JConsole and JVisualVM

Created	@February 18, 2022 4:37 PM
Reviewed	☐

Table of contents

$JConsole$

Using JConsole

Chapter 3 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

https://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html

▼ Introduction

▼ Starting JConsole

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. If this directory is in your system path, you can start JConsole by simply typing jconsole in a command (shell) prompt. Otherwise, you have to type the full path to the executable file.

▼ Setting up local monitor

You start JConsole by typing the following command at the command line from anywhere in the PC.

```
% jconsole
```

When JConsole starts, you will be given a choice of all the Java applications that are running locally that JConsole can connect to.

If you want to monitor a specific application, and you know that application's process ID, then you can also start JConsole so that it connects to that application. This application must be running with the same user ID as JConsole. The command syntax to start JConsole for local monitoring of a specific application is the following.

```
% jconsole processID
```

In the command above *processID* is the application's process ID (PID).

Determine an application's PID:

- On Solaris, Linux, or Mac OS X systems, you can use the ps command to find the PID of the java instance that is running.
- On Windows systems, you can use the Task Manager to find the PID of java or javaw.
- You can also use the **jps** command-line utility to determine PIDs. See the manual page for the **jps** tool (Solaris, Linux, or Mac OS X or Windows).
- ps -a also works.

▼ Setting up remote monitor

To start JConsole for remote monitoring, you use the following command syntax.

```
% jconsole hostName:portNum
```

In the command above, *hostName* is the name of the system running the application and *portNum* is the port number you specified when you enabled the JMX agent when you started the Java VM. For more information, see Remote Monitoring and Management.

▼ Presenting the JConsole Tabs

Once you have connected JConsole to an application, JConsole is composed of six tabs.

- **Overview:** Displays overview information about the Java VM and monitored values.
- **Memory:** Displays information about memory use.
- **Threads:** Displays information about thread use.
- **Classes:** Displays information about class loading.
- **VM Summary:** Displays information about the Java VM.
- **MBeans:** Displays information about MBeans. An MBean can represent a device, an **application, or any resource that needs to be managed**.

You can use the green connection status icon in the upper right-hand corner of JConsole at any time, **to disconnect from or reconnect to a running Java VM.** You can connect to any number of running Java VMs at a time by selecting Connection then New Connection from the drop-down menu.

▼ Saving chart data

JConsole allows you to save the data presented in the charts in a Comma Separated Values (**CSV**) file. To save data from a chart, simply right-click on any chart, select Save data as..., and then specify the file in which the data will be saved. You can save the data from any of the charts displayed in any of JConsole's different tabs in this way.

The CSV format is commonly used for data exchange between spreadsheet applications. The CSV file can be imported into spreadsheet applications and can be used to create diagrams in these applications. The data is presented as two or

more named columns, where the first column represents the time stamps. After importing the file into a spreadsheet application, you will usually need to select the first column and change its format to be "date" or "date/time" as appropriate.

▼ Java hotspot VM

HotSpot, released as Java HotSpot Performance Engine, is a Java virtual machine for desktop and server computers, developed by Sun Microsystems and now maintained and distributed by Oracle Corporation. It features improved performance via methods such as just-in-time compilation and adaptive optimization.

▼ Memory

▼ Video

Good video on complete JRE understanding.

https://www.youtube.com/watch?v=aAjkJW08BGQ

JVM architecture: basically how it works

https://www.youtube.com/watch?v=LxU4KeMqZL0

Cover many topics.

https://www.youtube.com/watch?v=bspS-uTK0IM

▼ Heap, Non-heap Memory and stack

The bar chart on the lower right-hand side shows the memory consumed by the memory pools in heap and non-heap memory. The bar will turn red when the memory used exceeds the memory usage threshold. **You can set the memory usage threshold through an attribute of the MemoryMXBean.**

Stack is a separate part which does not belongs to neither heap nor non-heap. A JVM stack, also called thread stack, is **a data area in the JVM memory created for a single execution thread**. The JVM stack of a thread is used by the thread to store local variables, partial results, and data for method invocations and returns.

For every thread, **JVM** creates a separate stack at the time of thread creation. The memory for a Java Virtual Machine stack does not need to be contiguous. The Java virtual machine only performs two operations directly on Java stacks: it pushes and pops frames. And stack for a particular thread may be termed as **Run – Time Stack**. Every method call performed by that thread is stored in the corresponding run-time stack including parameters, local variables, intermediate computations, and other data. After completing a method, the corresponding entry from the stack is removed. After completing all method calls the stack becomes empty and that empty stack is destroyed by the JVM just before terminating the thread. The data stored in the stack is available for the corresponding thread and not available to the remaining threads. Hence we can say local data thread-safe. Each entry in the stack is called **Stack Frame** or **Activation Record**.

Link: https://www.geeksforgeeks.org/java-virtual-machine-jvm-stack-area/#:~:text=For every thread%2C JVM creates,it pushes and pops frames.&text=Hence we can say local data thread-safe

The Java VM manages two kinds of memory: heap and non-heap memory, both of which are created when the Java VM starts.

- **Heap memory** is the runtime data area from which memory for all **class instances i.e. objects and arrays** are allocated. It is created at the Java virtual machine start-up. Heap memory for objects is reclaimed by an automatic memory management system which is known as a g*arbage collector*.

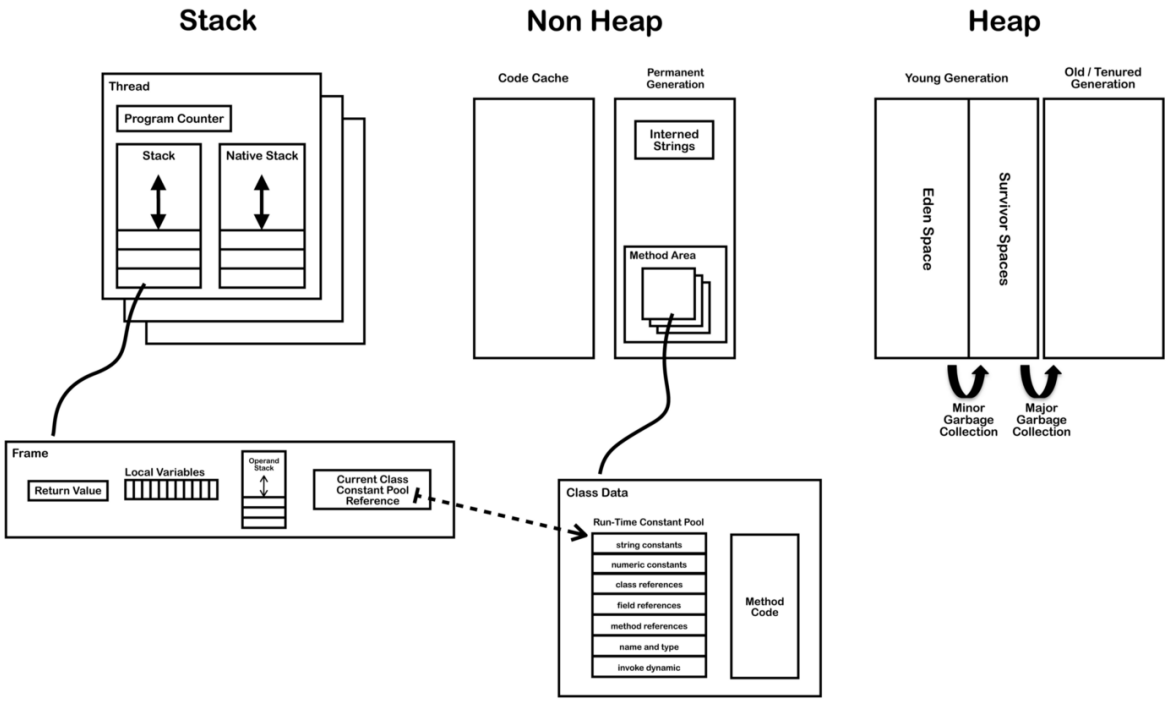
  The heap may be of a fixed size or may be expanded and shrunk. The memory for the heap does not need to be contiguous.

