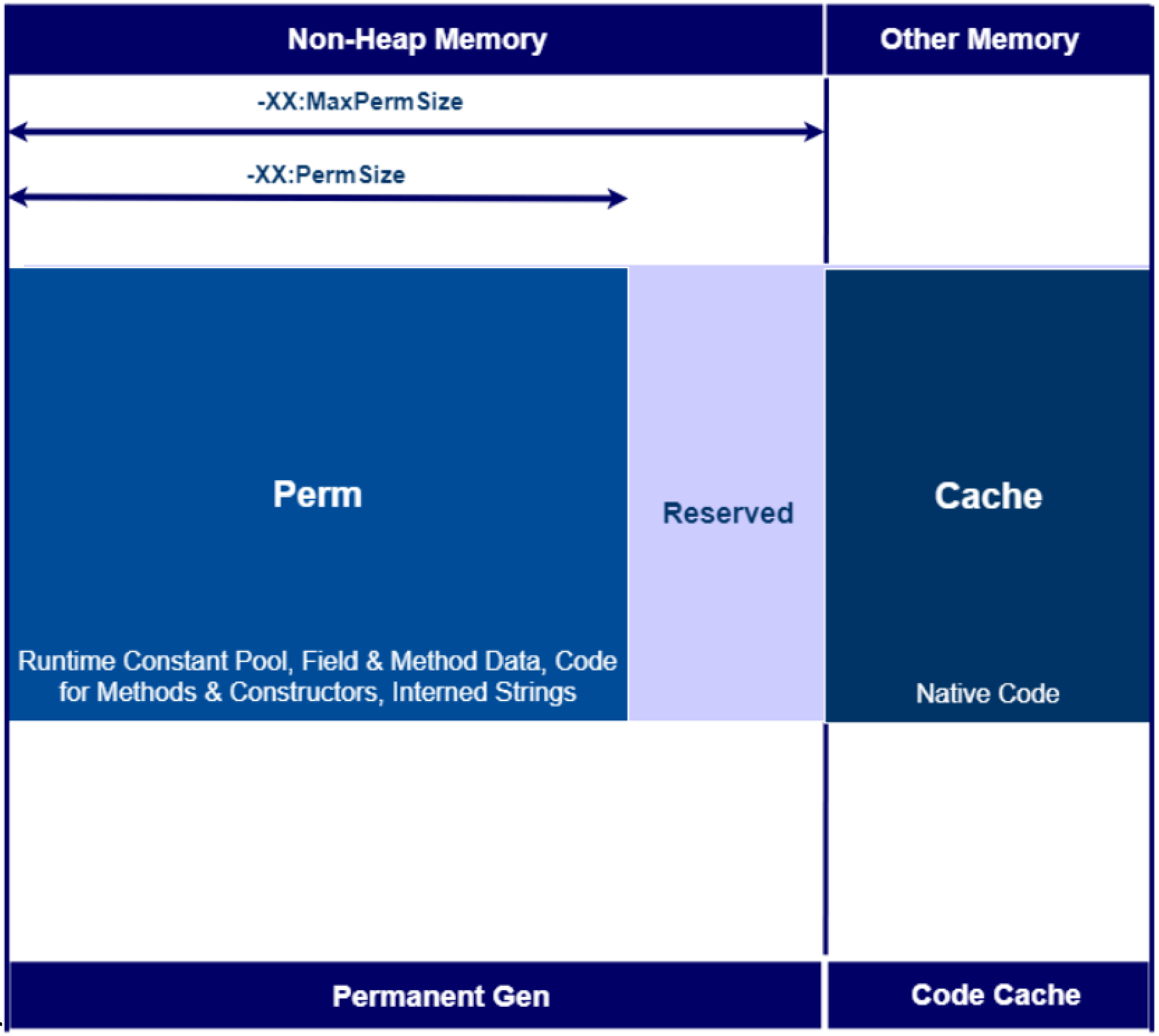
InitialCodeCacheSize – 160K default

***ReservedCodeCacheSize* – the default maximum size is 48MB**

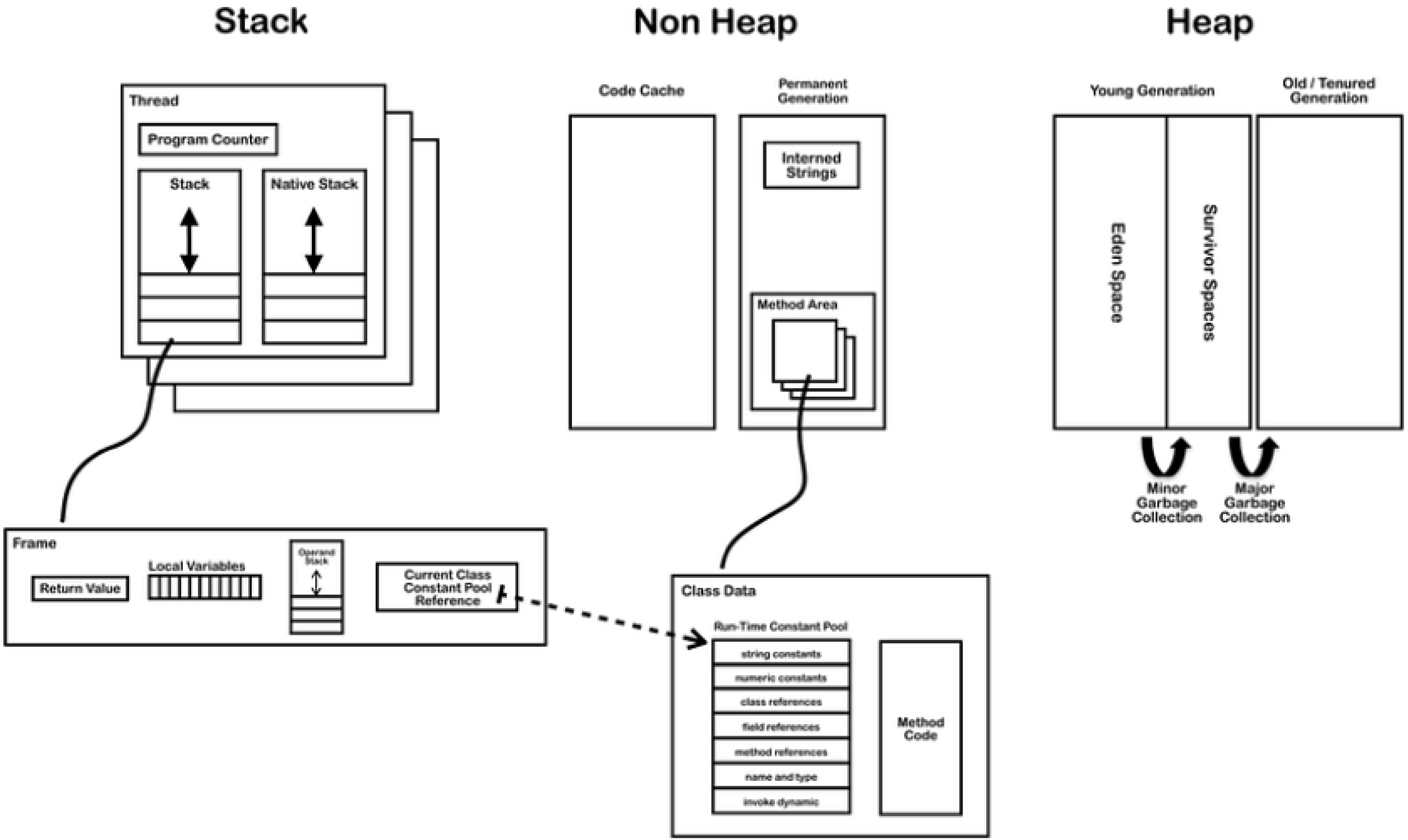
CodeCacheExpansionSize – the expansion size of the code cache, 32KB or 64KB

Segmented Code Cache – since Java 9 (JEP 197)

Three distinct segments (**non-method, profiled-code, non-profiled** segments) each of which contains a particular type of compiled code.



JVM Non-Heap & Cache Memory
(Image: PlatformEngineer.com)

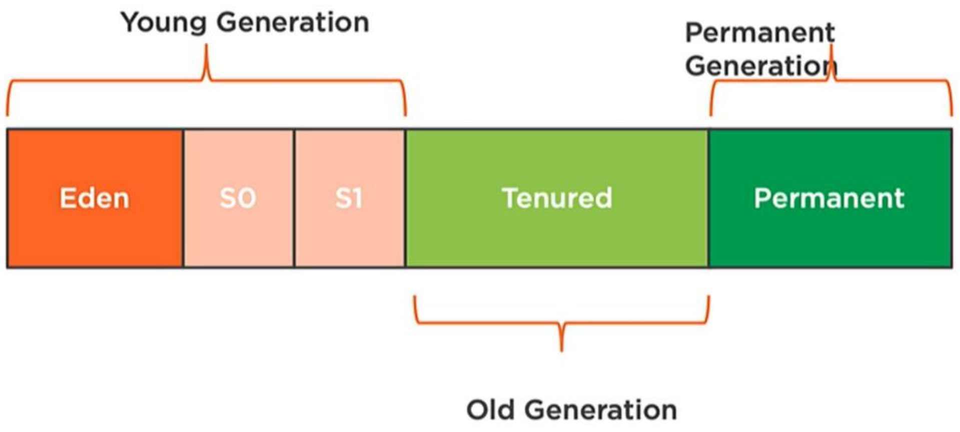


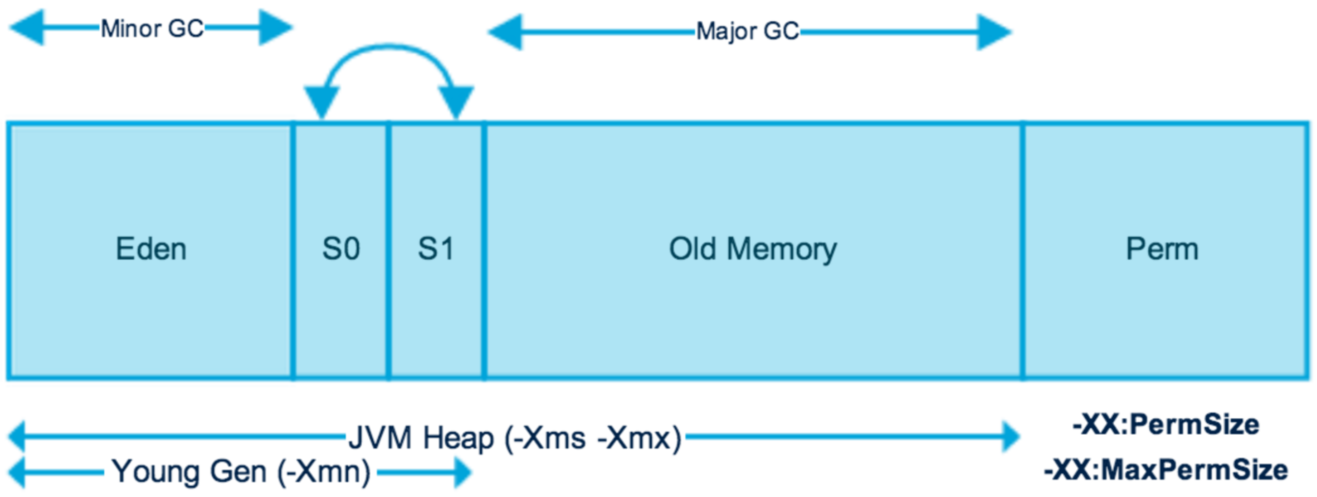
JVM Stack, Non-Heap, and Heap (Image: jamesdbloom.com)

