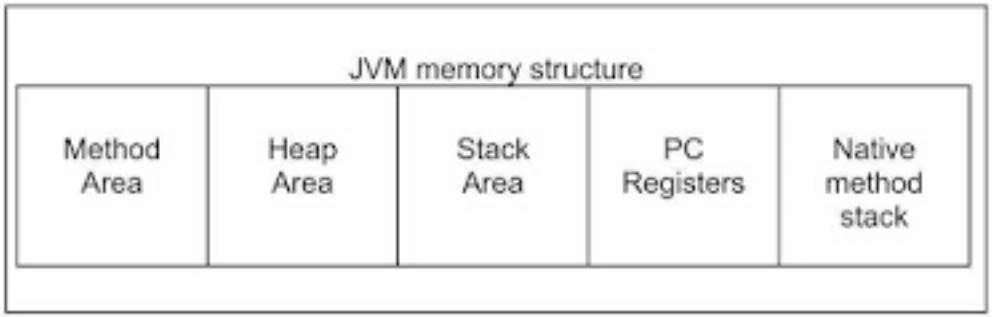
Java Memory Model

Shalini Mittal
Corporate Trainer

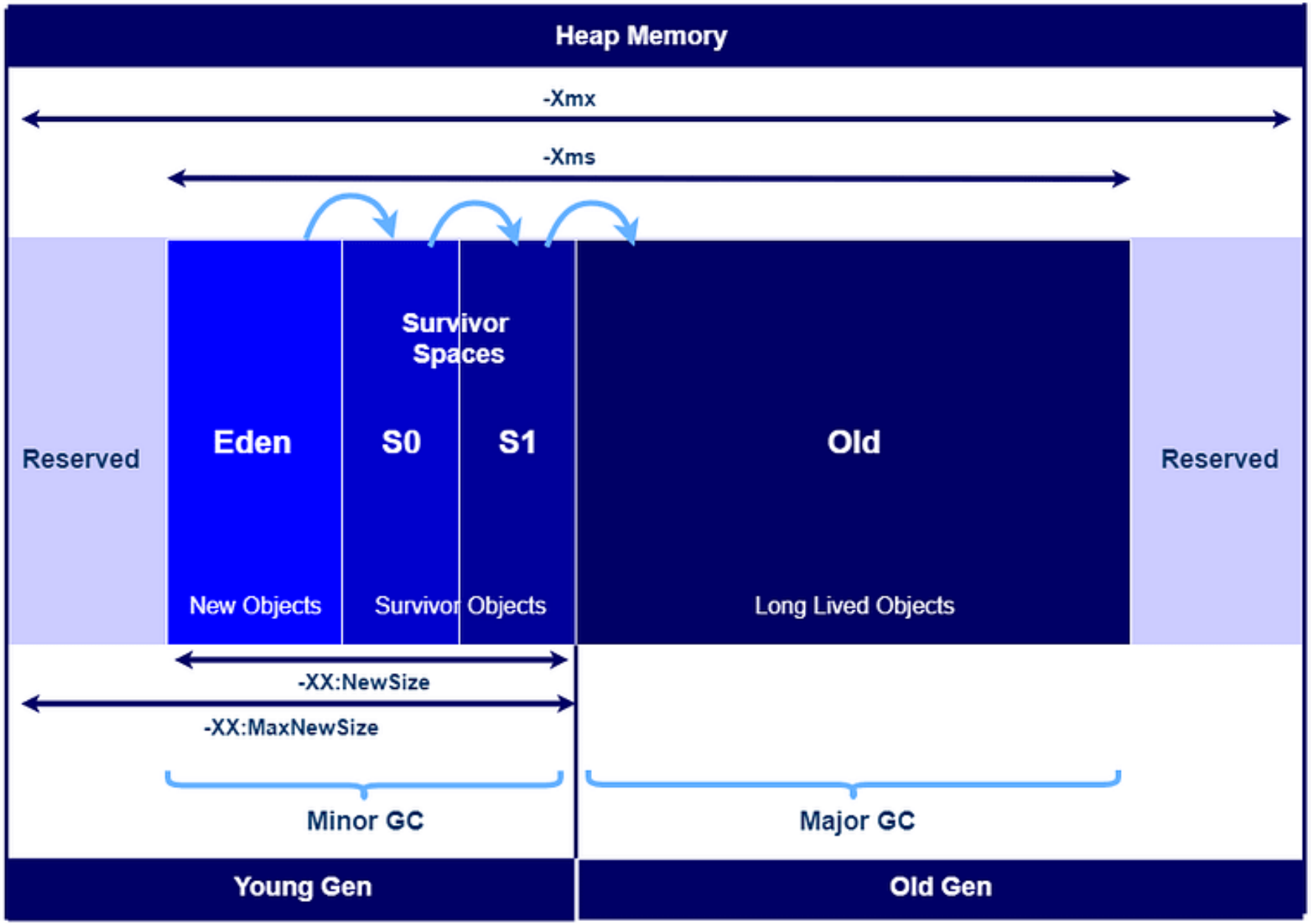
Java Memory Model

- Three primary components of Java's memory model are heap, stack and metaspace.
- The Java Memory Model (JMM) defines how threads interact with memory in the Java Virtual Machine, particularly in multithreaded environments.
- It establishes rules around visibility, ordering, and atomicity of variable access, ensuring that concurrent programs behave consistently across different hardware and JVM implementations.
- The Java Virtual Machine (JVM) divides its memory into several logical runtime data areas, each serving a specific role during program execution. This structured memory model ensures isolation between different types of data, supports multithreading, and enables features like automatic garbage collection.
- At runtime, the JVM creates a set of runtime data areas. These include:
- Heap Memory
- Stack Memory
- Method Area
- Program Counter (PC) Register
- Native Method Stack



Heap

- Primary area for storing Java objects. Whenever an object is created using the new keyword in Java, memory for that object is allocated from the heap. This is also where the Garbage Collector operates, reclaiming memory used by objects that are no longer needed, which helps prevent memory leaks and excessive memory usage. This is the shared memory accessible to all threads, where objects are stored.
- **Structure of the Heap:** The Java heap is subdivided into two regions and layout is called as **Generational Heap Model**
    - Young Generation
    - Old Generation



JVM Heap Memory
(Image: PlatformEngineer.com)

Young Generation

- **Young Generation:**
  - This is where all new objects are allocated. It is optimized for fast allocation and frequent garbage collection. This region is collected often using Minor Garbage Collections (Minor GCs), which are typically fast and efficient.
  - The Young Generation is further divided into one '**Eden' space** and **two 'Survivor' spaces (S0 and S1)**. Objects initially reside in Eden and move to a Survivor space if they remain alive after a garbage collection event.
- Minor GCs are **stop-the-world events**, meaning all application threads are paused briefly while the garbage collection is performed.
- Proper tuning of the Young Generation can help reduce promotion rates and delay costly collections in the Old Generation. For example, you can adjust:
  - The size of the Young Generation with -Xmn
  - The Eden-to-Survivor space ratio using -XX:SurvivorRatio
  - The promotion age threshold with -XX:MaxTenuringThreshold

- These parameters are particularly useful for applications with high allocation rates, such as REST APIs, streaming pipelines, or real-time event processing systems.