- **Non-heap memory:** The Java virtual machine manages memory other than the heap (referred as *non-heap memory*).

  The Java virtual machine has a *method area* that is shared among all threads. The method area belongs to non-heap memory. **It stores per-class structures such as a runtime constant pool, field and method data, and the code for methods and constructors**. It is created at the Java virtual machine start-up.

  The method area is logically part of the heap but a Java virtual machine implementation may choose not to either garbage collect or compact it. Similar to the heap, the method area may be of a fixed size or may be expanded and shrunk. The memory for the method area does not need to be contiguous.

  In addition to the method area, a Java virtual machine implementation may require memory for internal processing or optimization which also belongs to non-heap memory. For example, the JIT compiler **requires memory for storing the native machine code** translated from the Java virtual machine code for high performance



▼ Memory types

The Memory tab features a “Perform GC” button that you can click to perform garbage collection whenever you want. The chart shows the memory use of the Java VM over time, for heap and non-heap memory, as well as for specific memory pools. The memory pools available depend on which version of the Java VM is being used. For the HotSpot Java VM, the memory pools for serial garbage collection are the following.

- *Eden Space (heap)*: The pool from which memory is **initially allocated** for most objects.
- *Survivor Space (heap)*: The pool containing objects that have survived the garbage collection of the Eden space.

  What happens if survivor space is full?

  Note: If the To space becomes full, **the live objects from Eden or From that have not been copied to it are tenured**, regardless of how many young generation collections they have survived.

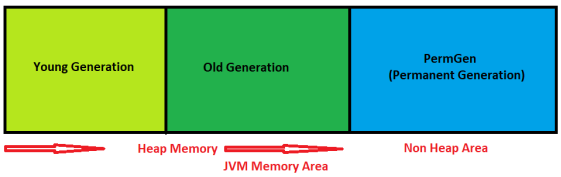
- *Tenured Generation (heap) or old gen*: The pool containing objects that **have existed for some time** in the survivor space.
- *Permanent Generation (non-heap)*: The pool containing all the reflective data of the virtual machine itself, such as **class and method objects**. With Java VMs that use class data sharing, this generation is divided into read-only and read-write areas.

  PermGen is replaced by **Metaspace** with the capability to auto increase the native memory as per the requirement to load the class metadata.

▼ Permgen and metaspace

Method Area is a part of space in the PermGen and it is used to store the class structure and the code for methods and constructors. The biggest disadvantage of PermGen is that it contains a limited size which leads to an **OutOfMemoryError**. The default size of PermGen memory is 64 MB on 32-bit JVM and 82 MB on the 64-bit version. Due to this, JVM had to change the size of this memory by frequently performing Garbage collection which is a costly operation. Java also allows to manually change the size of the PermGen memory. However, the PermGen space cannot be made to auto increase. So, it is difficult to tune it. And also, the garbage collector is not efficient enough to clean the memory.

Due to the above problems, PermGen has been completely removed in Java 8. In the place of PermGen, a new feature called Meta Space has been introduced. MetaSpace grows automatically by default. Here, the garbage collection is automatically triggered when the class metadata usage reaches its maximum metaspace size.

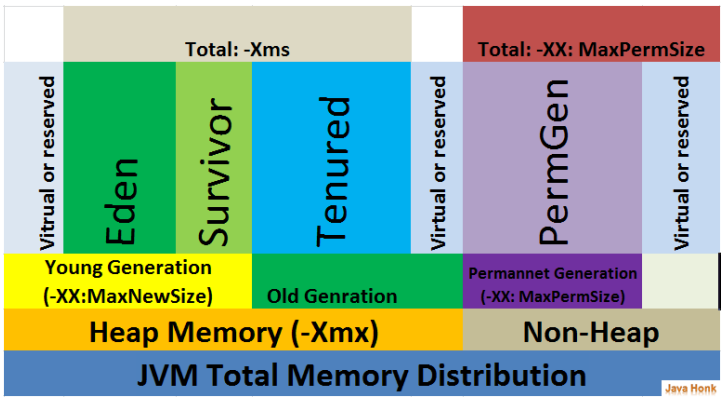


Method area contains:

- Class structure
- static variables
- constants

> In addition to the method area, a Java VM may require memory for internal processing or optimization which also belongs to non-heap memory. For example, the Just-In-Time (JIT) compiler requires memory for storing the native machine code translated from the Java VM code for high performance.

- *Code Cache (non-heap)*: The HotSpot Java VM also includes a code cache, containing memory that is used for **compilation and storage of native code**. Native code refers **to programming code that is configured to run on a specific processor.**



Minor GC performed on the young generation.

Major GC performed on the old generation which takes more amount of time.

Stack memory size is very less as compared to the heap memory.

Java stack memory stores (store based on concept of LIFO):

- methods specific value
- local variables (int i =0)
- objects references which are referring to some objects on heap memory

  Test t1 = new Test()

  t1 is reference and would be stored in the stack while new test would be created in the heap memory.
- public void mehtod1(){

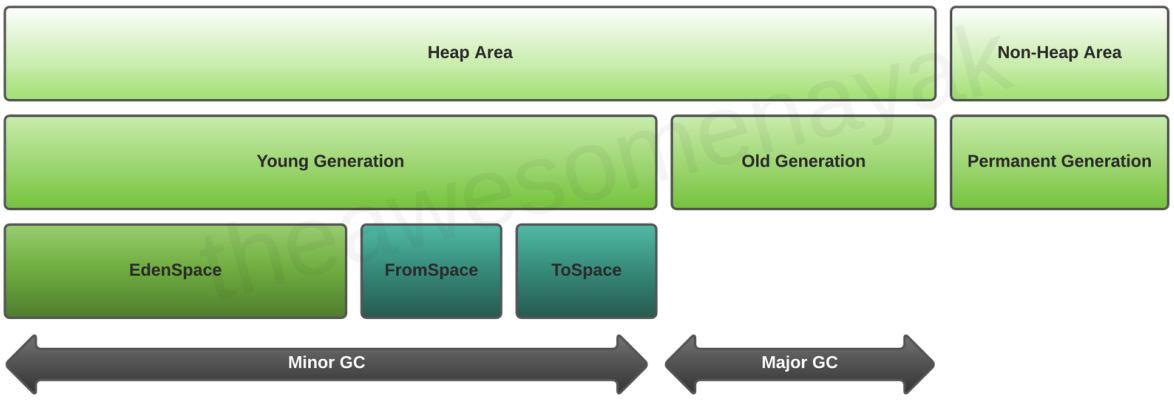
  sout(”my new method”)

  }

  When such method would be called it would be stored in a method block inside the stack memory. And when it is executed successfully the method block space is given to other method.

▼ Garbage collector

GC does not work in non heap.



Garbage collection (GC) is how the Java VM frees memory occupied by objects that are no longer referenced. It is common to think of objects that have active references as being "alive" and non-referenced (or unreachable) objects as

"dead." Garbage collection is the process of releasing memory used by the dead objects. The algorithms and parameters used by GC can have dramatic effects on performance.

There is no GC in non heap. **The size of the non heap memory automatically increases with the help of JVM.** When we manually perform GC in jvisualVM then it recalculates and sets the non-heap size accordingly.

The Java HotSpot VM garbage collector uses generational GC. Generational GC takes advantage of the observation that most programs conform to the following generalizations.