Java Memory Model for Heap and Perm Spaces

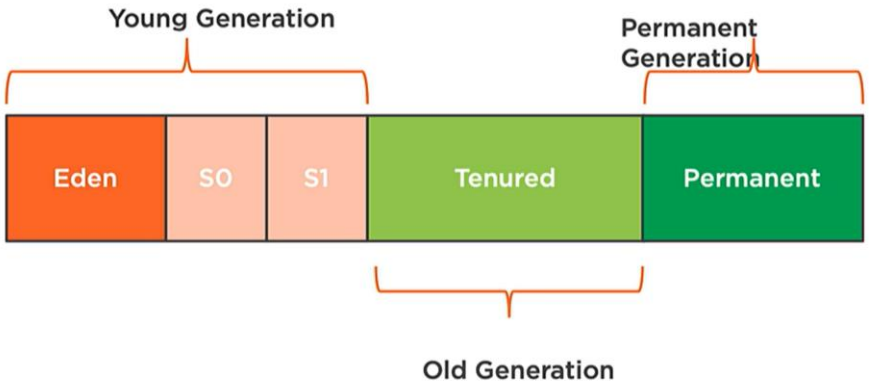
- Memory space is broken into two spaces: Young and Old (Tenured Space) Generations.
- The JVM internally separates the Young Generation into Eden and Survivor Spaces (S0, S1).
- Newly allocated objects are going to Eden [Eden Gardens] space.
- Objects **survived** in young generation promoted to one of Survivor spaces.
- One survivor space is used at a time, and objects are copied between survivor spaces
- Old generation where **long lived objects** (survived many times) live (most stable objects, ..)
- GC runs frequently on Young Gen. So motto is: die young or live forever.

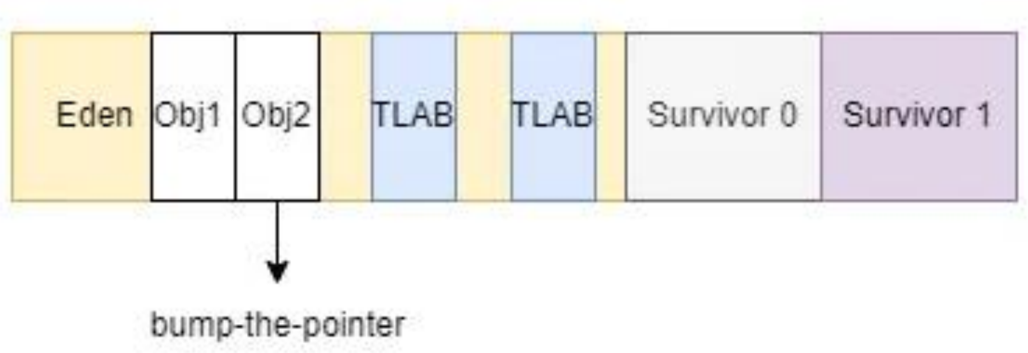
- Permanent space (never GC run here) – used by Java Runtime (class info, symbolic instructions stored)





Memory allocation in Heap





Java instances are mostly in Heap Area. As Heap Area is shared among threads, heap allocation **needs to introduce locks** to ensure **thread safety**, which incurs a higher cost of instance creation. In order to improve the allocation efficiency, Eden space in Young Generation in HostSpot VM uses **two technics** to achieve this — **bump-the-pointer** and TLAB (**Thread-Local Allocation Buffers**). Bump-the-pointer aims to track the last instance created so when a new instance is created, we only need to check if there is enough space after the last instance. TLAB is pertaining to multi-threading. It will create a new space in Eden Space for each new thread. The size of TLAB can be configured using **-XX:TLABWasteTargetPercent** (default 1%). Usually, JVM will prioritize TLAB allocation. If the instance is too big or no space is left for TLAB, JVM will continue to allocate in Heap Area.

Algorithms: bump the pointer, TLAB, using free memory addresses, using card-tables, randomly via reserved space

Memory Modifications

The above Java memory model is the most commonly-discussed implementation. However, the latest JVM versions have different modifications such as introducing the following new memory spaces.

- **Keep Area** — a new memory space in the **Young Generation** to contain the most recently allocated objects. No GC is performed until the next Young Generation. This area prevents objects from being promoted just because they were allocated right before a young collection is started.
- **Metaspace** — Since **Java 8**, Permanent Generation is replaced by Metaspace. It can auto increase its size (up to what the underlying OS provides) even though Perm Gen always has a fixed maximum size. As long as the classloader is alive, the metadata remains alive in the Metaspace and can't be freed.
- **Segmented Code Cache –** since Java 9 (JEP 197)

NOTE: You are always advised to go through the vendor docs to find out what works for your JVM version.

Memory Issues

Once there is a critical memory issue, JVM throws exception

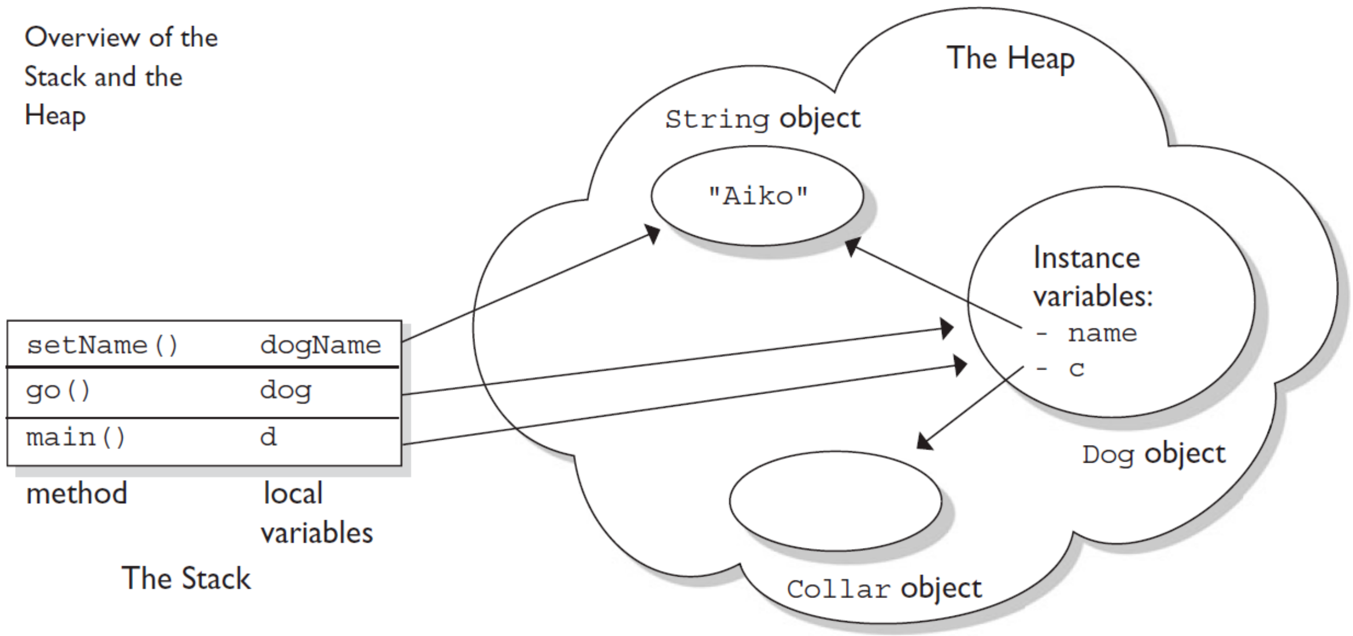
- **java.lang.StackOverFlowError** — thrown when Stack Memory is full
- **java.lang.OutOfMemoryError: Java heap space** thrown when Heap Memory is full
- **java.lang.OutOfMemoryError: GC Overhead limit exceeded** — thrown when app. Spend more time in GC
- **java.lang.OutOfMemoryError: Permgen space** — thrown when Permanent Generation space is full
- **java.lang.OutOfMemoryError: Metaspace** — thrown when Metaspace is full (since Java 8)
- **java.lang.OutOfMemoryError: Unable to create new native thread** — thrown when JVM native code can't create a new native thread in underlying OS because previously created threads consume all the available memory for the JVM
- **java.lang.OutOfMemoryError: request size bytes for reason** — when swap memory space is fully consumed by app.
- **java.lang.OutOfMemoryError: Requested array size exceeds VM limit**– app. uses an array size more than allowed in OS

These exceptions can only indicate the impact that the JVM had, not the actual error. The actual error and its root cause conditions can occur somewhere in your code (e.g. memory leak, GC issue, synchronization problem), resource allocation, or maybe even hardware setting.

Therefore, you need to monitor resource usage, profile each category, go through heap dumps, check and debug/optimize your code etc. And if none of your efforts seems to work and/or your context knowledge indicates that you need more resources, go for it.