Old Generation

- The Old Generation, also known as the Tenured Generation(controlled by the JVM flag -XX:MaxTenuringThreshold), is designed to hold long-lived objects, those that have survived multiple Minor GCs.
- Examples include:
    - Persistent application-level caches
    - Large object graphs such as sessions or user data
    - Static or shared data structures that are retained across requests
- Garbage collection in this region is referred to as a Major GC, and when both generations are collected together, the process is known as a Full GC.
- Collections in the Old Generation are more expensive and typically involve:
    - A full stop-the-world pause
    - Tracing all reachable objects starting from the GC roots
    - Compacting memory to eliminate fragmentation
- To control the size and behavior of the Old Generation, you can adjust:
    - The total heap size using -Xmx (maximum) and -Xms (initial)
    - The size of the Young Generation using -Xmn, which affects how much memory is left for the Old Generation
    - The ratio between the two using -XX:NewRatio

-Xms512m -Xmx2g -Xmn512m -XX:NewRatio=3

Stack

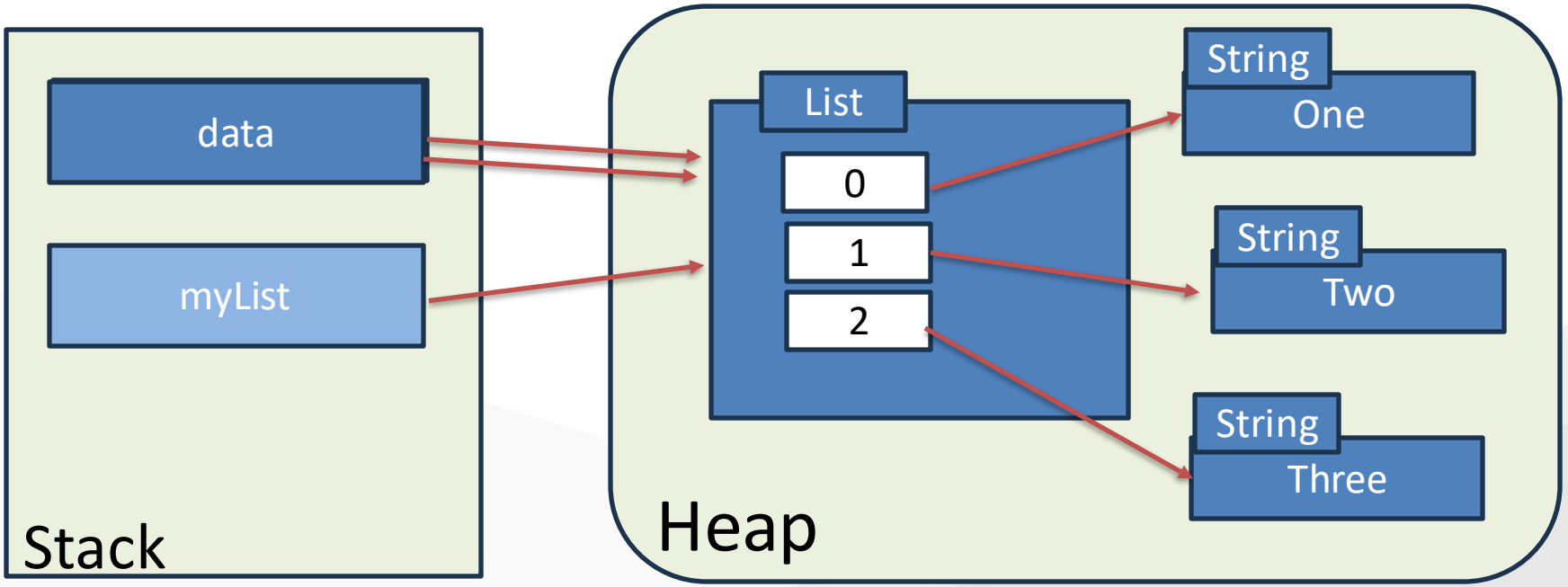
- Each thread in a Java application has its own stack, which is used for storing short-lived variables and method call information.
- The stack is smaller in size compared to the heap but is crucial for handling method invocations and storing local variables and intermediate outcomes of expressions
- It is a structured, fast-access memory region that tracks the flow of method calls and stores data relevant to individual method invocations.
- When a thread is created, the JVM allocates a new Java stack for that thread. This stack consists of a series of stack frames, each representing a single method invocation.
- Each stack frame contains:
    - Local variable array: Holds all method parameters and local variables declared within the method.
    - Operand stack: Used internally by the JVM to evaluate expressions and store intermediate computations.
    - Return value slot: Stores the result of the method call, if any, before passing it back to the calling method.
    - Reference to the runtime constant pool: Allows the method to resolve field names, method names, and literals.

Stack Size and Overflow

- Each thread's stack is limited in size, which can be configured using the -Xss JVM option -Xss1m
- StackOverflowError typically occurs in scenarios like:
  - Deep recursion with no base case
  - Improper termination conditions in recursive algorithms
  - Method chaining or complex functional-style programming without tail-call optimization
- Situations where understanding stack behavior is important:
  - **Debugging runtime errors**: Stack traces printed during exceptions show the call stack at the point of failure, which directly maps to the frames on the JVM stack.
  - **Avoiding StackOverflowError**: Writing recursive algorithms requires awareness of stack depth, especially when processing large data sets or graphs.
  - **Configuring high-concurrency systems:** In applications that create thousands of threads (e.g., web servers, microservices), tuning stack size with -Xss may be necessary to avoid exhausting physical memory.

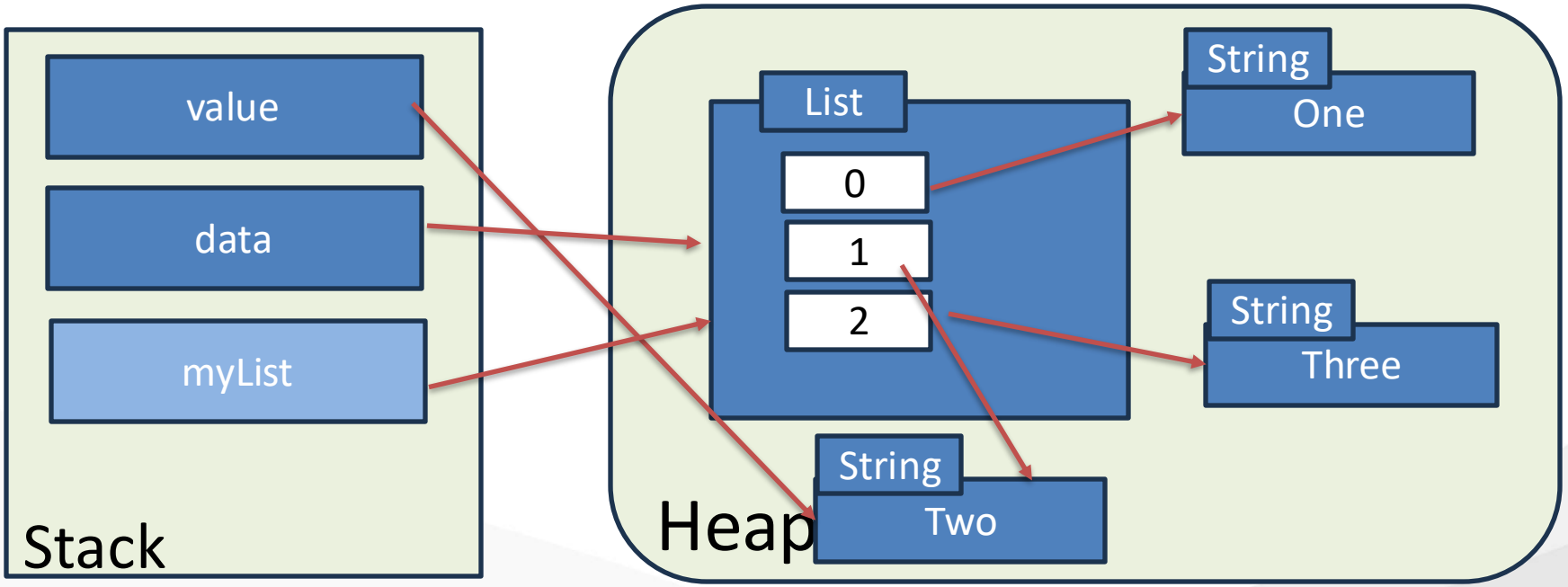
Sample Program

```
public static void main(String[] args) {
    List<String> myList = new ArrayList<String>();
    myList.add("One");
    myList.add("Two");
    myList.add("Three");
    printList(myList);
}
public static void printList(List<String> data) {
    System.out.println(data);
}
```



Sample Program

```
public static void printList(List<String> data) {
    String value = data.get(1);
    data.add("Four");
    System.out.println(value);
}
```



Memorytest -> explore

Memory Leak

- A "Java memory leak" is when objects that are no longer needed by the application still hold onto memory because they are referenced by live objects, preventing the garbage collector from freeing that memory.
- Common causes include unclosed resources (like files, database connections, or network sockets), incorrect use of static variables, and failure to shut down thread pools properly. Symptoms include increasing memory usage, sluggish performance, frequent full garbage collection cycles, and ultimately an OutOfMemoryError.