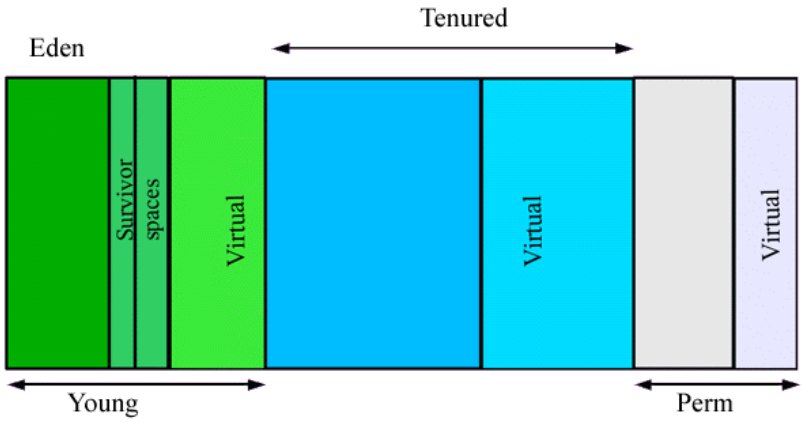
- They create many objects that have short lives, for example, iterators and local variables.
- They create some objects that have very long lives, for example, high level persistent objects.

Generational GC divides memory into several generations, and assigns one or more memory pools to each. When a generation uses up its allotted memory, the VM performs a partial GC (also called a minor collection) on that memory pool to reclaim memory used by dead objects. This partial GC is usually much faster than a full GC.

The Java HotSpot VM defines two generations: the young generation (sometimes called the "nursery") and the old generation. The young generation consists of an "Eden space" and two "survivor spaces." The VM initially assigns all objects to the Eden space, and most objects die there. When it performs a minor GC, the VM moves any remaining objects from the Eden space to one of the survivor spaces. The VM moves objects that live long enough in the survivor spaces to the "tenured" space in the old generation. When the tenured generation fills up, there is a full GC that is often much slower because it involves all live objects. The permanent generation holds all the reflective data of the virtual machine itself, such as class and method objects.

The default arrangement of generations looks something like Figure 3-7.

Figure 3-7 Generations of Data in Garbage Collection



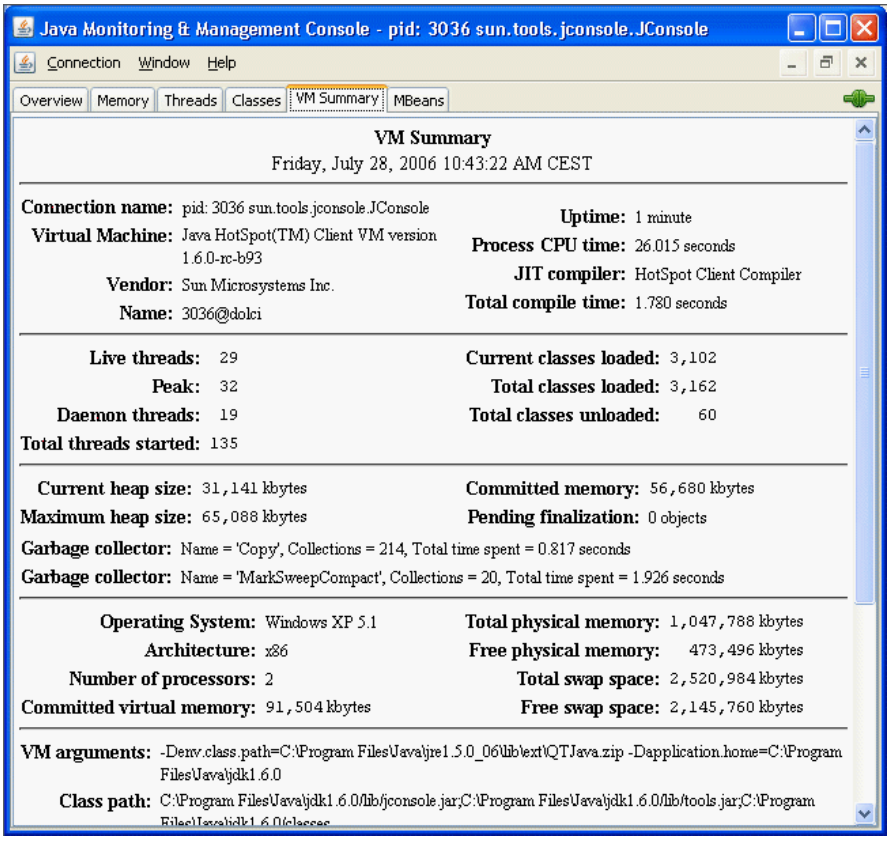
If the garbage collector has become a bottleneck, you can improve performance by customizing the generation sizes. Using JConsole, you can investigate the sensitivity of your performance metric by experimenting with the garbage collector parameters. For more information, see Tuning Garbage Collection with the 5.0 HotSpot VM.

▼ Memory metrices

- *Used*: the amount of memory currently used, including the memory occupied by all objects, both **reachable and unreachable**.
- *Committed*: the amount of memory guaranteed to be available for use by the Java VM. The amount of committed memory may change over time. The Java virtual machine may release memory to the system and the amount of committed memory could be less than the amount of memory initially allocated at start up. The amount of **committed memory will always be greater than or equal to the amount of used memory**.

- *Max*: the maximum amount of memory that can be used for memory management. Its value may change or be undefined. A memory allocation may fail if the Java VM attempts to increase the used memory to be greater than committed memory, even if the amount used is less than or equal to max (for example, when the system is low on virtual memory).
- *GC time*: the cumulative time spent on garbage collection and the total number of invocations. It may have multiple rows, each of which represents one garbage collector algorithm used in the Java VM.

▼ VM summary



The information presented in this tab includes the following.

- **Summary**
  - *Uptime*: Total amount of time since the Java VM was started.
  - *Process CPU Time*: Total amount of CPU time that the Java VM has consumed since it was started.
  - *Total Compile Time*: Total accumulated time spent in JIT compilation. The Java VM determines when JIT compilation occurs. The Hotspot VM uses adaptive compilation, in which the VM launches an application using a standard interpreter, but then analyzes the code as it runs to detect performance bottlenecks, or "hot spots".
- **Threads**
  - *Live threads*: Current number of live daemon threads plus non-daemon threads.
  - *Peak*: Highest number of live threads since Java VM started.
  - *Daemon threads*: Current number of live daemon threads.
  - *Total threads started*: Total number of threads started since Java VM started, including daemon, non-daemon, and **terminated** threads.

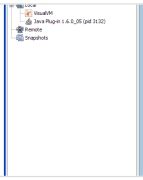
- **Classes**
  - *Current classes loaded*: Number of classes currently loaded into memory.
  - *Total classes loaded*: Total number of classes loaded into memory since the Java VM started, including those that have subsequently been unloaded.
  - *Total classes unloaded*: Number of classes unloaded from memory since the Java VM started.
- **Memory**
  - *Current heap size*: Number of kilobytes currently occupied by the heap.
  - *Committed memory*: Total amount of memory allocated for use by the heap.
  - *Maximum heap size*: Maximum number of kilobytes occupied by the heap.
  - *Objects pending for finalization*: Number of objects pending for finalization.
  - *Garbage collector*: Information about garbage collection, including the garbage collector names, number of collections performed, and total time spent performing GC.
- **Operating System**
  - *Total physical memory*: Amount of random-access memory (RAM) the operating system has.
  - *Free physical memory*: Amount of free RAM available to the operating system.
  - *Committed virtual memory*: Amount of virtual memory guaranteed to be available to the running process.
- **Other Information**
  - *VM arguments*: The input arguments the application passed to the Java VM, not including the arguments to the main method.
  - *Class path*: The class path that is used by the system class loader to search for class files.
  - *Library path*: The list of paths to search when loading libraries.
  - *Boot class path*: The boot class path is used by the bootstrap class loader to search for class files.

$$Java\ VisualVM$$

Java VisualVM

Java VisualVM is a tool that provides a visual interface for viewing detailed information about Java technology-based applications (Java applications) while they are running on a Java Virtual Machine (JVM). Java VisualVM organizes JVM data that is retrieved by the Java Development Kit (JDK) tools and presents the information in

https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/intro.html



▼ Overview

The Overview tab displays the following general information about the application and the runtime environment.