Where Storage Lives

CODE: WhereStorageLives.java

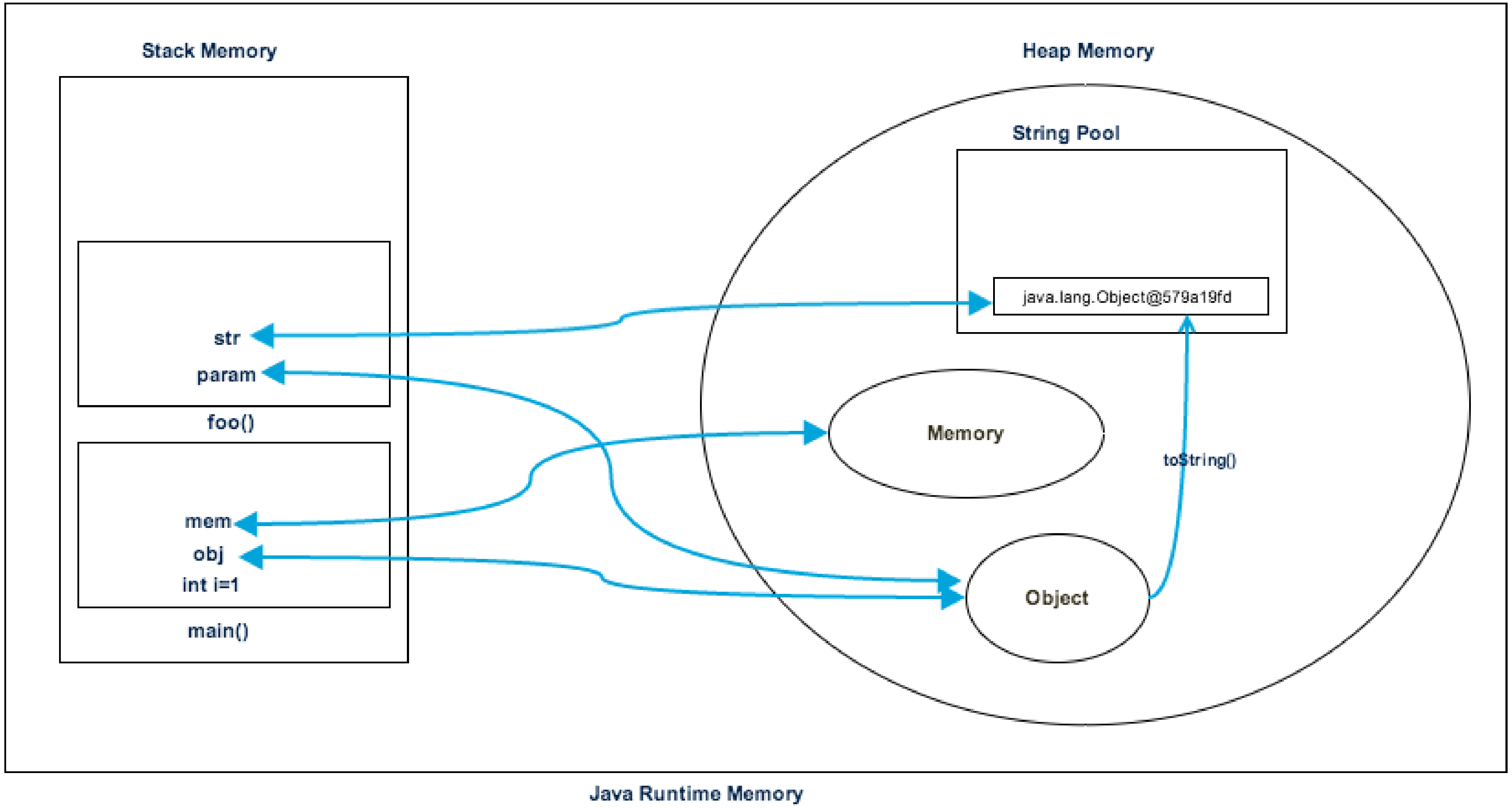


- Instance variables(references), and member primitives and objects live on HEAP
- Methods and Local (automatic/stack/method) variables[references and primitives] live on STACK, but its referencing object or created object live on HEAP.
- Static class variables are also stored on the HEAP along with the class definition.

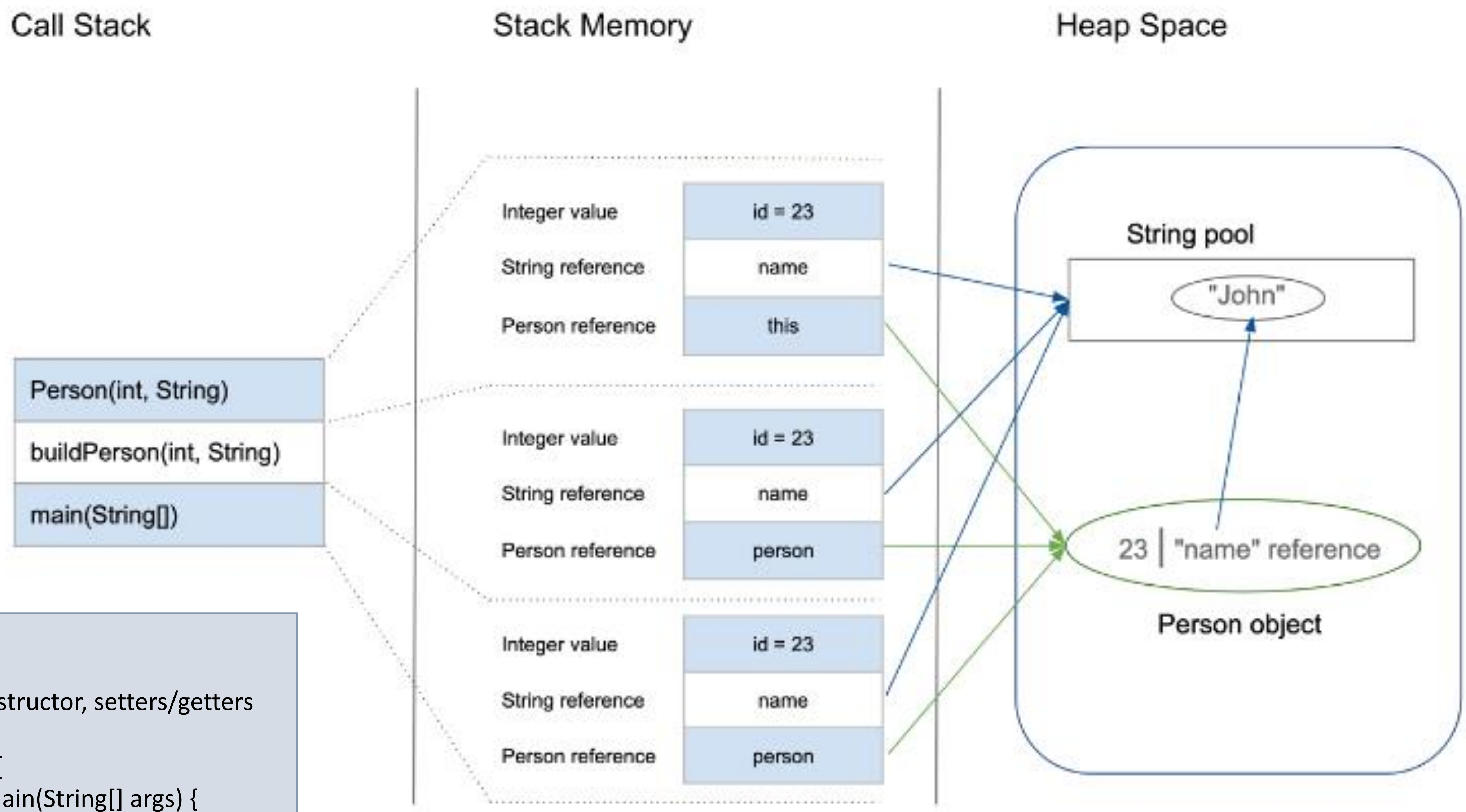
Note: local variables created on stack whereas Instance variables created on HEAP

Where Storage Lives

```
class Memory {

public static void main(String[]
args) {
int i = 1;
Object obj = new Object();
Memory mem = new Memory();
mem.foo(obj);
}

private void foo(Object param) {
String str = param.toString();
System.out.println(str);
}

}
```



Reference: [Difference between Stack and Heap Memory](#).




```
class Person {
int pid;
String name;// constructor, setters/getters
}
public class Driver {
public static void main(String[] args) {
int id = 23;
String pName = "Jon";
Person p = null;
p = new Person(id, pName); } }
```

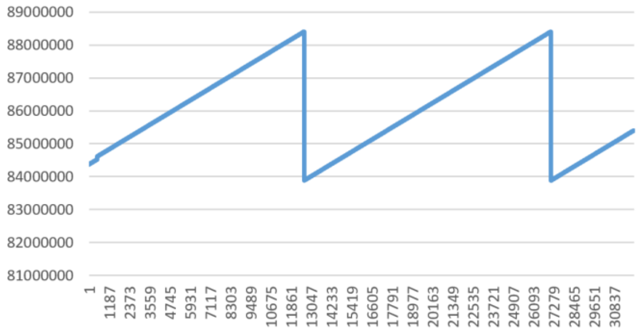
https://www.baeldung.com/java-stack-heap

Differences Between Stack and Heap Memory

- Heap is used by all the parts of the app. whereas stack memory is used only by one thread of execution.
- Whenever object created, it's always stored in the Heap and stack contains the reference to it. Stack memory only contains local primitive variables and reference variables to objects in heap space.
- Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that is getting referred from the method.
- Objects in the heap are globally accessible whereas stack memory can't be accessed by other threads (thread safe – because each thread operates in its own stack) . Stack memory allocated/deallocated when a method is called & returned. Heap memory managed by GC.
- Memory management in stack is done **in LIFO manner**. Heap memory is **more complex** because it's used globally, it is divided into Young-Generation, Old-Generation etc, more details at Java Garbage Collection.
- Stack memory is short-lived whereas heap memory lives from the start till the end of application execution.
- We can use **-Xms** and **-Xmx** JVM option to define the startup size and maximum size of heap memory. We can use **-Xss** to define the stack memory size. Stack **size is very less than Heap memory**.
- When stack memory is full, Java runtime **throws java.lang.StackOverFlowError** whereas if heap memory is full, it throws **java.lang.OutOfMemoryError.** Stack memory is **very fast** when compared to heap memory.

Java Garbage Collector

CODE: Memory Alloc. – **GC illustration** via addr of Obj using **Unsafe** by Sun.



Can we force to run GC? No

Java's garbage collector provides an automatic solution to memory management.

- Garbage collection cannot ensure that there is enough memory, only it manages available memory as efficiently as possible.
- The garbage collector is under the control of the JVM. The JVM decides when to run the garbage collector.
- JVM will typically run the garbage collector when it senses that memory is running low.
- It tracks each and every object available in the JVM heap space and removes unused ones.
- You can only ask [System.gc()] for GC (can't force) but no guarantee when it happens.
- **Pros**: Automatic memory management (compare by C/C++..) . **Cons:** Managed by JVM. Stop the World issue. Not like man-mem-efficiency
- An object is eligible for garbage collection when can't be reached by any live thread

The garbage collection implementation lives in the JVM.
Each JVM can implement its own version of garbage collection.