Memory Leak How?

1.1. Objects are still referenced:
Even if you think an object is no longer needed, if there's a persistent reference to it somewhere in the application, the garbage collector won't consider it eligible for collection.

2.2. Resource leaks:
Resources like file streams, database connections, and sockets must be explicitly closed after use. If they're not, the JVM keeps them alive, consuming memory and holding onto underlying OS resources.

3.3. Thread leaks:
Failing to shut down thread pools or not terminating threads properly can lead to a growing number of threads consuming memory and CPU resources.

4.4. Static references:
Static variables live for the entire application's lifetime, so if they hold references to large objects or collections, these objects won't be garbage collected even when no longer useful.

Signs of Memory Leak

- Increasing memory usage: The application's RAM usage continuously rises and doesn't stabilize.
- Sluggish performance: As memory fills up, the garbage collector has to work harder, leading to slowdowns.
- Frequent full GC events: The garbage collector is constantly running full cycles, indicating it's struggling to free memory.
- OutOfMemoryError: The most severe symptom, where the JVM runs out of memory and crashes.

Fix memory Leaks

- Use try-with-resources:
- For resources like streams and connections, use the try-with-resources statement to ensure they are automatically closed, even if errors occur.
- Properly shut down components:
- Shut down thread pools and other components when they are no longer needed.
- Be mindful of static variables:
- Avoid using static variables to hold large datasets or long-lived references unless absolutely necessary.
- Use profiling tools:
- Tools like JProfiler or YourKit can help analyze heap dumps and identify memory leaks.
- Use heap dump analyzers:
- These tools can show you which objects are consuming memory and what references are keeping them alive.

SoftLeaks Project with and without –Xmx option

- Use VisualVM / jstat/ jconsole to see the heap size.

jstat

- *Jps – to get process id of running application*
- *jstat –gc <JAVA PID> 1000*

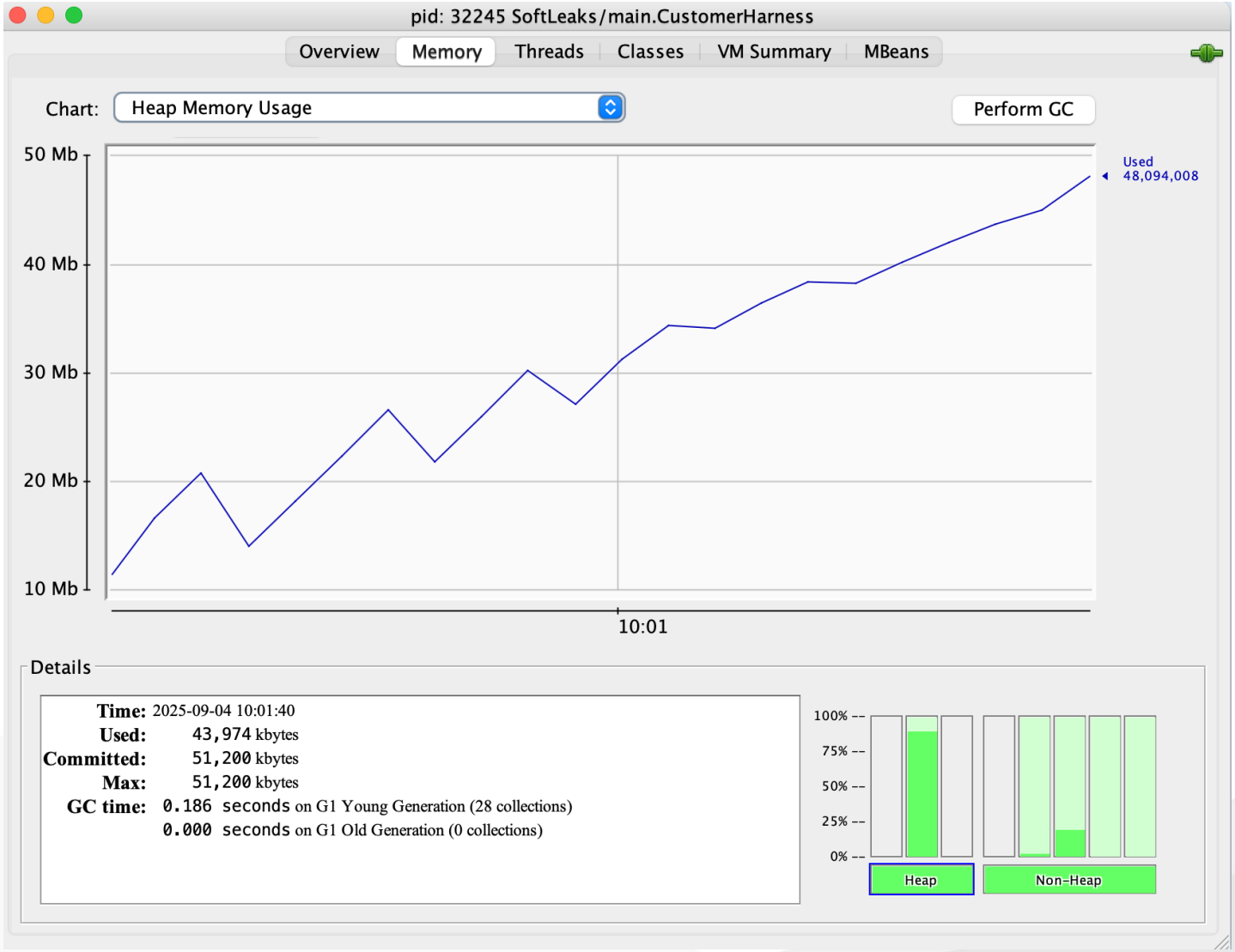
S0C	Current survivor space 0 capacity (KB)
S1C	Current survivor space 1 capacity (KB)
S0U	Survivor space 0 utilization (KB)
S1U	Survivor space 1 utilization (KB)
EC	Current eden space capacity (KB)
EU	Eden space utilization (KB)
OC	Current old space capacity (KB)
OU	Old space utilization (KB)
MC	Metasapce capacity (KB)
MU	Metaspace utilization (KB)
CCSC	Compressed class space capacity (KB)
CCSU	Compressed class space used (KB)
YGC	Number of young generation garbage collection events
YGCT	Young generation garbage collection time
FGC	Number of full GC events
FGCT	Full garbage collection time
GCT	Total garbage collection time

Jconsole

- The JConsole graphical user interface is a monitoring tool that complies to the Java Management Extensions (JMX) specification. JConsole uses the extensive instrumentation of the Java Virtual Machine (Java VM) to provide information about the performance and resource consumption of applications running on the Java platform.
- The jconsole executable can be found in JDK_HOME/bin, where JDK_HOME is the directory in which the Java Development Kit (JDK) is installed.
- Not recommended for production environment as it consumes resources.
- https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr009.html
- Jconsole -> start the process
- Run SoftLeaks Application
- Create new connection in jconsole with this Main process and monitor the application