- **PID.** This is the process ID of the application. The process ID is also displayed next to the application node in the Applications window.
- **Host.** This is the location of the system that the application is running on.
- **Main class.** The class containing the method.

  main
- **Arguments.** Displays any options passed to the application on startup.
- **JVM.** This is the JDK version of the JVM software.

- **Java Home.** This is the location of the JDK software.
- **JVM flags.** This displays any flags used by the JVM software when starting the JDK software.
- **Heap dump on OOME.** This displays the status of the Heap Dump on OOME option. When enabled, a heap dump is taken when the application encounters an OutOfMemory exception. You can enabled or disable the option in the application popup menu.

▼ Profiling

Java VisualVM presents data for local and remote applications in a tab specific for that application. You can have multiple application tabs open. Each application tab contains sub-tabs that display different types of information about the application.

Profiling applications

The Profiler tab of an application enables you to start and stop the profiling session of a local application. Profiling results are displayed in the Profiler tab. You can use the toolbar to refresh the profiling results, invoke garbage collection and save the profiling data.

By default the profiling tool is not running until you are ready to profile the application. You can choose from the following profiling options:

- **CPU Profiling.** Choose this to profile the performance of the application.
- **Memory Profiling.** Choose this to analyze the memory usage of the application. The results display the objects allocated by the application and the class allocating those objects.

When you start a profiling session, Java VisualVM attaches to the local application and starts collecting profiling data. When profiling results are available they are automatically displayed in the Profiler tab.

▼ Heap dump

You can use Java VisualVM to browse the contents of a heap dump file and quickly see the allocated objects in the heap. Heap dumps are displayed in the heap dump sub-tab in the main window. You can open binary format heap dump files (.hprof) saved on your local system or use Java VisualVM to take heap dumps of running applications.

A heap dump is a snapshot of all the objects in the Java Virtual Machine (JVM) heap at a certain point in time. The JVM software allocates memory for objects from the heap for all class instances and arrays. The garbage collector reclaims the heap memory when an object is no longer needed and there are no references to the object. By examining the heap you can locate where objects are created and find the references to those objects in the source. If the JVM software is failing to remove unneeded objects from the heap, Java VisualVM can help you locate the nearest garbage collecting root for the object.

Instances View

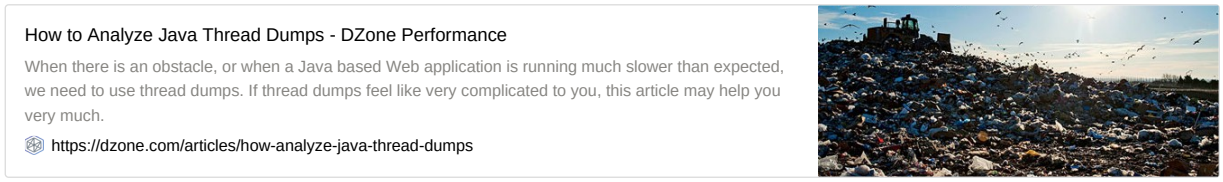
The Instance view displays object instances for a selected class. When you select an instance from the Instance pane, Java VisualVM displays the fields of that class and references to that class in the respective panes. In the References pane, you can right-click an item and choose Show Nearest GC Root to display the nearest garbage collection root object.

▼ Thread dump

Healthy thread: thread waiting for a task

Non-healthy thread: Thread executing a task for too long or STUCK

When there is an obstacle, or when a Java based Web application is running much slower than expected, we need to use **thread dumps**. If thread dumps feel like very complicated to you, this article may help you very much. Here I will explain what threads are in Java, their types, how they are created, how to manage them, how you can dump threads from a running application, and finally how you can analyze them and determine the bottleneck or blocking threads. This article is a result of long experience in Java application debugging.



Java and Thread

A web server uses tens to hundreds of threads to process a large number of concurrent users. If two or more threads utilize the same resources, a *contention* between the threads is inevitable, and sometimes deadlock occurs.

Thread contention is a status in which one thread is waiting for a lock, held by another thread, to be lifted. Different threads frequently access shared resources on a web application. For example, to record a log, the thread trying to record the log must obtain a lock and access the shared resources.

Deadlock is a special type of thread contention, in which two or more threads are waiting for the other threads to complete their tasks in order to complete their own tasks.

Different issues can arise from thread contention. To analyze such issues, you need to use the **thread dump**. A thread dump will give you the information on the exact status of each thread.

Background Information for Java Threads

Thread Synchronization

A thread can be processed with other threads at the same time. In order to ensure compatibility when multiple threads are trying to use shared resources, one thread at a time should be allowed to access the shared resources by using *thread synchronization*.

Thread synchronization on Java can be done using **monitor**. Every Java object has a single monitor. The monitor can be owned by only one thread. For a thread to own a monitor that is owned by a different thread, it needs to wait in the wait queue until the other thread releases its monitor.

Thread Status

In order to analyze a thread dump, you need to know the status of threads. The statuses of threads are stated on java.lang.Thread.State.

Figure 1: Thread Status.

- **NEW**: The thread is created but has **not been processed** yet.
- **RUNNABLE**: The thread is occupying the CPU and **processing a task**. (It may be in WAITING status due to the OS's resource distribution.)
- **BLOCKED**: The thread is waiting for a different thread to **release its lock** in order to get the monitor lock.
- **WAITING**: The thread is waiting by using a wait, join or park method.
- **TIMED_WAITING**: The thread is waiting by using a sleep, wait, join or park method. (The **difference from WAITING is that the maximum waiting time is specified by the method parameter**, and *WAITING* can be relieved by time as well as external changes.)

Thread Types

Java threads can be divided into two:

1. daemon threads;
2. and non-daemon threads.

Daemon threads stop working when there are no other non-daemon threads. Even if you do not create any threads, the Java application will create several threads by default. Most of them are daemon threads, mainly for processing tasks such as garbage collection or JMX.

A thread running the 'static void main(String[] args)' method is created as a non-daemon thread, and when this thread stops working, all other daemon threads will stop as well. (The thread running this main method is called the **VM thread in HotSpot VM**.)

Getting a Thread Dump

We will introduce the three most commonly used methods. Note that there are many other ways to get a thread dump. A thread dump can only show the thread status at the time of measurement, so in order to see the change in thread status, it is recommended to extract them from 5 to 10 times with 5-second intervals.

Thread Information from the Thread Dump File

```
"pool-1-thread-13" prio=6 tid=0x000000000729a000 nid=0x2fb4 runnable [0x0000000007f0f000] java.lang.Thread.State: RUNNABLE
          at java.net.SocketInputStream.socketRead0(Native Method)

          at java.net.SocketInputStream.read(SocketInputStream.java:129)

          at sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:264)

          at sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:306)

          at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:158)

          - locked <0x0000000780b7e688> (a java.io.InputStreamReader)

          at java.io.InputStreamReader.read(InputStreamReader.java:167)

          at java.io.BufferedReader.fill(BufferedReader.java:136)

          at java.io.BufferedReader.readLine(BufferedReader.java:299)

          - locked <0x0000000780b7e688> (a java.io.InputStreamReader)

          at java.io.BufferedReader.readLine(BufferedReader.java:362)
```

- Thread name: When using Java.lang.Thread class to generate a thread, the thread will be named Thread-(Number), whereas when using java.util.concurrent.ThreadFactory class, it will be named pool-(number)-thread-(number).
- Priority: Represents the priority of the threads.
- Thread ID: Represents the unique ID for the threads. (Some useful information, including the CPU usage or memory usage of the thread, can be obtained by using thread ID.)
- **nid** is Native Thread ID : Crucial information as this native Thread Id allows you to correlate for example which Threads from an OS perspective are using the most CPU within your JVM etc.
- Thread status: Represents the status of the threads.
- Thread callstack: Represents the call stack information of the threads.