Java Garbage Collector

When Objects are Eligible for Collection?

Once Garbage collector runs, it can discover those objects and delete them.

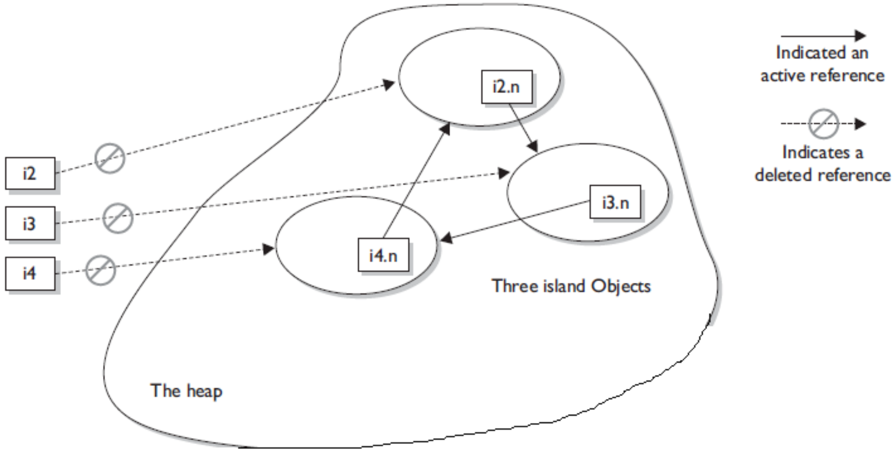
- Nulling a Reference
- Reassigning a Reference Variable
- Once method call is over BUT …
- Anonymous object – not referenced
- Islands of Isolation (circular) //self
- .. ? //pools (interning), caches, …

```java
public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("Hey GC");
    System.out.println(sb); //not eligible for collection
    sb = null;  //Nulling a Reference makes it Eligible for GC
    // Now StringBuffer object is eligible for collection
}
```

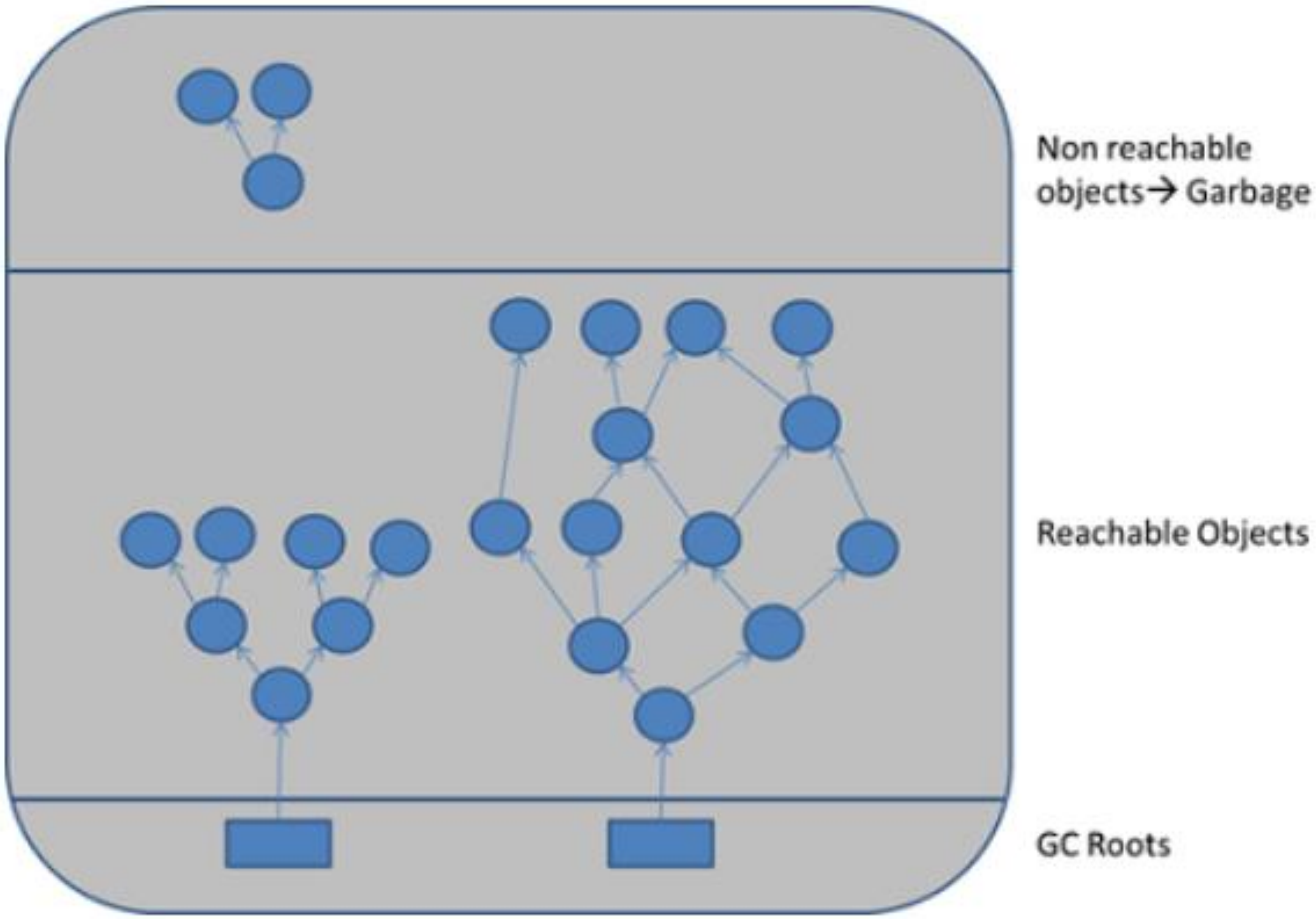
```java
public static void main(String[] args) {
    StringBuffer sb1 = new StringBuffer("Hello GC");
    StringBuffer sb2 = new StringBuffer("Hi GC");
    System.out.println(sb1); // "Hello GC" not eligible for collection
    sb1=sb2; // sb2 re-referenced to "Hi GC" object
    // Now "Hello GC" object
}
```

```java
public static void main(String[] args) {
    Date d = getDate();
    System.out.println("d = " + d);
}

public static Date getDate() {
    Date d2 = new Date();
    StringBuffer now = new StringBuffer(d2.toString());
    System.out.println(now);
    return d2;
    //StringBuffer object will be eligible for Collection
    //Date object will not be
}
```

```java
public static void main(String[] args) {
    IsolatingAReference i2 = new IsolatingAReference();
    IsolatingAReference i3 =
    IsolatingAReference i4 =
    i2.i = i3; // i2 refers t
    i3.i = i4; // i3 refers t
    i4.i = i2; // i4 refers t
    i2 = null;
    i3 = null;
    i4 = null;
    // do calc
```



new Person(“lenin”); //anonymous object



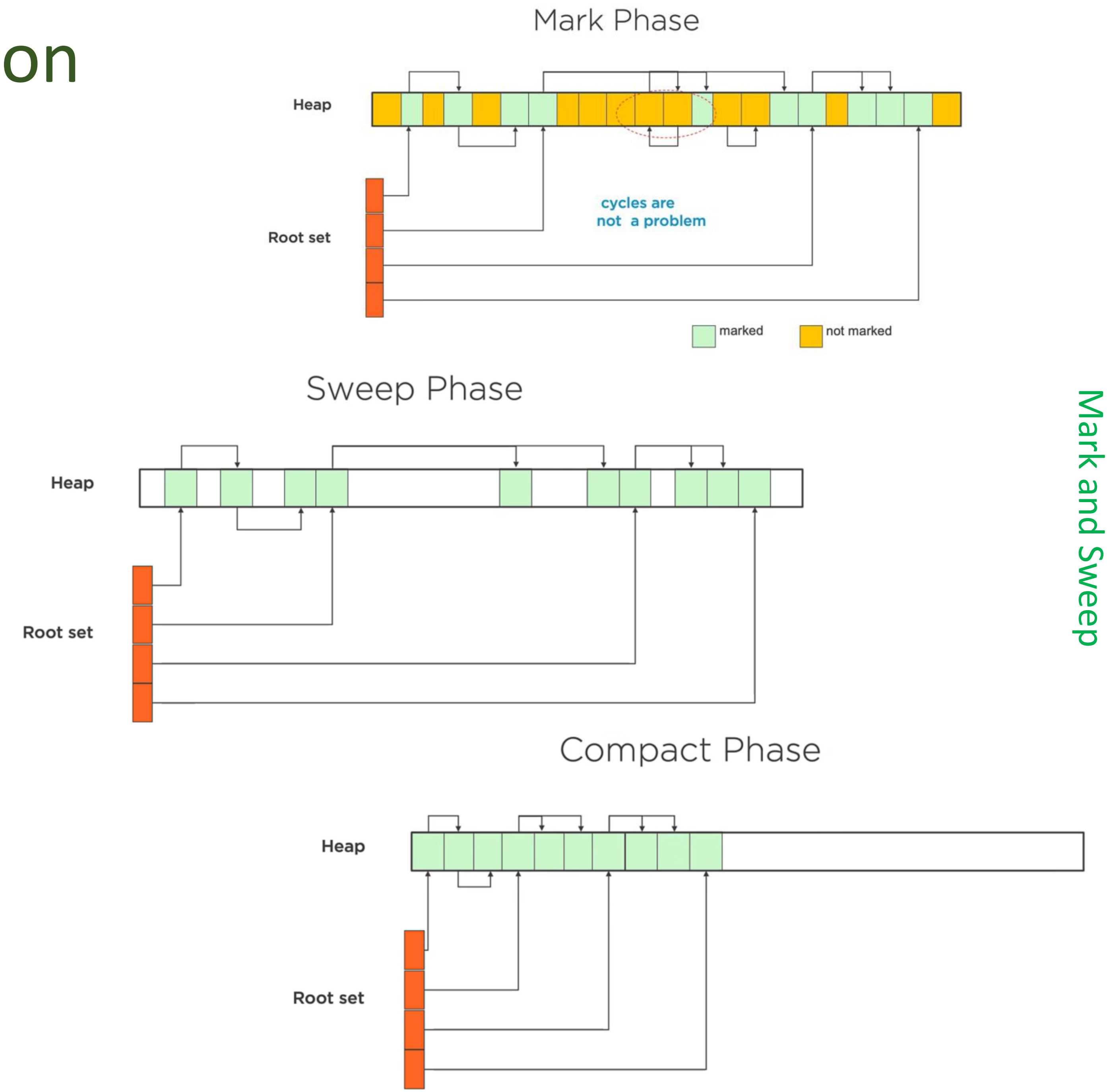
GC Roots are objects that are themselves referenced by the JVM and thus keep every other object from being garbage-collected (Image: dynatrace.com)

Types of Garbage Collection

- No Garbage Collection – no GC happens, e.g. new Epsilon GC

- Reference Counting – Count incremented once object started to be referenced, and count decreased once un-referenced, on e count zero it becomes eligible for GC. Issue with circular ref. It happens when one object refers to another, and that other one refers to the first object.