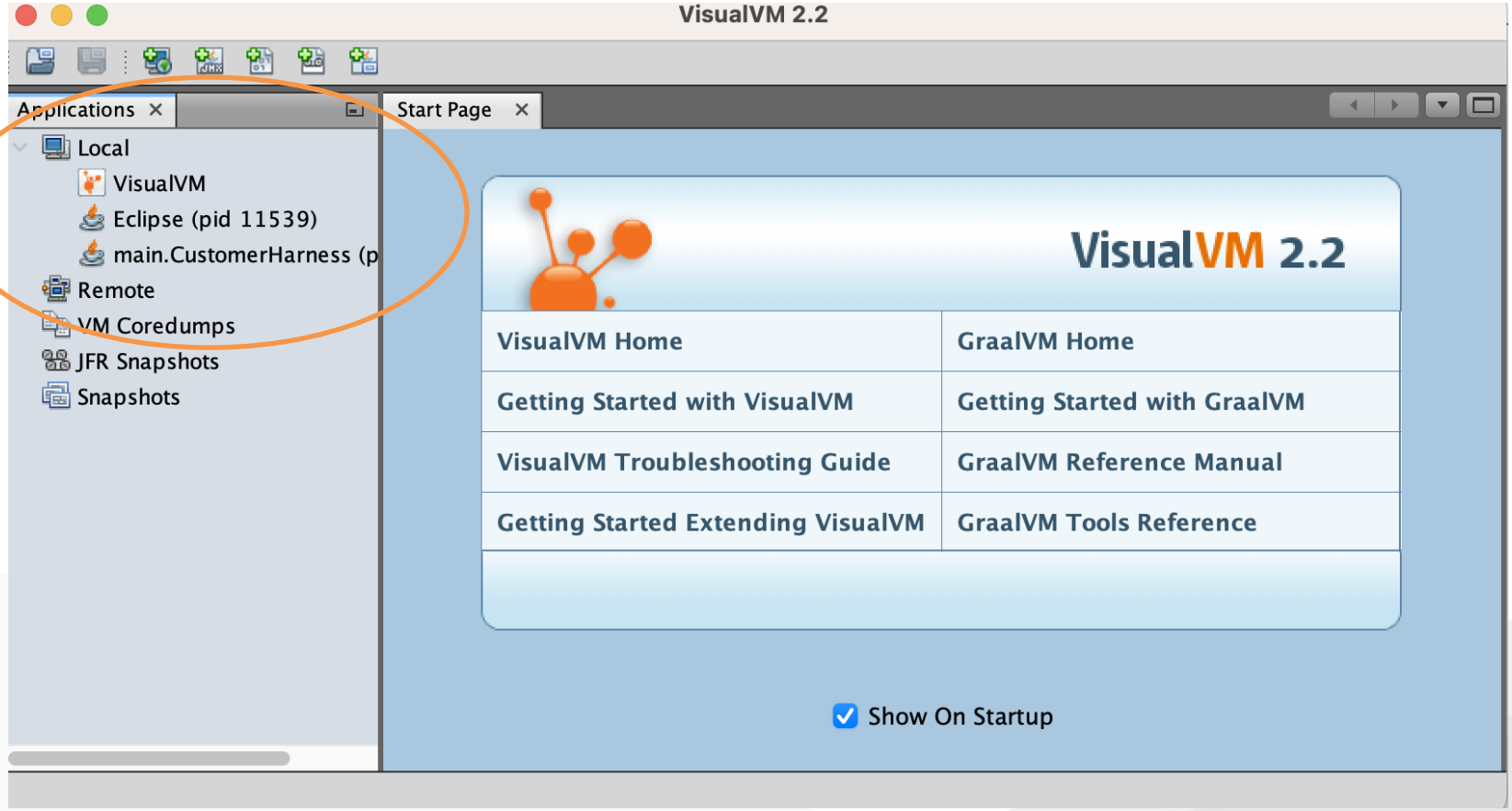
https://medium.com/@abhimanyumaxfort/monitoring-java-applications-in-real-time-with-jconsole-a3953650ff8b

JConsole Screenshot



VisualVM

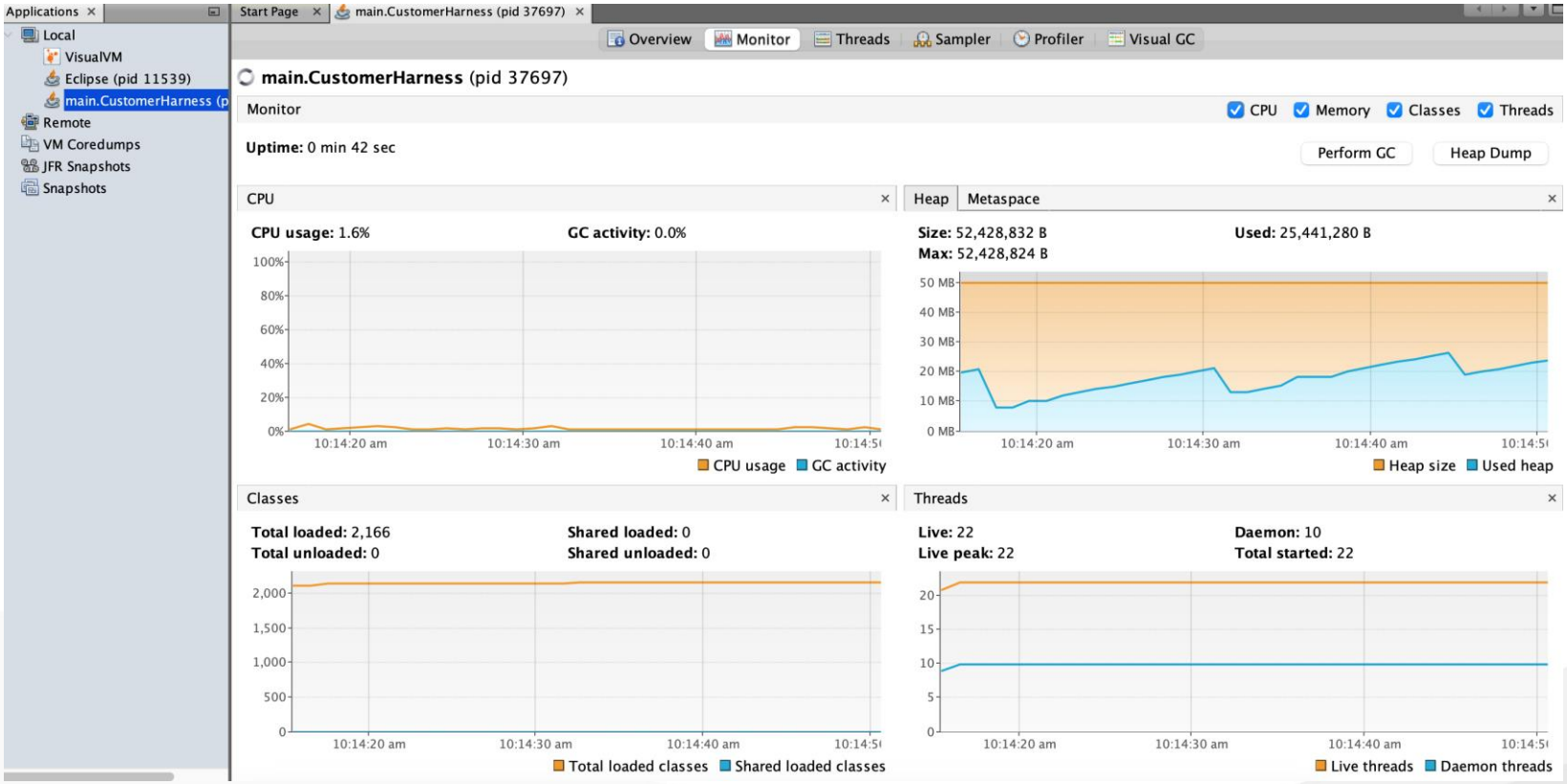
- Download VisualVM : https://visualvm.github.io/download.html
- Install and open should see screen as below and left section shows the current applications running.