  A call stack is **a stack data structure that stores information about the active subroutines of a computer program**.

Thread Dump Patterns by Type

When Unable to Obtain a Lock (BLOCKED)

This is when the overall performance of the application slows down because a thread is occupying the lock and prevents other threads from obtaining it. In the following example, BLOCKED_TEST pool-1-thread-1 thread is running with <0x0000000780a000b0> lock, while BLOCKED_TEST pool-1-thread-2 and BLOCKED_TEST pool-1-thread-3 threads are waiting to obtain <0x0000000780a000b0> lock.

```
"BLOCKED_TEST pool-1-thread-1" prio=6 tid=0x0000000006904800 nid=0x28f4 runnable [0x000000000785f000]
   java.lang.Thread.State: RUNNABLE
                at java.io.FileOutputStream.writeBytes(Native Method)
                at java.io.FileOutputStream.write(FileOutputStream.java:282)
                at java.io.BufferedOutputStream.flushBuffer(BufferedOutputStream.java:65)
                at java.io.BufferedOutputStream.flush(BufferedOutputStream.java:123)
                - locked <0x0000000780a31778> (a java.io.BufferedOutputStream)
                at java.io.PrintStream.write(PrintStream.java:432)
                - locked <0x0000000780a04118> (a java.io.PrintStream)
                at sun.nio.cs.StreamEncoder.writeBytes(StreamEncoder.java:202)
                at sun.nio.cs.StreamEncoder.implFlushBuffer(StreamEncoder.java:272)
                at sun.nio.cs.StreamEncoder.flushBuffer(StreamEncoder.java:85)
                - locked <0x0000000780a040c0> (a java.io.OutputStreamWriter)
                at java.io.OutputStreamWriter.flushBuffer(OutputStreamWriter.java:168)
                at java.io.PrintStream.newLine(PrintStream.java:496)
                - locked <0x0000000780a04118> (a java.io.PrintStream)
                at java.io.PrintStream.println(PrintStream.java:687)
                - locked <0x0000000780a04118> (a java.io.PrintStream)
                at com.nbp.theplatform.threaddump.ThreadBlockedState.monitorLock(ThreadBlockedState.java:44)
                - locked <0x0000000780a000b0> (a com.nbp.theplatform.threaddump.ThreadBlockedState)
                at com.nbp.theplatform.threaddump.ThreadBlockedState$1.run(ThreadBlockedState.java:7)
                at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:886)
                at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:908)
                at java.lang.Thread.run(Thread.java:662)

   Locked ownable synchronizers:
                - <0x0000000780a31758> (a java.util.concurrent.locks.ReentrantLock$NonfairSync)

"BLOCKED_TEST pool-1-thread-2" prio=6 tid=0x0000000007673800 nid=0x260c waiting for monitor entry [0x0000000008abf000]
   java.lang.Thread.State: BLOCKED (on object monitor)
                at com.nbp.theplatform.threaddump.ThreadBlockedState.monitorLock(ThreadBlockedState.java:43)
                - waiting to lock <0x0000000780a000b0> (a com.nbp.theplatform.threaddump.ThreadBlockedState)
                at com.nbp.theplatform.threaddump.ThreadBlockedState$2.run(ThreadBlockedState.java:26)
                at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:886)
                at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:908)
                at java.lang.Thread.run(Thread.java:662)

   Locked ownable synchronizers:
                - <0x0000000780b0c6a0> (a java.util.concurrent.locks.ReentrantLock$NonfairSync)

"BLOCKED_TEST pool-1-thread-3" prio=6 tid=0x00000000074f5800 nid=0x1994 waiting for monitor entry [0x0000000008bbf000]
   java.lang.Thread.State: BLOCKED (on object monitor)
                at com.nbp.theplatform.threaddump.ThreadBlockedState.monitorLock(ThreadBlockedState.java:42)
                - waiting to lock <0x0000000780a000b0> (a com.nbp.theplatform.threaddump.ThreadBlockedState)
                at com.nbp.theplatform.threaddump.ThreadBlockedState$3.run(ThreadBlockedState.java:34)
                at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:886
                at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:908)
                at java.lang.Thread.run(Thread.java:662)

   Locked ownable synchronizers:
                - <0x0000000780b0e1b8> (a java.util.concurrent.locks.ReentrantLock$NonfairSync)
```

When in Deadlock Status

This is when *thread A* needs to obtain *thread B*'s lock to continue its task, while *thread B* needs to obtain *thread A*'s lock to continue its task. In the thread dump, you can see that DEADLOCK_TEST-1 thread has 0x00000007d58f5e48 lock, and is trying to obtain 0x00000007d58f5e60 lock. You can also see that DEADLOCK_TEST-2 thread has 0x00000007d58f5e60 lock, and is trying to obtain 0x00000007d58f5e78 lock. Also, DEADLOCK_TEST-3 thread has 0x00000007d58f5e78 lock, and is

trying to obtain 0x00000007d58f5e48 lock. As you can see, each thread is waiting to obtain another thread's lock, and this status will not change until one thread discards its lock.

```
"DEADLOCK_TEST-1" daemon prio=6 tid=0x0000000000690f800 nid=0x1820 waiting for monitor entry [0x000000000805f000]
   java.lang.Thread.State: BLOCKED (on object monitor)
                at com.nbp.theplatform.threaddump.ThreadDeadLockState$DeadlockThread.goMonitorDeadlock(ThreadDeadLockState.java:197)
                - waiting to lock <0x00000007d58f5e60> (a com.nbp.theplatform.threaddump.ThreadDeadLockState$Monitor)
                at com.nbp.theplatform.threaddump.ThreadDeadLockState$DeadlockThread.monitorOurLock(ThreadDeadLockState.java:182)
                - locked <0x00000007d58f5e48> (a com.nbp.theplatform.threaddump.ThreadDeadLockState$Monitor)
                at com.nbp.theplatform.threaddump.ThreadDeadLockState$DeadlockThread.run(ThreadDeadLockState.java:135)

   Locked ownable synchronizers:
                - None

"DEADLOCK_TEST-2" daemon prio=6 tid=0x0000000006858800 nid=0x17b8 waiting for monitor entry [0x000000000815f000]
   java.lang.Thread.State: BLOCKED (on object monitor)
                at com.nbp.theplatform.threaddump.ThreadDeadLockState$DeadlockThread.goMonitorDeadlock(ThreadDeadLockState.java:197)
                - waiting to lock <0x00000007d58f5e78> (a com.nbp.theplatform.threaddump.ThreadDeadLockState$Monitor)
                at com.nbp.theplatform.threaddump.ThreadDeadLockState$DeadlockThread.monitorOurLock(ThreadDeadLockState.java:182)
                - locked <0x00000007d58f5e60> (a com.nbp.theplatform.threaddump.ThreadDeadLockState$Monitor)
                at com.nbp.theplatform.threaddump.ThreadDeadLockState$DeadlockThread.run(ThreadDeadLockState.java:135)

   Locked ownable synchronizers:
                - None

"DEADLOCK_TEST-3" daemon prio=6 tid=0x0000000006859000 nid=0x25dc waiting for monitor entry [0x000000000825f000]
   java.lang.Thread.State: BLOCKED (on object monitor)
                at com.nbp.theplatform.threaddump.ThreadDeadLockState$DeadlockThread.goMonitorDeadlock(ThreadDeadLockState.java:197)
                - waiting to lock <0x00000007d58f5e48> (a com.nbp.theplatform.threaddump.ThreadDeadLockState$Monitor)
                at com.nbp.theplatform.threaddump.ThreadDeadLockState$DeadlockThread.monitorOurLock(ThreadDeadLockState.java:182)
                - locked <0x00000007d58f5e78> (a com.nbp.theplatform.threaddump.ThreadDeadLockState$Monitor)
                at com.nbp.theplatform.threaddump.ThreadDeadLockState$DeadlockThread.run(ThreadDeadLockState.java:135)

   Locked ownable synchronizers:
                - None
```