- Mark and Sweep [+ compact] – "**mark**" phase identifies the objects are still in being used, "**sweep**" phase removes un-used objects, "**compact**" phase defragments the memory.

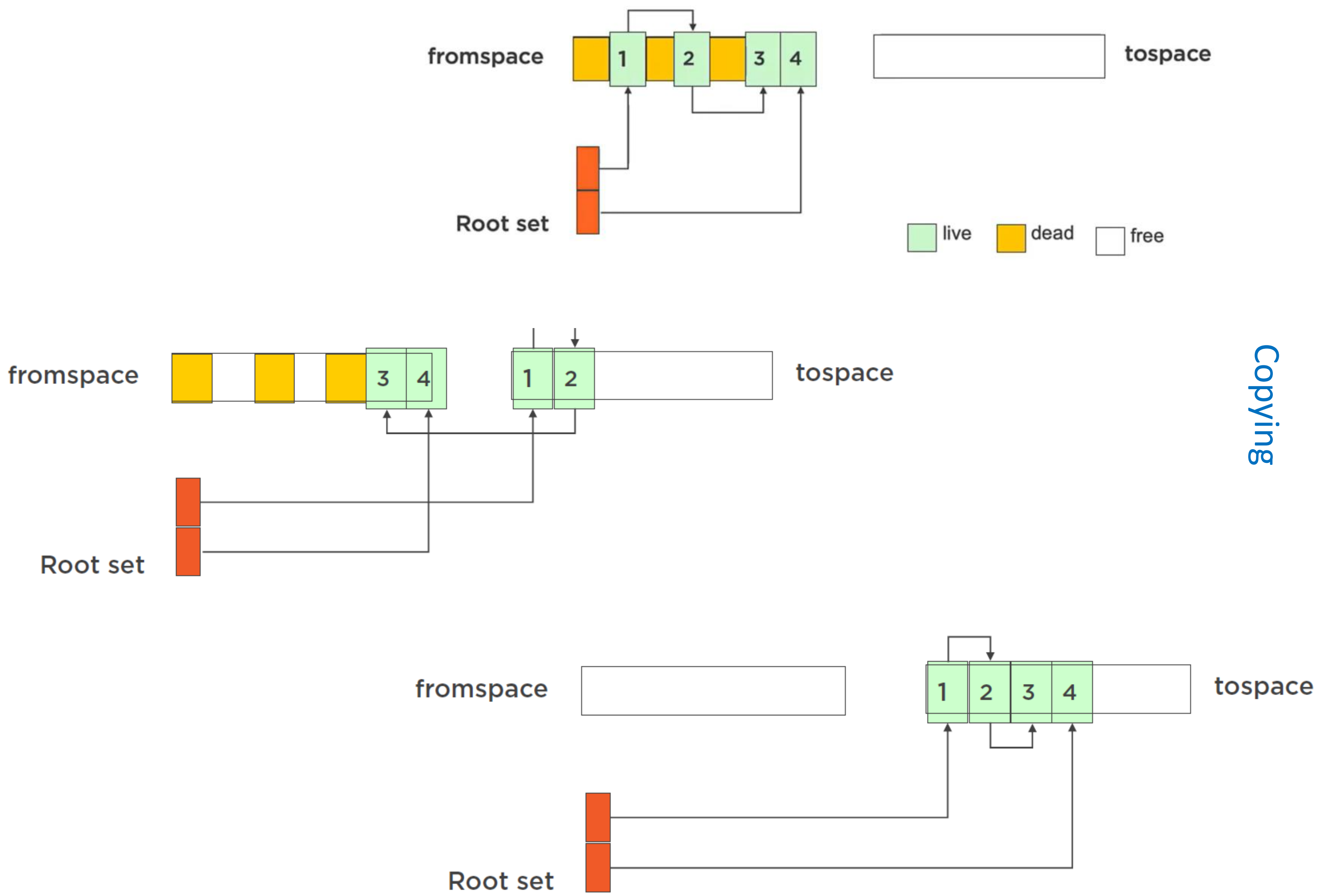


Mark and Sweep

Types of Garbage Collection

- Copying – mark & sweep happens but uses different spaces to manage the memory. During copying memory is compacted at the same time, and from space is cleared after all live objects are moved.
- Generational – Manages different generations for memory, long living objects (survived) are promoted to old-gen. Sweeps happen more often in young generations than old one.
- Incremental – does not scan all the memory all the time.

Copying Fromspace to Tospace

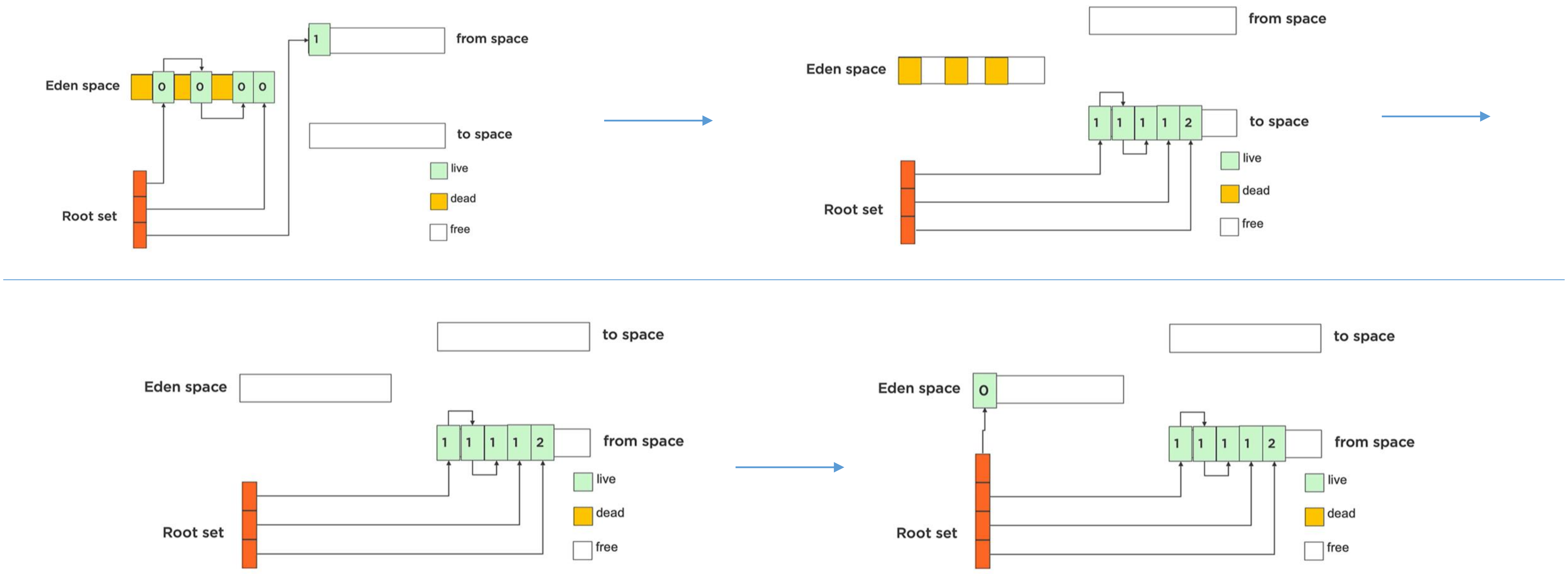


Copying

How Garbage Collection Works - Minor GC

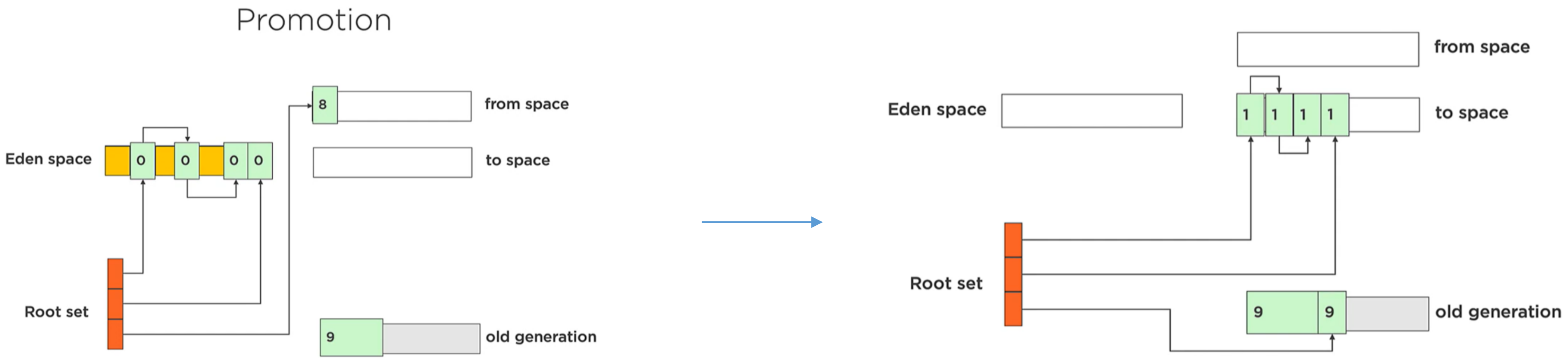
- Minor GC happens when the Young Generation is FULL, happens more frequently.
- New objects are allocated to Eden space. Initial survivor space is called ‘from space’
- When GC runs (**Minor GC happens on Eden Space**), objects are copied to newer survivor space, Eden is cleared, and survivor spaces will be swapped

Young Generation Allocation



How Garbage Collection Works – Major GC, Full GC

- **Major GC** triggered **when tenured** (Old Generation) **is FULL**. **Happens less frequently,** slower than Minor GC
- **Full GC -** Collects OLD and YOUNG gen. at the same time (on Oracle Java VM)
- In Major CG memory is copied from Young to Old generation – called promotion
  - after survived several times
  - once survivor space is full.
  - Or JVM has been told, create object in Old space (via JVM flag set: -XX:+AlwaysTenure)



Allocating Objects to Old Space

- Objects with certain size can be directly allocated to Old space. There is no JVM flag for this to allocate always.
- But can be done via: -XX:PretenureSizeThreshold=<n> if objSize>n then objects allocated to Old gen. If object size fits to TLAB, then JVM allocates it into TLAB (Thread Local Allocation Buffer). **TLAB stands for Thread Local Allocation Buffer** and it is a region inside Eden, which is exclusively assigned to a each thread, that is why no synchronization mechanism needed.