https://visualvm.github.io/gettingstarted.html

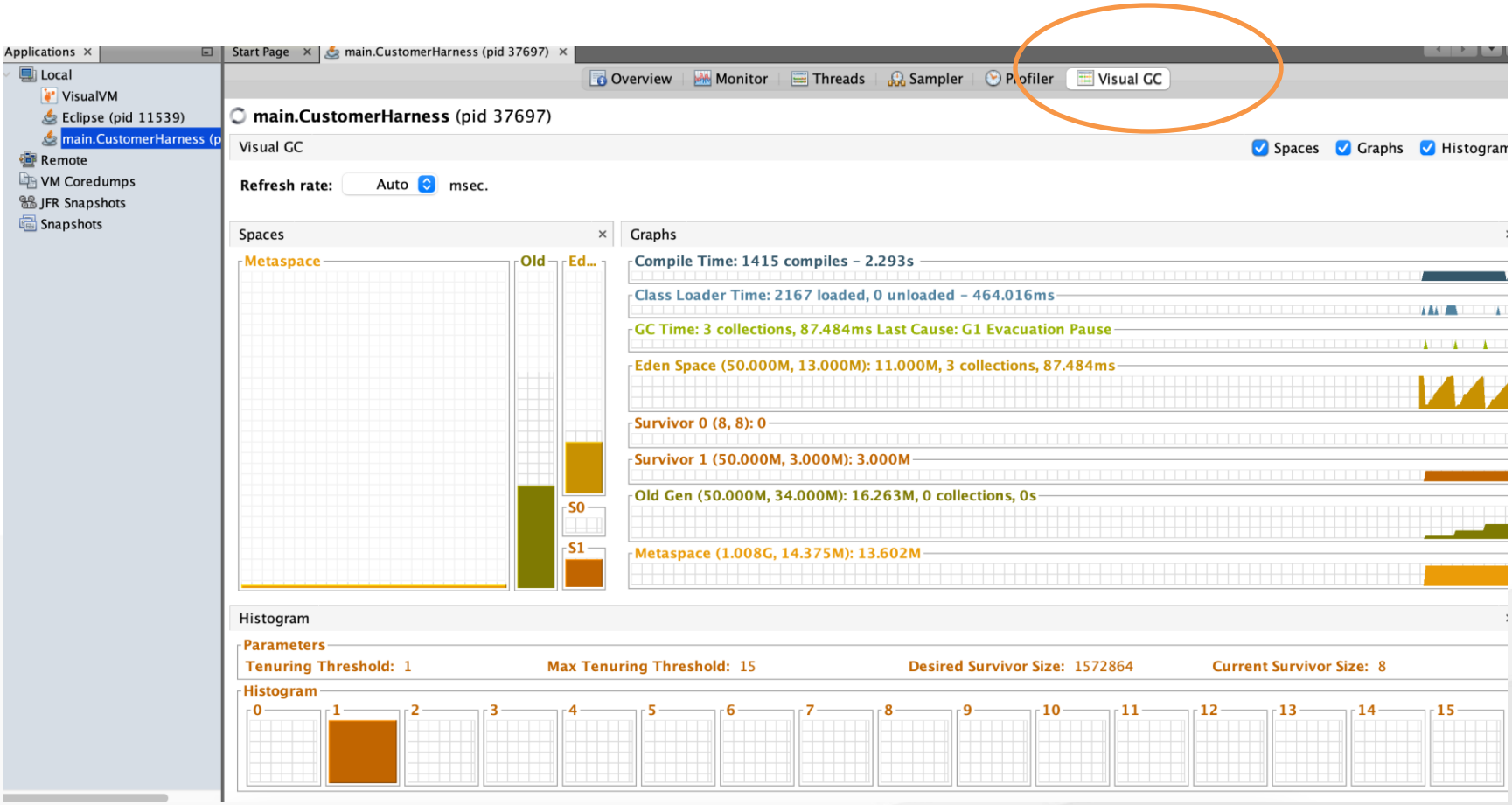
VisualVM Screen

- Once click on application and explore different tabs:



Install VisualGC Plugin

- Go to Tools -> Plugin -> Click Available Plugins -> Search for VisualGC -> Install
- You will VisualGC tab within your VisualVM as below