When Continuously Waiting to Receive Messages from a Remote Server

The thread appears to be normal, since its state keeps showing as RUNNABLE (when dump taken at different interval through showing runnable only). However, when you align the thread dumps chronologically, you can see that socketReadThread thread is waiting infinitely to read the socket.

```
"socketReadThread" prio=6 tid=0x0000000006a0d800 nid=0x1b40 runnable [0x00000000089ef000]
   java.lang.Thread.State: RUNNABLE
                at java.net.SocketInputStream.socketRead0(Native Method)
                at java.net.SocketInputStream.read(SocketInputStream.java:129)
                at sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:264)
                at sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:306)
                at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:158)
                - locked <0x00000007d78a2230> (a java.io.InputStreamReader)
                at sun.nio.cs.StreamDecoder.read0(StreamDecoder.java:107)
                - locked <0x00000007d78a2230> (a java.io.InputStreamReader)
                at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:93)
                at java.io.InputStreamReader.read(InputStreamReader.java:151)
                at com.nbp.theplatform.threaddump.ThreadSocketReadState$1.run(ThreadSocketReadState.java:27)
                at java.lang.Thread.run(Thread.java:662)
```

When Waiting

The thread is maintaining WAIT status. In the thread dump, IoWaitThread thread keeps waiting to receive a message from LinkedBlockingQueue. If there continues to be no message for LinkedBlockingQueue, then the thread status will not change.

```
"IoWaitThread" prio=6 tid=0x0000000007334800 nid=0x2b3c waiting on condition [0x000000000893f000]
   java.lang.Thread.State: WAITING (parking)
               at sun.misc.Unsafe.park(Native Method)
               - parking to wait for  <0x00000007d5c45850> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
               at java.util.concurrent.locks.LockSupport.park(LockSupport.java:156)
               at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:1987)
               at java.util.concurrent.LinkedBlockingDeque.takeFirst(LinkedBlockingDeque.java:440)
               at java.util.concurrent.LinkedBlockingDeque.take(LinkedBlockingDeque.java:629)
               at com.nbp.theplatform.threaddump.ThreadIoWaitState$IoWaitHandler2.run(ThreadIoWaitState.java:89)
               at java.lang.Thread.run(Thread.java:662)
```

When Thread Resources Cannot be Organized Normally

Unnecessary threads will pile up when thread resources cannot be organized normally. If this occurs, it is recommended to monitor the thread organization process or check the conditions for thread termination.

How to Solve Problems by Using Thread Dump

Example 1: When the CPU Usage is Abnormally High

1. Extract the thread that has the highest CPU usage.

```
[user@linux ~]$ ps -mo pid.lwp.stime.time.cpu -C java

       PID         LWP        STIME            TIME            %CPU
       10029        -         Dec07          00:02:02          99.5
         -       10039        Dec07          00:00:00          0.1
         -       10040        Dec07          00:00:00          95.5
```

Use ***top -c***
(then press ctrl+i to close the iris mode)
The top program provides a dynamic real-time view of a running system. It can display system summary information as well as a list of processes or threads currently being managed by the Linux kernel.

ps -o pid,lwp,stime,time,cpu -C java
-o helps to specify the format of the columns.

From the application, find out which thread is using the CPU the most.

Acquire the *Light Weight Process* (LWP) that uses the CPU the most and convert its unique number (10039) into a hexadecimal number (0x2737).

2. After acquiring the thread dump, check the thread's action.

Extract the thread dump of an application with a PID of 10029, then find the thread with an nid of 0x2737.

```
"NioProcessor-2" prio=10 tid=0x0a8d2800 nid=0x2737 runnable [0x49aa5000]
java.lang.Thread.State: RUNNABLE
               at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)
               at sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:210)
               at sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:65)
               at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:69)
               - locked <0x74c52678> (a sun.nio.ch.Util$1)
               - locked <0x74c52668> (a java.util.Collections$UnmodifiableSet)
               - locked <0x74c501b0> (a sun.nio.ch.EPollSelectorImpl)
               at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:80)
               at external.org.apache.mina.transport.socket.nio.NioProcessor.select(NioProcessor.java:65)
               at external.org.apache.mina.common.AbstractPollingIoProcessor$Worker.run(AbstractPollingIoProcessor.java:708)
               at external.org.apache.mina.util.NamePreservingRunnable.run(NamePreservingRunnable.java:51)
```

```
                at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:886)
                at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:908)
                at java.lang.Thread.run(Thread.java:662)
```

Extract thread dumps several times every hour, and check the status change of the threads to determine the problem.

Example 2: When the Processing Performance is Abnormally Slow

After acquiring thread dumps several times, find the list of threads with BLOCKED status.

```
" DB-Processor-13" daemon prio=5 tid=0x003edf98 nid=0xca waiting for monitor entry [0x000000000825f000]
java.lang.Thread.State: BLOCKED (on object monitor)
                at beans.ConnectionPool.getConnection(ConnectionPool.java:102)
                - waiting to lock <0xe0375410> (a beans.ConnectionPool)
                at beans.cus.ServiceCnt.getTodayCount(ServiceCnt.java:111)
                at beans.cus.ServiceCnt.insertCount(ServiceCnt.java:43)

"DB-Processor-14" daemon prio=5 tid=0x003edf98 nid=0xca waiting for monitor entry [0x000000000825f020]
java.lang.Thread.State: BLOCKED (on object monitor)
                at beans.ConnectionPool.getConnection(ConnectionPool.java:102)
                - waiting to lock <0xe0375410> (a beans.ConnectionPool)
                at beans.cus.ServiceCnt.getTodayCount(ServiceCnt.java:111)
                at beans.cus.ServiceCnt.insertCount(ServiceCnt.java:43)

" DB-Processor-3" daemon prio=5 tid=0x00928248 nid=0x8b waiting for monitor entry [0x000000000825d080]
java.lang.Thread.State: RUNNABLE
                at oracle.jdbc.driver.OracleConnection.isClosed(OracleConnection.java:570)
                - waiting to lock <0xe03ba2e0> (a oracle.jdbc.driver.OracleConnection)
                at beans.ConnectionPool.getConnection(ConnectionPool.java:112)
                - locked <0xe0386580> (a java.util.Vector)
                - locked <0xe0375410> (a beans.ConnectionPool)
                at beans.cus.Cue_1700c.GetNationList(Cue_1700c.java:66)
                at org.apache.jsp.cue_1700c_jsp._jspService(cue_1700c_jsp.java:120)
```

Acquire the list of threads with BLOCKED status after getting the thread dumps several times.

If the threads are BLOCKED, extract the threads related to the lock that the threads are trying to obtain.

Through the thread dump, you can confirm that the thread status stays BLOCKED because <0xe0375410> lock could not be obtained. This problem can be solved by analyzing stack trace from the thread currently holding the lock.