Garbage Collectors Implementations

Which GC to choose? Consider: Stop the world events (1), memory fragmentation(2-related to latency?), Throughput(3), Low latency(4). And profile (via mxbean, jstat, visualgc) the app. As close to production load.

- Serial Garbage Collector – simplest GC impl., works with a single thread (1) :

 >java -XX:+UseSerialGC -jar App.java

Good for apps. that do not have small pause time req. and to run on client-style machines.

- Parallel [**Parallel GC, Parallel Old GC**] Garbage Collector – also called **Throughput Collector** (3).

Uses multiple threads for managing heap space. Mostly used in production servers.

Parallel for MinorGC and serial (1) for MajorGC -XX:+UseParallelGC, and parallel for both -XX:+UseParallelOldGC.

Tune max. gc. *threads and pause time, throughput, and footprint* (heap size). Max-pause-goal: -XX:MaxGCPauseMillis=<N>

Serial and Parallel GC - allocates memory using "**bump the pointer**" algorithm, is fastest memory allocation.

- CMS Garbage Collector (**low latency**) (4) – Deprecated in Java 9, removed in 14. Concurrent Mark Sweep uses multiple garbage collector threads for gc. designed for apps. in need shorter gc. pauses, throughput is higher …

–XX:+UseConcMarkSweepGC , –XX:-UseParNewGC. With serial (1) young space collector

–XX:+UseConcMarkSweepGC , –XX:+UseParNewGC. With parallel young space collector

We can limit the number of threads in CMS collector using –XX::ParallelCMSThreads=<n> JVM option.

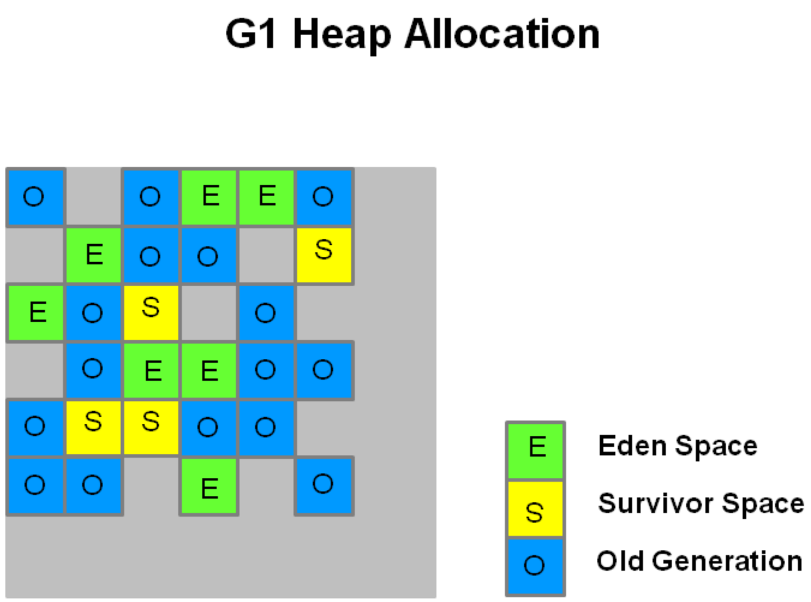
CMS allocates memory using "defragmentation"

Java 8 JVM has new param – Optimizing heap-memory for reducing the unnecessary use of memory by creating too many instances of the same *String.* > -XX:+UseStringDeduplication

Throughput[HIGH good] – number of trx(operation) per second. Latency[LOW good] – average time of a transaction.

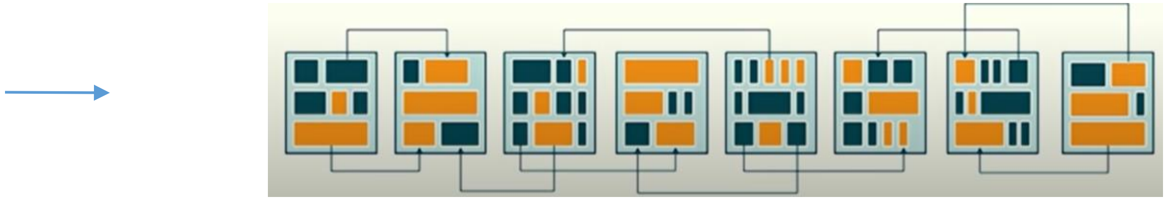
Garbage Collectors Implementations

- G1 (Garbage First, or **compacting-collector**) Garbage Collector – incremental GC, still (1) happens in Major collection. Paper 2004, experimental in Java 6, official in Java 7, and default GC since Java 9. Designed for apps running on multi-cores with large memory space. Goal is: **Througput/Latency** (3&4) **balance.** Tuneable pause goals. Since Java10 returns un-used memory, and Parallel.
- G1 allocates memory using "free memory addresses"



A bit more CPU intensive. Regions (#2000, size 2MB for 4GB heap)

- Evacuation (moved/copied between regions). Minor & Major collection,..
- >java -XX:+UseG1GC -jar App.java



Garbage Collectors Implementations

- Epsilon Collector – A **No-Op GC**. Apps with predictable, bounded memory usage, performance[stress] testing, short-lived obj. Allocates memory but not collect any garbage (memory allocation), once the Java heap is exhausted, the JVM will shut down.

Params: -XX:+UnlockExperimentalVMOptions, -XX:+UseEpsilonGC

- Z Garbage Collector (ZGC is **scalable low** latency GC) (4) [pause time **under 10 ms**, maybe in future 1 ms] – is a scalable low-latency garbage collector, experimental in Java 11 for Linux, in Java 14 for Win. & macOS. Since Java 15 ZGC is onwards. Single generation(all opt. done here via colored pointers), **No pause time increase with heap size increase**, Scale to **multi-terabyte heaps**. -XX:+UnlockExperimentalVMOptions, -XX:+UseZGC

- Shenandoah Garbage Collector (derived from G1, scalable low-latency garbage collector) –

experimental in Java 12, [Brooks pointers] support large Heaps, **Low pause times** (4)
Contributed by **RedHat** to OpenJDK. Became product feature in Java 15. Oracle JDK and Open JDK has not this feature. Single generation. Can be used in Java 8 so no need colored pointers like ZGC.
-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC

Throughput[HIGH good] – number of trx(operation) per second. Latency[LOW good] – average time of a transaction.

Garbage Collection Tuning – Which GC?

- **Java Garbage Collection Tuning** should be the last option to increasing the throughput of your app.

- Do it when you see a drop in performance because of longer GC timings causing application timeout.

- If you see **`java.lang.OutOfMemoryError:`** `PermGen space` errors in logs, then **try to monitor** and increase the Perm Gen memory space using **-XX:PermGen** and **-XX:MaxPermGen** JVM options.

- You might also try using **`-XX:+CMSClassUnloadingEnabled`** and check how it's performing with CMS Garbage collector. If you see a lot of Full GC operations, then you should try increasing Old generation memory space.

- Overall garbage collection tuning takes a lot of effort and time and there is no hard and fast rule for that.

- You would need to try different options (-XX:+UseG1GC or –XX::ParallelCMSThreads or etc.) and compare them to find out the best one suitable for your application.

- No easy answer to choose which one – just profile your app., and test app. Using different GC and see which GC fits (gives HIGH Througput, and LOW Latency or best BALANCE) most

Garbage Collection Tools

- **MX Beans** – management bean [GarbageCollectorMXBean] to monitor GC, to get name of garbage collections, numbers of collectors, times of collections and info about memories, … Each bean for per memory manager (old, young)

CODE: MainMXBeanDemo.java, E_BigObjMemoryAddrs
Run with: a) //default GC
b) -XX:+UseParallelGC
c) -XX:+UseConcMarkSweepGC

- **Jstat** – command line tool to monitor memory and GC activities, also can be used to profile remote JVM

CODE: E_BigObjMemoryAddrs

Run with: jstat –option <pid> <interval> <count>
//To get PID
>jps //to get Java apps. PID
// or ps -eaf | grep java command
E.g. >jstat -gc 10632 100 10