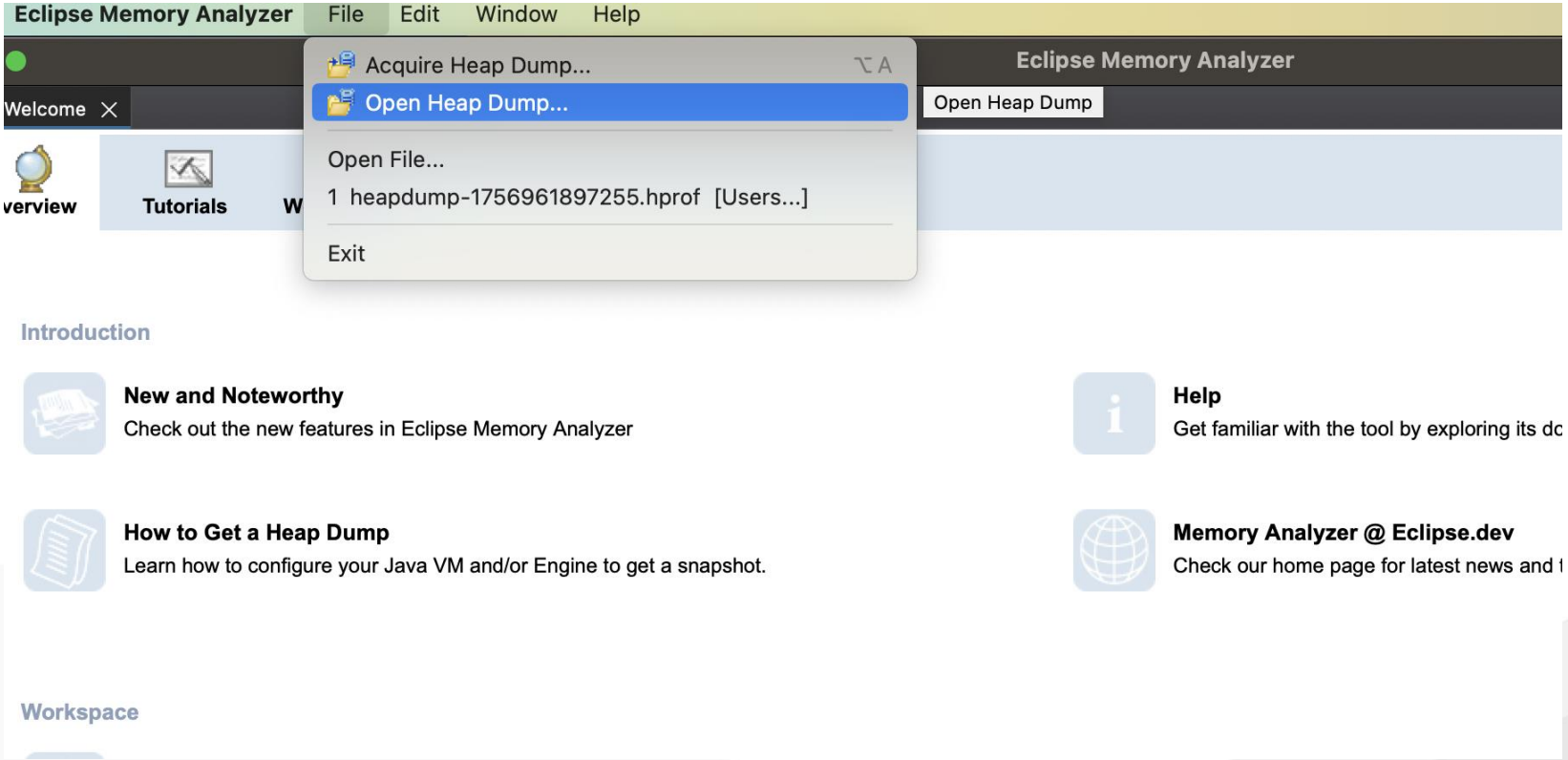


Heap Dump

- Run SoftLeaks setting VM arg: –Xmx50m
- View the application within VisualVm Monitor tab and once the heap is nearly full click on Heap Dump on the right.
- In windows it provides the location where heap dump is saved.
- In MAC it by default provides information but to save the results of Heap Dump by right click on file within Applications tab -> Save As and save it with .hprof extension.
- Alternatively can use JVM argument to generate heap dump on out of memory error -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=filepath
- If HeapDumpPath not used the default location is your project
- To see the content of .hprof use Eclipse MAT.
- Download : https://eclipse.dev/mat/download/ or install via Install software within ecliplse

Eclipse Memory Analyzer Tool

- Open MAT installed and screen looks as below:
- Click File -> Open Heap Dump, choose the .hprof file saved from VisualVM and it provides with options.



Eclipse Memory Analyzer Options

- Choose Leak Suspects Memory and visualize the issue with your code.

Calculate JVM Memory Consumption

- Many programmers figure out the maximum heap value for their application's JVM correctly but discover that the JVM is using even more memory.
- The value of the -Xmx parameter specifies the maximum size of the Java heap, but that is not the only memory consumed by the JVM.
- Permanent Generation (the name prior to JDK 8) or Metaspace (the name from JDK 8 onward), the CodeCache, the native C++ heap used by other JVM internals, space for the thread stacks, direct byte buffers, garbage collection overhead, and other things are counted as part of the JVM's memory consumption.
- You can calculate the memory used by a JVM process as follows:

  JVM memory = Heap memory+ Metaspace + CodeCache + (ThreadStackSize * Number of Threads) + DirectByteBuffers + Jvm-native Copy snippet

- Therefore, JVM memory usage can be more than the -Xmx value under peak business load.

Escape Reference

- Refers to a situation where a reference to an object, particularly a mutable one, is made accessible outside the intended scope or encapsulation, potentially allowing unintended modification of the object's internal state.
- This can lead to bugs, especially in concurrent environments.
- Common Scenarios of Escaping References:
    - Returning a mutable internal object:
      If a method returns a direct reference to a mutable object (like a List or Map) that is a private field of the class, external code can modify that object, bypassing the class's intended control over its state.
    - Publishing this during construction:
      Starting a thread in a constructor or registering this as a listener before the object is fully constructed can expose a partially initialized object to other threads, leading to race conditions and inconsistent states.
    - Storing mutable arguments directly:
      If a constructor or setter method directly stores a mutable object passed as an argument without making a defensive copy, external changes to the original argument will affect the internal state of the class.
- EscapingReferences.java
- String Intern : ExploringStrings.java

Avoid Escape Reference

- **Iterable Interface:**
  Implements Iterable interface and override iterator method, getter method can be removed from the class so that we will not return the object reference.
- **Duplicating collection:**
  Creating a new collection of the actual collection while returning from a getter method could solve this issue. In regards to performance, there is a minimal overhead in copying the references for a new copy of object/collection. But this is negligible.
- **Passing Immutable collection:**
  We can create immutable collection out of original collection and pass it to the caller. There are two ways that we can create immutable collections.
  - Collections.unmodifiablexxx(<original collection>)
  - In Java versions >=10, we can use <collection>.copyOf(<original collection>).
- **Duplicating Objects:**
  When passing an object like we have discussed for collections above, always return a copy of an object instead of passing reference to the original object.
- **Using an interface that has only get methods:**
  We can refactor our class to create a new interface with only get methods and return the new object with this interface reference. By doing this we are limiting the caller to only get methods instead of set methods.
- BookCatalog - Exercise

String Pool Statistics

- See the example ExploringStrings for intern strings.
- To print statistics about the Java String pool (also known as the StringTable), the PrintStringTableStatistics JVM flag is utilized.
- When the application terminates, the JVM will print detailed statistics about the StringTable to the standard output. This output typically includes:
    - Number of buckets: The total number of hash buckets in the StringTable.
    - Average bucket size: The average number of String entries per bucket.
    - Variance of bucket size: A measure of the spread of entries across buckets.
    - Std. dev. of bucket size: The standard deviation of the bucket sizes.
    - Maximum bucket size: The largest number of entries found in a single bucket.

String Pool

- Comment the code in main and run below command for java 17:
  java -XX:+PrintStringTableStatistics main.Main

```
(base) Manishs-MacBook-Pro:bin Shalini$ java -XX:+PrintStringTableStatistics main.Main
SymbolTable statistics:
Number of buckets       :     32768 =    262144 bytes, each 8
Number of entries       :        21 =       336 bytes, each 16
Number of literals      :        21 =       896 bytes, avg  42.000
Total footprint         :           =    263376 bytes
Average bucket size     :     0.001
Variance of bucket size :     0.001
Std. dev. of bucket size:     0.025
Maximum bucket size     :         1
StringTable statistics:
Number of buckets       :     65536 =    524288 bytes, each 8
Number of entries       :         9 =       144 bytes, each 16
Number of literals      :         9 =       576 bytes, avg  64.000
Total footprint         :           =    525008 bytes
Average bucket size     :     0.000
Variance of bucket size :     0.000
Std. dev. of bucket size:     0.012
Maximum bucket size     :         1
```

- Below is with java 11
  export JAVA_HOME=$(/usr/libexec/java_home -v 11)
  java -XX:+PrintStringTableStatistics main.Main [Output Varies]

String Pool Tuning

- Add a loop for creating string:

```
(base) Manishs-MacBook-Pro:bin Shalini$ java -XX:+PrintStringTableStatistics -XX:StringTableSize=4194307 main.Main
Elapsed time was 4329 ms.
SymbolTable statistics:
Number of buckets       :     20011 =    160088 bytes, each 8
Number of entries       :     20749 =    497976 bytes, each 24
Number of literals      :     20749 =    782416 bytes, avg  37.709
Total footprint         :           =   1440480 bytes
Average bucket size     :     1.037
Variance of bucket size :     1.047
Std. dev. of bucket size:     1.023
Maximum bucket size     :         9
StringTable statistics:
Number of buckets       :   8388608 =  67108864 bytes, each 8
Number of entries       :  10001743 = 160027888 bytes, each 16
Number of literals      :  10001743 = 480114824 bytes, avg  48.003
Total footprsize_t      :           = 707251576 bytes
Average bucket size     :     1.192
Variance of bucket size :     1.383
Std. dev. of bucket size:     1.176
Maximum bucket size     :         9
You have new mail in /var/mail/Shalini
(base) Manishs-MacBook-Pro:bin Shalini$ java -XX:+PrintStringTableStatistics main.Main
Elapsed time was 12239 ms.
SymbolTable statistics:
Number of buckets       :     20011 =    160088 bytes, each 8
Number of entries       :     20749 =    497976 bytes, each 24
Number of literals      :     20749 =    782416 bytes, avg  37.709
Total footprint         :           =   1440480 bytes
Average bucket size     :     1.037
Variance of bucket size :     1.047
Std. dev. of bucket size:     1.023
Maximum bucket size     :         9
StringTable statistics:
Number of buckets       :   4194304 =  33554432 bytes, each 8
Number of entries       :  10001746 = 160027936 bytes, each 16
Number of literals      :  10001746 = 480115008 bytes, avg  48.003
Total footprsize_t      :           = 673697376 bytes
Average bucket size     :     2.385
Variance of bucket size :     3.864
Std. dev. of bucket size:     1.966
Maximum bucket size     :        13
```