There are two reasons why the above pattern frequently appears in applications using DBMS. The first reason is **inadequate configurations**. Despite the fact that the threads are still working, they cannot show their best performance because the configurations for DBCP and the like are not adequate. If you extract thread dumps multiple times and compare them, you will often see that some of the threads that were BLOCKED previously are in a different state.

The second reason is the **abnormal connection**. When the connection with DBMS stays abnormal, the threads wait until the time is out. In this case, even after extracting the thread dumps several times and comparing them, you will see that the threads related to DBMS are still in a BLOCKED state. By adequately changing the values, such as the timeout value, you can shorten the time in which the problem occurs.

▼ jstack and jmap

- ***jstack -l 150759 > MyFile.tdump***

  Long listing. Prints additional information about locks such as a list of owned java.util.concurrent ownable synchronizers. **(-l IS MUST)**

  Options :-
  -l Long listing. Prints additional information about locks such as list of owned java.util.concurrent ownable synchronizers.
  -F Force a stack dump when 'jstack [-l] pid' does not respond.

-m prints mixed mode (both Java and native C/C++ frames) stack trace.
-h for help.

If we want to take the Thread dump and store in some file then we can use following command.

jstack [option] pid > fileName.txt -> .txt for storing in txt file

jstack [option] pid > fileName.tdump -> .tdump if want to open file in JVisualVM

and we can open this thread dump file in any software to get proper view.

- ***jmap -dump:live,file=heapdump.hprof 150759***

  dump:[live,] format=b, file=filename
  Dumps the Java heap in **hprof** binary format to filename. The live suboption is optional, but when specified, only the active objects in the heap are dumped. To browse the heap dump, you can use the jhat(1) command to read the generated file.

- jhat - Analyzes the Java heap. This command is experimental and unsupported

  jhat FileName

▼ Memory leakage

https://www.baeldung.com/java-memory-leaks

Symptoms

=> Performance degradation when application is continously running for long time.
=> OutOfMemoryError heap error in application when no space to create and save the objects
=> Application crashes oftenly.
=> The application is occationally running out of connection objects.

Types of Memory Leaks in Java

In any application, memory leaks can occur for numerous reasons. In this section, we'll discuss the most common ones.

Memory Leak Through *static* Fields

The first scenario that can cause a potential memory leak is heavy use of *static* variables.

In Java, ***static* fields have a life that usually matches the entire lifetime of the running application** (unless *ClassLoader* becomes eligible for garbage collection).

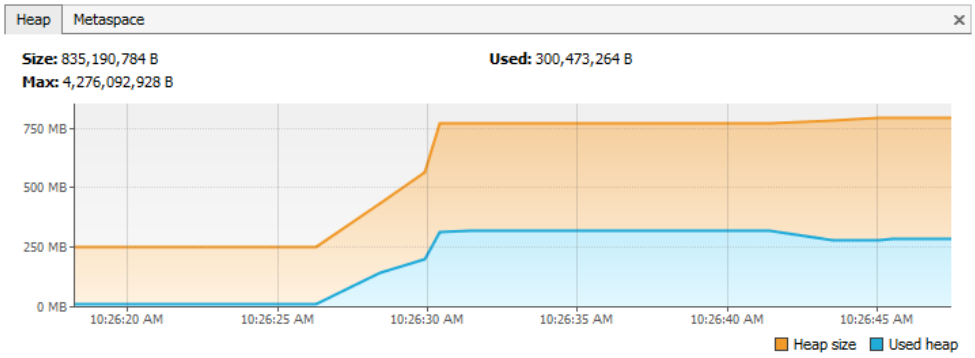
Let's create a simple Java program that populates a *static List:*

```
public class StaticTest {
public static List<Double> list =newArrayList<>();

publicvoidpopulateList() {
for (int i = 0; i < 10000000; i++) {
            list.add(Math.random());
        }
        Log.info("Debug Point 2");
    }

publicstaticvoidmain(String[] args) {
        Log.info("Debug Point 1");
newStaticTest().populateList();
```

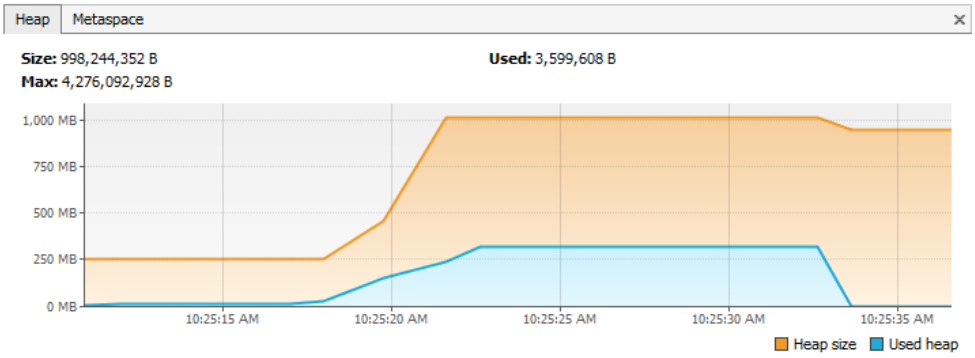
```
        Log.info("Debug Point 3");
    }
}
```

Now if we analyze the Heap memory during this program execution, then we'll see that between debug points 1 and 2, as expected, the heap memory increased.

But when we leave the *populateList()* method at the debug point 3, **the heap memory isn't yet garbage collected** as we can see in this VisualVM response:



However, in the above program, in line number 2, if we just drop the keyword *static*, then it will bring a drastic change to the memory usage, this Visual VM response shows:



The first part until the debug point is almost the same as what we obtained in the case of *static.* But this time after we leave the *populateList()* method, **all the memory of the list is garbage collected because we don't have any reference to it**.

Hence we need to pay very close attention to our usage of *static* variables. If collections or large objects are declared as *static*, then they remain in the memory throughout the lifetime of the application, thus blocking the vital memory that could otherwise be used elsewhere.

How to Prevent It?

- Minimize the use of *static* variables
- When using singletons, rely upon an implementation that lazily loads the object instead of eagerly loading

Lazy loading is a design pattern commonly used in computer programming and mostly in web design and development to defer initialization of an object until the point at which it is needed. It can contribute to efficiency in the program's operation if properly and appropriately used.

Through Unclosed Resources

Whenever we make a new connection or open a stream, the JVM allocates memory for these resources. A few examples include database connections, input streams, and session objects.

Forgetting to close these resources can block the memory, thus keeping them out of the reach of GC. This can even happen in case of an exception that prevents the program execution from reaching the statement that's handling the code to close these resources.

In either case, **the open connection left from resources consumes memory**, and if we don't deal with them, they can deteriorate performance and may even result in *OutOfMemoryError*.

How to Prevent It?

- Always use *finally* block to close resources
- The code (even in the *finally* block) that closes the resources should not itself have any exceptions
- When using Java 7+, we can make use of *try*with-resources block

Improper *equals()* and *hashCode()* Implementations

When defining new classes, a very common oversight is not writing proper overridden methods for *equals()* and *hashCode()* methods.

HashSet and *HashMap* use these methods in many operations, and if they're not overridden correctly, then they can become a source for potential memory leak problems.

How to Prevent It?