Try with: -XX:+UseParallelGC, and G1

See jstat descriptions [here]()

```
C:\Users\as892333>jps
25824
36576 Jps
30324 E_BigObjMemoryAddrs
27996 Launcher
33196 Eclipse

C:\Users\as892333>jstat -gcutil 30324
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     CGC    CGCT     GCT
  0.00  99.99  35.99  50.99  79.89  66.89    386   16.563     0    0.000    22    0.115   16.678

C:\Users\as892333>jstat -gccause 30324
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     CGC    CGCT     GCT    LGCC                 GCC
  0.00  99.99  37.99  52.47  80.33  66.89    512   22.065     0    0.000    28    0.148   22.213 Allocation Failure   Allocation Failure

C:\Users\as892333>jstat -gccapacity 30324
 NGCMN    NGCMX     NGC     S0C   S1C       EC      OGCMN      OGCMX       OGC         OC       MCMN     MCMX      MC     CCSMN    CCSMX     CCSC    YGC    FGC   CGC
   192.0 1107520.0 173376.0 17280.0 17280.0 138816.0       64.0  7209408.0  5808688.0  5808688.0      0.0 1056768.0   4864.0      0.0 1048576.0    512.0    612     0    34

C:\Users\as892333>jstat -gc 30324
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     CGC    CGCT     GCT
17280.0 17280.0  0.0   17279.1 138816.0   0.0    5808688.0  2404206.3  4864.0 3910.2 512.0  342.5     798   35.420   0      0.000  44      0.269   35.689

C:\Users\as892333>jstat -gc 30324 1000 10
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     CGC    CGCT     GCT
17280.0 17280.0 17280.0  0.0   138816.0 97152.2  5808688.0  2666411.2  4864.0 3910.2 512.0  342.5     873   39.133   0      0.000  48      0.289   39.422
17280.0 17280.0 17280.0  0.0   138816.0 69394.5  5808688.0  3097911.0  4864.0 3910.2 512.0  342.5     887   39.715   0      0.000  50      0.305   40.020
17280.0 17280.0 17280.0  0.0   138816.0 66618.7  5808688.0  2740607.6  4864.0 3910.2 512.0  342.5     901   40.334   0      0.000  50      0.305   40.639
17280.0 17280.0 17280.0  0.0   138816.0 127685.7 5808688.0  2967835.0  4864.0 3910.2 512.0  342.5     915   40.976   0      0.000  51      0.305   41.281
17280.0 17280.0 17280.0  0.0   138816.0   0.0    5808688.0  2706199.8  4864.0 3910.2 512.0  342.5     928   41.619   0      0.000  52      0.318   41.937
17280.0 17280.0  0.0   17279.1 138816.0   0.0    5808688.0  2474330.0  4864.0 3910.2 512.0  342.5     939   42.244   0      0.000  52      0.318   42.562
17280.0 17280.0  0.0   17279.1 138816.0 33309.4  5808688.0  3128731.5  4864.0 3910.2 512.0  342.5     952   42.831   0      0.000  53      0.319   43.150
17280.0 17280.0  0.0   17279.1 138816.0 38861.0  5808688.0  3108791.6  4864.0 3910.2 512.0  342.5     965   43.461   0      0.000  54      0.335   43.796
17280.0 17280.0 17280.0  0.0   138816.0 63842.8  5808688.0  2837968.4  4864.0 3910.2 512.0  342.5     979   44.070   0      0.000  54      0.335   44.404
17280.0 17280.0  0.0   17279.1 138816.0   0.0    5808688.0  3146669.0  4864.0 3910.2 512.0  342.5     991   44.702   0      0.000  56      0.345   45.047
```

Garbage Collection Tools

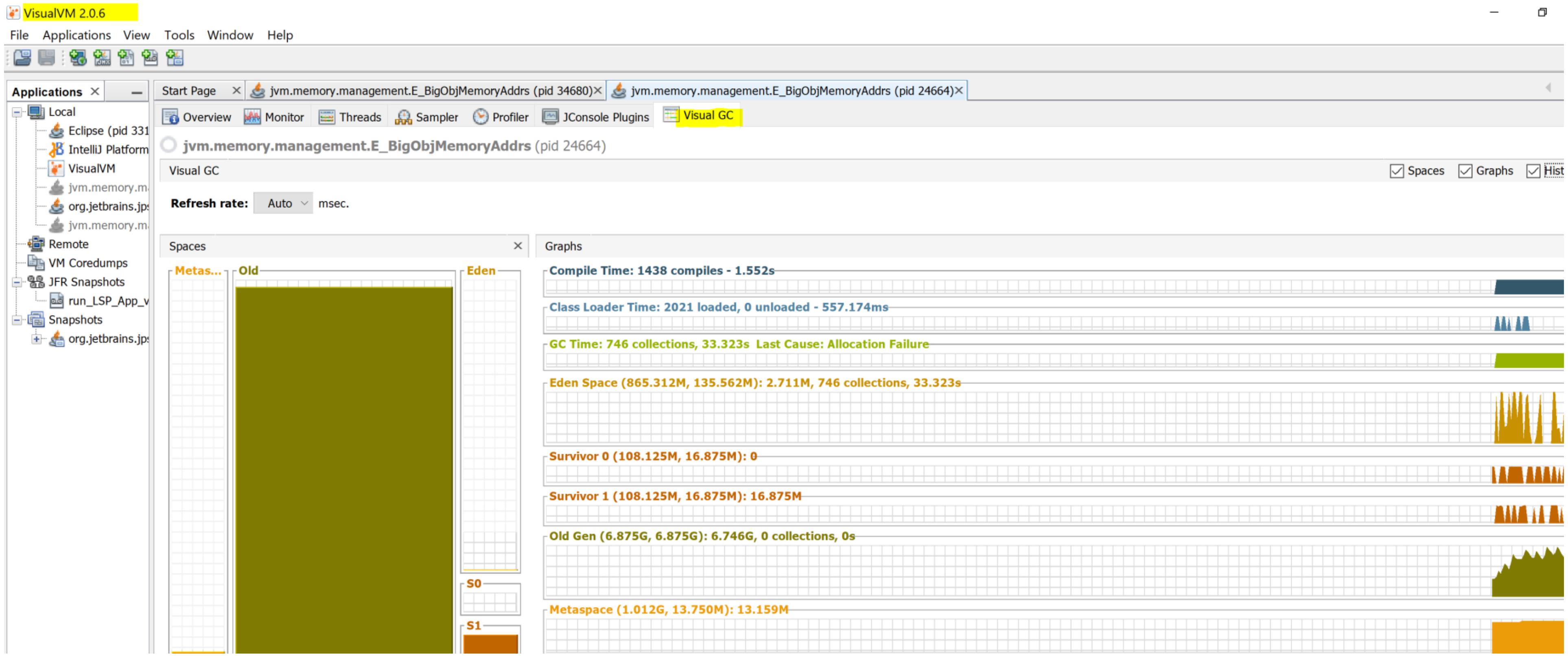
- Visualvm - VisualVM monitors and troubleshoots applications running on Java

Install Plugin for VisualGC

- visualgc – get more info in VM what is happening on GC

If you want to see memory and GC operations in GUI, then you can use jvisualvm tool. It was part of Java until Java 6, now need to download it separately. Once launched, you need to install **Visual GC** plugin from Tools -< Plugins option, as shown in below image.

> visualvm.exe --jdkhome C:\apps\Java\jdk-11

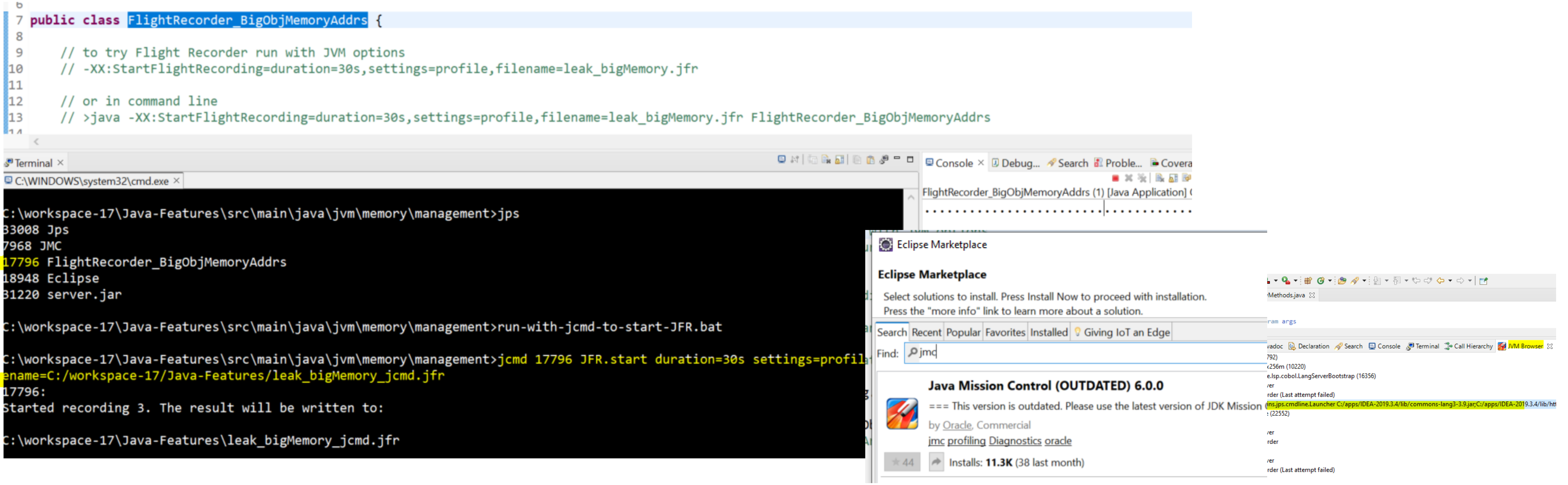


Telemetry Tools Oracle donated to open JDK

JCMD **-** utility is used to send diagnostic command requests to the JVM, where these requests are useful for controlling **Java Flight Recordings, troubleshoot, and diagnose** JVM and Java Applications. It must be used on the same machine where the JVM is running, and have the same effective user and group identifiers that were used to launch the JVM.

>jps //to get Java apps. PID //JMC can be downloaded as plugin or as a separate tool

>jcmd 11164 JFR.start duration=30s settings=profile filename=C:/workspace-JavaNew/Java-Features/leak.jfr //**creates dump**



Telemetry Tools Oracle donated to open JDK

- Java Flight Recorder (Telemetry events)

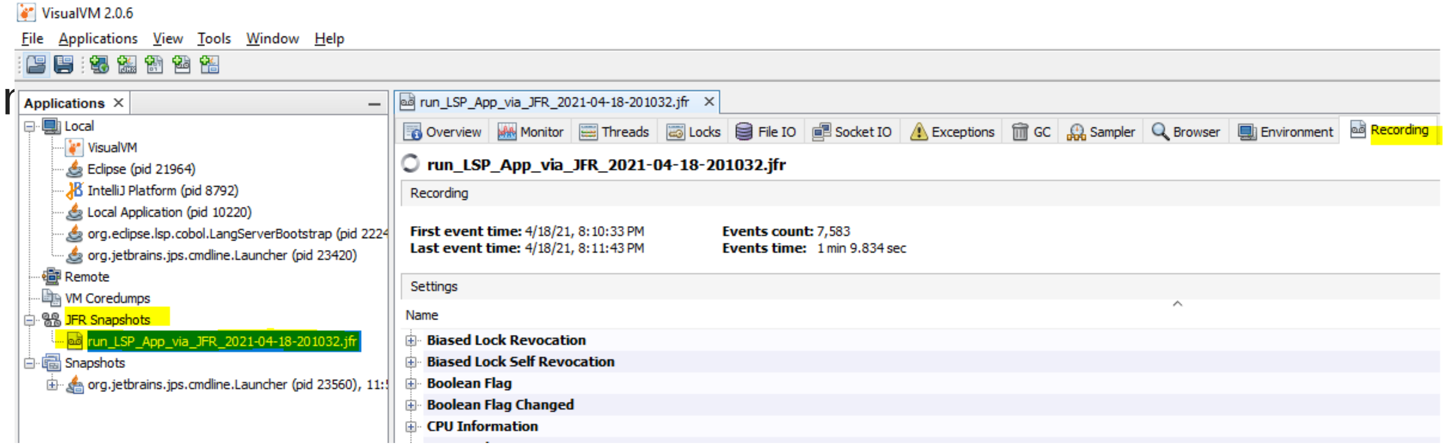
JFR is a tool for collecting diagnostic and profilir