Method Area (Metaspace in Java 8+)

- Non-heap memory area that came into existence with Java 8, replacing the Permanent Generation.
- Used to store metadata such as class definitions, method data, and field data.
- What the Method Area Stores
    - When a class or interface is loaded by the JVM, its definition is parsed and the following information is stored in the method area:
    - Class structure metadata, including class names, superclasses, implemented interfaces, and modifiers (public, final, etc.)
    - Runtime constant pool, which contains literal values and symbolic references used by the class
    - Static variables, which are class-level variables shared across all instances
    - Field and method information, including method signatures, access modifiers, and bytecode instructions
    - Constructor code, including initialization routines for object creation
    - Type information used for method resolution and dispatching

PermGen

- In JVM versions prior to Java 8, the method area was physically implemented in a fixed-size memory region called the Permanent Generation (PermGen).
- PermGen resided in the heap and had to be explicitly sized using JVM flags like:

  -XX:PermSize=128m -XX:MaxPermSize=256m
- The fixed nature of PermGen introduced several challenges:
  - Class metadata could exceed PermGen space, leading to java.lang.OutOfMemoryError: PermGen space, especially in applications that dynamically load many classes (e.g., application servers, modular frameworks).
  - Memory management for PermGen was limited and difficult to tune correctly.
  - It could not grow beyond its configured maximum, regardless of available system memory.

Metaspace

- Unlike the heap, Metaspace is allocated out of the native memory, and its size is not fixed but can increase dynamically (subject to system limitations)., which helps prevent the OutOfMemoryErrors that were possible with the Permanent Generation.
- To control Metaspace usage, the JVM provides the following flags: -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=512m
- MetaspaceSize defines the initial size, which affects when the first GC of class metadata will be triggered.
- MaxMetaspaceSize limits how large Metaspace can grow. If omitted, Metaspace will grow until it exhausts native memory.
- Applications that dynamically load and unload classes — such as servlet containers (Tomcat, Jetty), OSGi platforms, or scripting engines — should monitor Metaspace usage closely.
- Tools like VisualVM, JFR (Java Flight Recorder), or jcmd can be used to track Metaspace growth and identify memory leaks related to class loaders.

Class Unloading and GC

- Metaspace memory is eligible for collection when classes are unloaded, but this only happens if:
    - The class loader that loaded the class is no longer reachable.
    - The garbage collector is invoked and identifies the class loader as unreachable.
    - The JVM supports class unloading (some collectors may restrict this).
- Uncollected class loaders can lead to Metaspace leaks, especially in environments where classes are reloaded frequently (e.g., hot deployment in development servers).
- To mitigate this, JVM options like -XX:+ClassUnloading (enabled by default in modern JVMs) can help enable or improve class unloading behavior.
- For older versions of the JVM that use the CMS garbage collector, the flag -XX:+CMSClassUnloadingEnabled was used to enable class unloading. However, CMS was deprecated in Java 9 and removed in Java 14, so this flag is only applicable to legacy JVMs.

Comparison

Feature	Heap	Stack	Metaspace
Purpose	Stores objects, instance variables	Holds method calls, local variables	Stores class metadata and static data
Managed by	JVM Garbage Collector	JVM, automatically	JVM, partially managed by GC
Scope	Global (within JVM process)	Limited to method/block scope	Global
Lifetime	Until no references remain	Until method completes	Until class unloading
Speed	Slower	Faster	Moderate
Error	OutOfMemoryError	StackOverflowError	OutOfMemoryError
Accessibility	Shared across threads	Each thread has its own stack	Shared across threads

Program Counter

- It plays a critical role in tracking the flow of execution for Java applications.
- It is concerned with instruction-level control: keeping track of which bytecode instruction a thread should execute next.
- Each thread in the JVM has its own private Program Counter register.
- The Program Counter register contains the address of the next instruction to be executed in the current thread's method. This allows the JVM to resume execution from the correct point after:
    - A method call
    - A branch (e.g., if/else, loop)
    - An exception handler
    - A thread context switch
- Java developers do not interact with directly, but visible in several situations:
    - Stack traces: When an exception occurs, the JVM prints a stack trace showing the method calls and line numbers where the exception occurred. These line numbers are derived from the PC register at the time of the crash.
    - Debugging tools: Java debuggers and profilers internally rely on the PC register to determine where execution is paused, especially during step-through debugging or when setting breakpoints.
    - Thread state inspection: Tools like jstack use the PC register to show what each thread was doing at a given moment.

Native Method Stack

- It is a dedicated memory region in the JVM that supports the execution of native methods, methods written in languages other than Java, such as C or C++.
- These methods are typically called through the Java Native Interface (JNI), which acts as a bridge between the JVM and native libraries.
- Each thread in the JVM has its own native method stack, separate from the standard Java stack used for executing bytecode.
- the native method stack deals with:
    - Native language function calls
    - Operating system-level data structures
    - Registers and pointers specific to compiled native code

Heap Size

- java -XX:MaxHeapSize=100m -XX:+PrintStringTableStatistics -XX:StringTableSize=4194307 main.Main
- Initial Heap Size (-Xms1g):
  - This parameter specifies the initial amount of memory allocated to the JVM heap when the application starts.
  - It represents the minimum size the heap will maintain during its execution.
  - Setting a sufficiently large initial heap size can reduce the need for the JVM to dynamically expand the heap during early stages of execution, potentially improving startup performance for memory-intensive applications.
- Maximum Heap Size (-Xmx2g):
  - This parameter defines the maximum amount of memory that the JVM heap can consume.
  - The heap will not grow beyond this limit, even if the application requires more memory.
  - Setting an appropriate maximum heap size is crucial to prevent OutOfMemoryError exceptions, especially for applications handling large datasets or performing complex operations.
  - However, setting it excessively high can lead to excessive memory consumption and potential performance issues if the system experiences swapping.

Memory Leaks

- Despite Java's automatic memory management, developers still need to be aware of memory-related pitfalls that can cause applications to crash, slow down, or behave unpredictably.
- The most critical of these are runtime memory errors, such as OutOfMemoryError and StackOverflowError. Understanding the causes and available diagnostic tools can help developers detect memory issues early and resolve them efficiently.
- Memory Leak
- A memory leak occurs when objects that are no longer needed remain reachable and are not collected by the garbage collector. This usually results in gradual heap growth and eventual OutOfMemoryError.

Types Of Memory Leaks

Leak Type	Description
Static Field Leaks	Objects stored in static fields remain alive for the lifetime of the application. If these objects reference large structures or grow over time (e.g., lists, caches), they prevent GC from reclaiming memory.
Listener/Callback Leaks	Event listeners, callbacks, or observers registered with UI elements, services, or frameworks but never removed, continue to reference target objects even after those objects are no longer in use.
Thread Leaks	Threads that are never properly shut down, such as long-running background threads or improperly configured thread pools, retain references to their Runnable tasks or context, leading to accumulation over time.
Collection Growth Leaks	Collections like HashMap, List, or Set that grow indefinitely without bounds, typically due to missing eviction logic or keys that are never removed, gradually consume memory.
ClassLoader Leaks	In environments like web servers or OSGi containers, classes may be loaded and unloaded dynamically. If a class holds references to its classloader or external resources (e.g., threads, static fields), it can prevent class unloading and cause memory growth with each redeploy.