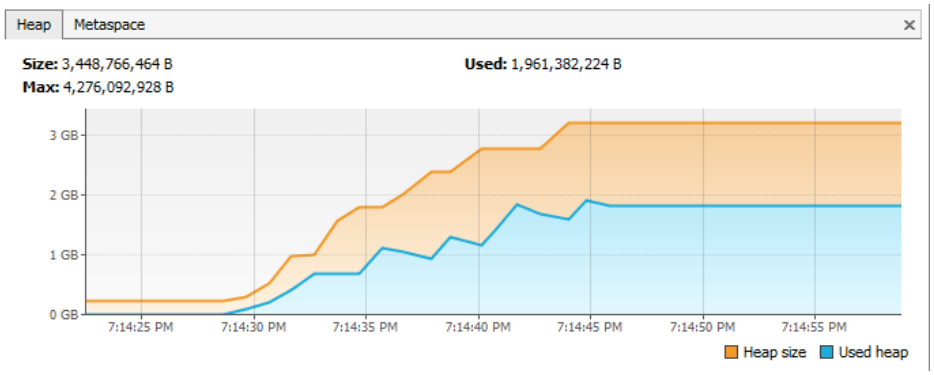
- As a rule of thumb, when defining new entities, always override *equals()* and *hashCode()* methods
- It's not just enough to override, but these methods must be overridden in an optimal way as well

Inner Classes That Reference Outer Classes

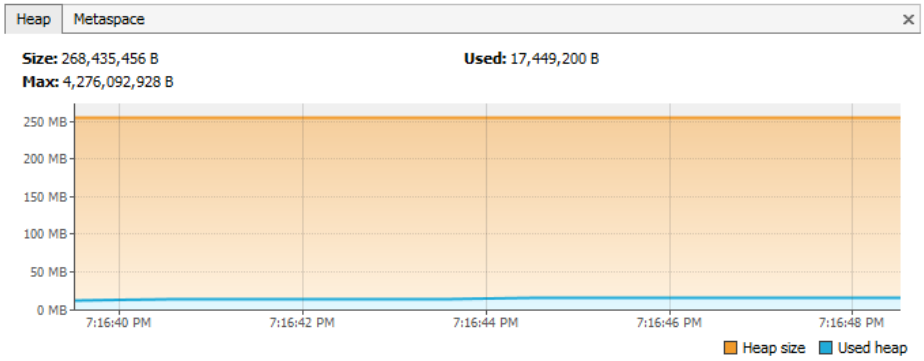
This happens in the case of non-static inner classes (anonymous classes). For initialization, these inner classes always require an instance of the enclosing class.

Every non-static Inner Class has, by default, an implicit reference to its containing class. If we use this inner class' object in our application, then **even after our containing class' object goes out of scope, it will not be garbage collected**.

Consider a class that holds the reference to lots of bulky objects and has a non-static inner class. Now when we create an object of just the inner class, the memory model looks like:



However, if we just declare the inner class as static, then the same memory model looks like this:



This happens because the inner class object implicitly holds a reference to the outer class object, thereby making it an invalid candidate for garbage collection. The same happens in the case of anonymous classes.

How to Prevent It?

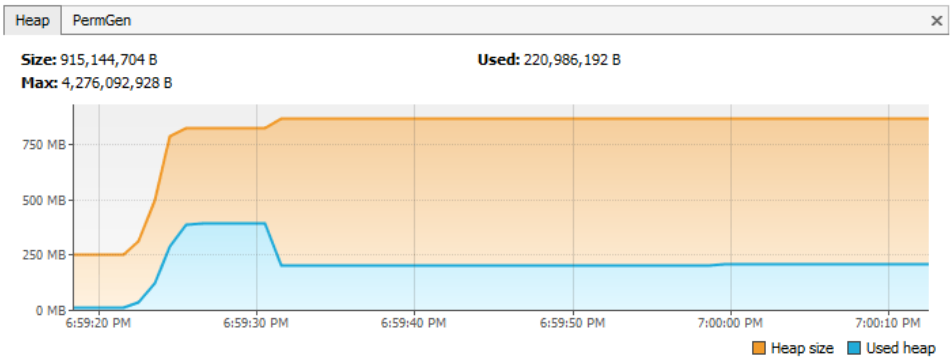
- If the inner class doesn't need access to the containing class members, consider turning it into a *static* class

Through *finalize()* Methods

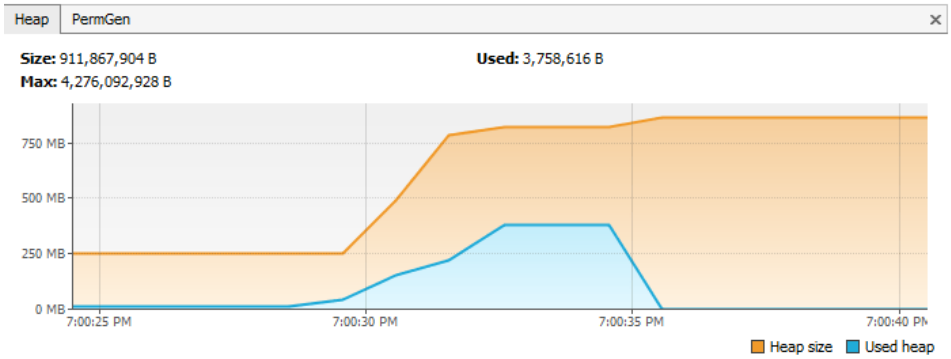
Use of finalizers is yet another source of potential memory leak issues. Whenever a class' *finalize()* method is overridden, then **objects of that class aren't instantly garbage collected.** Instead, the GC queues them for finalization, which occurs at a later point in time.

Additionally, if the code written in *finalize()* method is not optimal and if the finalizer queue cannot keep up with the Java garbage collector, then sooner or later, our application is destined to meet an *OutOfMemoryError*.

To demonstrate this, let's consider that we have a class for which we have overridden the *finalize()* method and that the method takes a little bit of time to execute. When a large number of objects of this class gets garbage collected, then in VisualVM, it looks like:



However, if we just remove the overridden *finalize()* method, then the same program gives the following response:



How to Prevent It?

- We should always avoid finalizers

Using *ThreadLocal*s

ThreadLocal (discussed in detail in Introduction to *ThreadLocal* in Java tutorial) is a construct that gives us the ability to isolate state to a particular thread and thus allows us to achieve thread safety.

When using this construct, **each thread will hold an implicit reference to its copy of a *ThreadLocal* variable and will maintain its own copy, instead of sharing the resource across multiple threads, as long as the thread is alive.**

Despite its advantages, the use of *ThreadLocal* variables is controversial, as they are infamous for introducing memory leaks if not used properly. Joshua Bloch once commented on thread local usage:

> "Sloppy use of thread pools in combination with sloppy use of thread locals can cause unintended object retention, as has been noted in many places. But placing the blame on thread locals is unwarranted."

Memory leaks with *ThreadLocals*

ThreadLocals are supposed to be garbage collected once the holding thread is no longer alive. But the problem arises when *ThreadLocals* are used along with modern application servers.

Modern application servers use a pool of threads to process requests instead of creating new ones (for example the *Executor* in case of Apache Tomcat). Moreover, they also use a separate classloader.

Verbose Garbage Collection

By enabling verbose garbage collection, we're tracking detailed trace of the GC.

Strategies to deal with Memory Leaks

=> Enable Profiling :- By using different tools like JVisualVM we can check the memory leaks
=> Verbose GC :- By enabling verbose garbage collection (verbose:gc) we can track the detailed trace of the GC.
=> Use Refrence objects to avoid memory leaks.

Conclusion

In layman's terms, we can think of memory leak as a disease that degrades our application's performance by blocking vital memory resources. And like all other diseases, if not cured, it can result in fatal application crashes over time.

Memory leaks are tricky to solve and finding them requires intricate mastery and command over the Java language. **While dealing with memory leaks, there is no one-size-fits-all solution, as leaks can occur through a wide range of diverse events.**

However, if we resort to best practices and regularly perform rigorous code walk-throughs and profiling, then we can minimize the risk of memory leaks in our application.

wait: thread has a task to accomplish, for that it needs some resource or event to happen, so thread will put itself into wait state, till it gets the input

park: thread doesnt have any task to accomplish, so its just sitting idle, as soon as it gets an input it will exit the parked state

finished:

sleeping: will return after sleep time