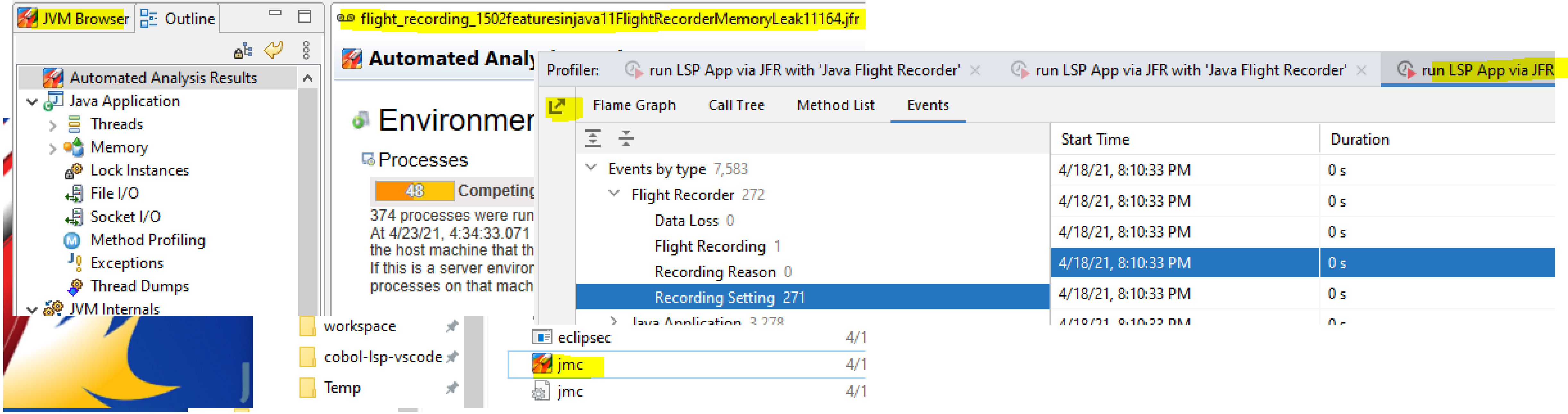
- visualVM can also see JFR snapshots
- [JMC - Java Mission Control]:

To see JFR output from **.jsr** file

- Supports Java Flight Recorder.
- Supports HotSpot VM, since JDK 7

Other Tools: jmap, jconsole, jstack, jcmd, ..





Java Reference Classes

Strong (Hard) References
This is a default type of reference – mostly we do not think referenced objects are garbage collected.
The object can't be garbage collected if it's reachable through any strong reference.

List<String> list = **new ArrayList**<>; // No GC, strong reference to it in the *list* variable
list = null; // Now, the *ArrayList* object can be collected because nothing holds a reference to it.

Strong -> Soft -> Weak -> Phantom

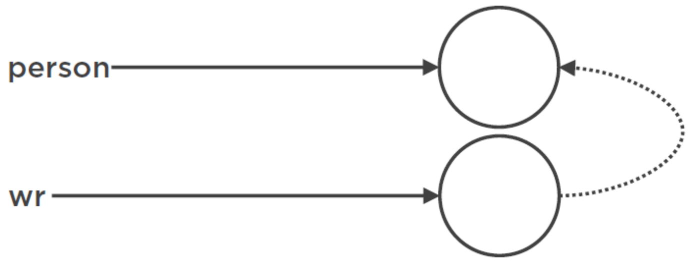
Object not GC if there is a **Strong** reference
Can be Garbage Collected if there is a **Soft, Weak** or **Phantom** reference

How GC acts on Soft | Weak | Phantom references

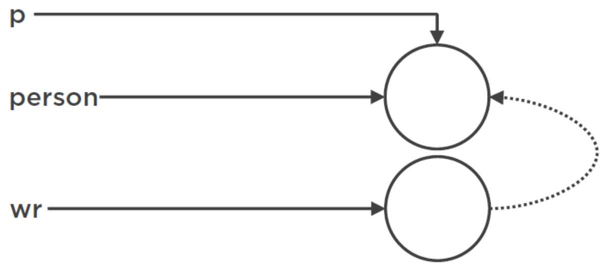
- **Soft** will be collected if there is **memory pressure**
- **Weak** will be collected **immediately**
- **Phantom** is different, can be used instead of Java 'finalize()' method
  - Cannot retrieve the object through a phantom reference, , strong reference is always NULL

Java Reference Classes

```
Person person = new Person(); //p is a strong reference
WeakReference<Person> wr = new WeakReference<Person>(person);// just wrap strong reference
```

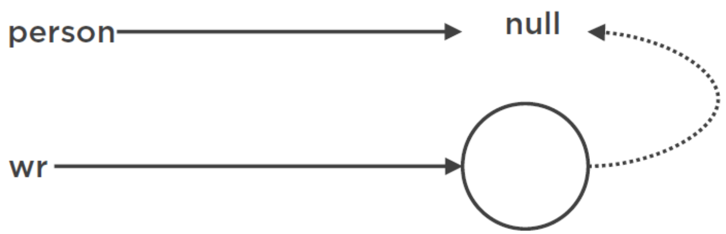



```
Person p = wr.get();
```

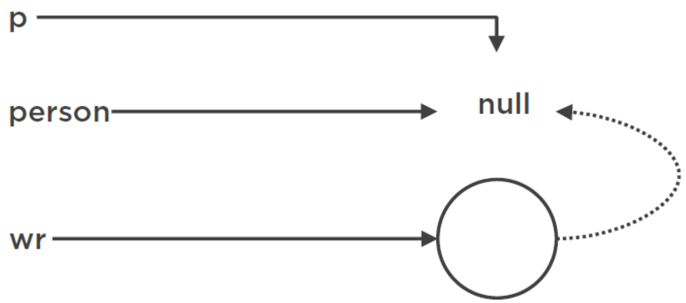


F_WeakReferences.java

```
person = null;
System.gc();
```




```
Person p = wr.get();
```



Usage of Java Reference Types

Soft will be collected if there is **memory pressure**

Soft Reference

- Hold a SoftReference to an object as well as a strong reference
- When strong reference is cleared soft is still available

- IT can be used **to make our code more resilient to errors connected to insufficient memory**. For example, we could create a memory-sensitive cache that automatically evicts objects when memory is scarce. We wouldn't need to manage the memory manually, as the garbage collector would do it for us.
- Yes can be used for caching – **but this is not real cache** we can not control it, no reference

Usage of Java Reference Types

- **Weak** will be collected **immediately**

Weak Reference

Associate meta data with another type and can be used with WeakHashMap.
Weak references are most often used **to create canonicalizing (**map only objects that can be reached**) mappings**. A great example is ***WeakHashMap*** *(to* create a short-living cache *)*, which **works like normal *HashMap***, but its keys are weakly referenced, and they are automatically removed when the referent is cleared. If we used a normal *HashMap,* the mere existence of the key in the map would prohibit it from being cleared by the garbage collector.

WeakHashMap

F_WeakHashMap.java

- Key is a weak reference to an object
- Store a weak reference to an object as a key
- Value is the object's 'meta data'

When object has no more strong references
- The key is released
- 'Meta data' goes away

Usage of Java Reference Types

- **Phantom** is least used one,
  - .get() always return NULL

ReferenceQueue

Pass a reference queue to constructor when creating the reference object
- Optional
- Except for **PhantomReference**

References types enqueued to ReferenceQueue
Useful when you want to associate **some cleanup mechanism** with an object

F_ReferenceQueue.java

When all strong references cleared - Reference object is added to the reference queue
ReferenceQueue has poll and remove methods
- poll returns immediately
- remove has a timeout
- Both remove object from the queue

F_PhantomReferences.java

Phantom Reference

Similarly to weak references, phantom references don't prohibit the garbage collector from enqueueing objects for being cleared. **Used to interact with GC – and works along with AutoCloseable objects**, and monitors object to make a cleanup before die, e.g. good fit to replace **finalize() .** Finalizers have issues – no predicted when it will be called, maybe not. Also expensive

Is It Possible to «Resurrect» an Object That Became Eligible for Garbage Collection?
When an object becomes eligible for garbage collection, the GC has to run the *finalize* method on it.
The *finalize* method is guaranteed to run only once, thus the GC flags the object as finalized and gives it a rest until the next cycle.
In the ***finalize* method** you can technically **"resurrect" an object,** for example, by assigning **it to a *static* field**.
The object would become **alive again and non-eligible for garbage collection**, so the GC would not collect it during the next cycle.
The object, however, would be marked as finalized, so when it would become eligible again, **the finalize method would not be called**. In essence, you can turn this "resurrection" trick only once for the lifetime of the object.
Beware that this ugly hack should be used only if you really know what you're doing — however, understanding this trick gives some insight into how the GC works.

Describe Strong, Weak, Soft and Phantom References and Their Role in Garbage Collection.
Much as memory is managed in Java, an engineer may need to perform as much optimization as possible to **minimize latency and maximize throughput**, in critical applications. Much as **it is impossible to explicitly control when garbage collection is triggered** in the JVM, **it is possible to influence how it occurs as regards the objects we have created.**
Java provides us with reference objects to control the relationship between the objects we create and the garbage collector. By default, every object we create in a Java program is strongly referenced by a variable:

https://www.baeldung.com/java-memory-management-interview-questions

THANK YOU

References

https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1_gc_tuning.html
https://www.baeldung.com/jvm-experimental-garbage-collectors
https://www.freecodecamp.org/news/garbage-collection-in-java-what-is-gc-and-how-it-works-in-the-jvm/#:~:text=In%20Java%2C%20garbage%20collection%20happens,programs%20perform%20automatic%20memory%20management.
https://www.digitalocean.com/community/tutorials/java-jvm-memory-model-memory-management-in-java
https://www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-java
https://docs.oracle.com/javase/7/docs/technotes/guides/vm/G1.html
http://tutorials.jenkov.com/java-concurrency/java-memory-model.html
https://www.youtube.com/watch?v=WU_mqNBEacw
https://shipilev.net/
https://www.youtube.com/watch?v=Gee7QfoY8ys https://www.youtube.com/watch?v=WU_mqNBEacw