Static Cache Leak

```
public class ConfigRegistry {
    private static final List<String> cache = new ArrayList<>();
    public static void add(String entry) {
        cache.add(entry); // grows indefinitely
    }
}
```

Fix: Use bounded caches or reference-based data structures:

```
private static final Map<String, Object> cache = new WeakHashMap<>();
```

OutOfMemory: java Heap Space

- This error occurs when the JVM cannot allocate an object because the heap memory is full, and garbage collection is unable to reclaim enough space.
- It often signals a memory leak or excessively large object allocations.
- Common causes include:
    - Accidental retention of objects (e.g., storing data in static fields or collections that grow unbounded)
    - Loading large files or large numbers of objects into memory
    - Inefficient caching mechanisms
    - Failure to properly release resources like database connections
- Troubleshooting tips:
    - Analyze heap dumps using tools like Eclipse MAT, VisualVM, or JProfiler
    - Monitor memory usage with jstat, jcmd, or Java Flight Recorder (JFR)
    - Increase heap size using -Xmx and optimize object allocation patterns

OutOfMemory: MetaSpace

- Introduced with Java 8, Metaspace holds class metadata. If the application loads too many classes or fails to unload them properly (often due to classloader leaks), Metaspace may grow until it exhausts native memory.
- Common causes:
    - Excessive class loading (e.g., repeated redeployment in application servers)
    - Classloader memory leaks in frameworks or custom classloaders
    - Lack of a defined Metaspace limit
- Troubleshooting tips:
    - Set a maximum limit using -XX:MaxMetaspaceSize
    - Monitor class loading with jcmd GC.class_histogram or jmap -clstats
    - Use -XX:+ClassUnloading to enable class unloading (enabled by default in modern JVMs)

OutOfMemory: GC Overheaded Limit Exceeded

- This error indicates that the JVM is spending too much time in garbage collection with too little memory actually being reclaimed, typically more than 98% of CPU time with less than 2% heap recovery.
- Causes:
    - High object allocation rate with insufficient heap
    - Memory leaks causing the heap to remain full
    - Inefficient GC configuration
- Troubleshooting tips:
    - Use profiling tools to identify allocation hotspots
    - Increase heap size and tune GC parameters
    - Investigate potential leaks and optimize object lifecycles

StackOverFlowError

- This occurs when a thread's call stack exceeds its configured size, often due to deep or infinite recursion.
- Common causes:
    - Recursive functions without a proper base case
    - Large or unbounded method call chains
    - Excessively deep expression trees in frameworks or template engines
- Troubleshooting tips:
    - Refactor recursive code to be iterative if possible
    - Review method call structure and stack trace
    - Increase stack size with -Xss, if needed (e.g., -Xss1m)

Best Practices For Memory Management

- Efficient memory management is essential for building fast, scalable, and reliable Java applications.
- While the JVM handles much of the complexity behind the scenes through automatic garbage collection and dynamic memory allocation, developers still play a critical role in ensuring that applications use memory responsibly.
- The following best practices help reduce memory waste, improve performance, and avoid common pitfalls such as memory leaks and excessive garbage collection.
  - Minimize Unnecessary Object Creation
  - Avoid Memory Leaks
  - Choose the Right Data Structures
  - Be Mindful of Object Retention in Collections
  - Tune the JVM for Your Workload
  - Monitor and Profile Regularly
  - Understand Application-Specific Memory Patterns

Minimize Unnecessary Object Creation

- Frequent object creation, especially of short-lived or redundant objects, increases pressure on the garbage collector. This can lead to more frequent minor GCs and higher CPU usage.
- Recommendations:
    - Reuse objects where possible (e.g., use StringBuilder instead of concatenating immutable strings).
    - Prefer primitive types over boxed types (e.g., int instead of Integer) when autoboxing is not needed.
    - Use object pools only when profiling shows significant performance gain.
    - Using local variables (with limited scope) can be advantageous as they get collected soon after they go out of scope.

Avoid Memory leaks

- Even with automatic garbage collection, memory leaks can occur when objects are unintentionally kept reachable and cannot be collected.
- Common causes:
    - Static references holding on to large data structures
    - Improperly managed caches
    - Unclosed resources (streams, connections)
    - Event listeners and callbacks that are never deregistered
- Best Practices
    - Use weak references (WeakReference, WeakHashMap) for caches or listeners
    - Deregister event listeners and callbacks explicitly
    - Close all resources in finally blocks or use try-with-resources
    - Monitor heap usage over time to detect unexpected growth

Choose Right Data Structure

- Using the wrong data structures can increase memory consumption and GC pressure.
- Tips:
  - Prefer ArrayList over LinkedList unless insert/remove operations dominate
  - Use EnumSet or EnumMap instead of general-purpose collections for enums
  - Avoid using synchronized collections when not needed
  - Pre-size collections if the final size is known to avoid resizing overhead
  - Be cautious with static collections (like static List or Map) as they can lead to memory leaks if not handled correctly
  - Java provides special reference types (SoftReference, WeakReference, and PhantomReference) that allow developers more control over object retention.
  - Objects referenced by a WeakReference are cleared by the garbage collector as soon as they are not strongly referenced. SoftReference is similar but can keep the object longer (useful for caches). PhantomReference can be beneficial for scheduling pre-finalization actions.

Be Mindful Of Object Retention

- Using the wrong data structures can increase memory consumption and GC pressure.
- Collections like HashMap, ArrayList, or ConcurrentHashMap can retain references to objects that are no longer needed, preventing them from being garbage collected.
- Avoid:

```
List<byte[]> memoryLeakList = new ArrayList<>(); while (true) {
        memoryLeakList.add(new byte[1024 * 1024]); // 1MB chunks }
```

- Do:
  - Clear collections when they are no longer needed
  - Limit the scope of long-lived caches or queues
  - Use bounded collections where applicable (e.g., LinkedBlockingQueue with capacity)

Tune JVM For Your WorkLoad

- JVM memory settings have a significant impact on performance. Use flags like: -Xms512m -Xmx2g -Xss1m -XX:+UseG1GC
- Tuning advice:
    - Set heap size (-Xms, -Xmx) based on expected workload
    - Choose a GC algorithm that aligns with your latency/throughput goals
    - Profile with real-world traffic and fine-tune GC behavior
    - Monitor GC logs or use JFR to guide tuning decisions

Monitor And Profile Regularly

- Proactive monitoring is essential for identifying memory issues before they cause production outages.
- Recommended tools:
    - VisualVM, JFR + Mission Control for real-time profiling
    - GarbageCat for offline GC log analysis
    - jstat, jmap, jcmd for command-line diagnostics
    - Establish regular performance profiling as part of your CI/CD or release workflow.

Understand Application Specific Patterns

- Different types of Java applications have different memory footprints:
  - Web servers may have many short-lived objects (benefits from efficient young generation tuning)
  - Stream processors often maintain stateful objects (heap size and compaction become critical)
  - GUI applications are sensitive to GC pauses (low-latency GC collectors may be required)
- Know your application's behavior and design accordingly.

Key Takeaways

- JVM memory is divided into distinct areas, including the heap, stack, Metaspace, PC register, and native method stack, each serving specific roles in program execution.
- Heap memory stores all dynamically allocated objects and is managed by the garbage collector. It is further divided into Young and Old Generations to optimize object lifecycle management.
- Stack memory is thread-specific and holds method frames, including local variables and object references. The actual object data referenced in the stack resides in the heap.
- Metaspace, introduced in Java 8, replaces PermGen and stores class metadata in native memory. While it grows automatically, it can still lead to OutOfMemoryError if not monitored.
- Garbage Collection (GC) is automatic in Java but can be tuned with JVM flags. Understanding GC phases (mark, sweep, compact) and the difference between Minor, Major, and Full GC is key to effective memory management.

ThankYou

shalini06mittal@gmail.com

7738460004