**JConsole**

By –

Madhavilatha.Kolla@cognizant.timeinc.com

Contents

[1. About jConsole: 3](#_Toc485656616)

[2. Invoking Methods: 3](#_Toc485656617)

[3. The jconsole interface 4](#_Toc485656618)

[3A.VMSummary tab: 4](#_Toc485656619)

[3b.Memory tab: 5](#_Toc485656620)

[Heap and Non-heap Memory 6](#_Toc485656621)

[Memory Pools and Memory Managers 6](#_Toc485656622)

[Garbage Collection 7](#_Toc485656623)

[3c. Thread tab: 8](#_Toc485656624)

[3d. Classes tab: 8](#_Toc485656625)

[3e. MBeans tab: 8](#_Toc485656626)

# About jConsole:

**JConsole** is a graphical monitoring tool to monitor [Java Virtual Machine](https://en.wikipedia.org/wiki/Java_Virtual_Machine) (JVM) and Java applications both on a local or remote machine.

JConsole uses underlying features of [Java Virtual Machine](https://en.wikipedia.org/wiki/Java_Virtual_Machine) to provide information on performance and resource consumption of applications running on the Java platform using [Java Management Extensions](https://en.wikipedia.org/wiki/Java_Management_Extensions) (JMX) technology. JConsole comes as part of [Java Development Kit](https://en.wikipedia.org/wiki/Java_Development_Kit) (JDK) and the graphical console can be started using "jconsole" command.

# Invoking Methods:

**JConsole** can be invoked in two ways –

1. From Java Bin folder , Go to C:\Program Files\Java\jdk1.8.0\_111\bin and click on “jConsole”
2. Using command prompt ,

cmd->cd C:\Program Files\Java\jdk1.8.0\_111\bin-> jconsole (OR)

cmd->cd C:\Program Files\Java\jdk1.8.0\_111\bin-> jconsole PID

You can use jconsole to monitor both local applications (those running on the same system as jconsole) and remote applications (those running on other systems).

**How to Know the PID of application?**

* Invoke cmd prompt -> tasklist (It lists all tasks running on the server)
* Select the appropriate task name followed by process ID.

Ex: java.exe (PID- 5325)

Note: If we don't specify a process ID, jconsole will automatically detect all local Java applications, and display a dialog box that lets you select the one you want to monitor (see the next section).

**Jconsole GUI-**

#### 

**Remote Monitoring**

To connect to remote monitoring,

* Enter the HostName:POrtNumber in the GUI with valid credentials (OR)
* Invoke CMD-> jconsole [hostName:portNum]

# 3. The jconsole interface

The jconsole interface is composed of six tabs:

* **Overview tab:** displays information on the JVM.
* **Memory tab:** displays information on memory use.
* **Threads tab:** displays information on thread use.
* **Classes tab:** displays information on class loading.
* **VMSummary tab:** displays summary information on the JVM and monitored values.
* **MBeans:** displays information on MBeans.

## 3A.VMSummary tab:

* Uptime: how long the JVM has been running
* Total compile time: the amount of time spent in just-in-time (JIT) compilation.
* Process CPU time: the total amount of CPU time consumed by the JVM

Threads

* Live threads: Current number of live daemon threads plus non-daemon threads
* Peak: Highest number of live threads since JVM started.
* Daemon threads: Current number of live daemon threads
* Total started: Total number of threads started since JVM started (including daemon, non-daemon, and terminated).

Memory

* Current heap size: Number of Kbytes currently occupied by the heap.
* Committed memory: Total amount of memory allocated for use by the heap.
* Maximum heap size: Maximum number of Kbytes occupied by the heap.
* Objects pending for finalization: Number of objects pending for finalization.
* Garbage collector information: Information on GC, including the garbage collector names, number of collections performed, and total time spent performing GC.

Classes

* Current classes loaded: Number of classes currently loaded into memory.
* Total classes loaded: Total number of classes loaded into memory since the JVM started, included those subsequently unloaded.
* Total classes unloaded: Number of classes unloaded from memory since the JVM started.

Operating System

* Total physical memory: Amount of random-access memory (RAM) that the OS has.
* Free physical memory: Amount of free RAM the OS has.
* Committed virtual memory: Amount of virtual memory guaranteed to be available to the running process.

## 3b.Memory tab:

The Memory tab provides information on memory consumption and memory pools.

The Chart under Memory tab JVM's memory use versus time, for heap and non-heap memory, and for specific memory pools. The memory pools available depend on the JVM being used. For the HotSpot JVM, the pools are:

* Eden Space (heap): pool from which memory is initially allocated for most objects.
* Survivor Space (heap): pool containing objects that have survived GC of eden space.
* Tenured Generation (heap): pool containing objects that have existed for some time in the survivor space.
* Permanent Generation (non-heap): holds all the reflective data of the virtual machine itself, such as class and method objects. With JVMs that use [class data sharing](https://www.cs.mun.ca/java-api-1.5/guide/vm/class-data-sharing.html), this generation is divided into read-only and read-write areas.
* Code Cache (non-heap): HotSpot JVM also includes a "code cache" containing memory used for compilation and storage of native code.

The Details area shows several current memory metrics:

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

The bar chart at the lower right shows 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.

### Heap and Non-heap Memory

The JVM manages two kinds of memory: heap and non-heap memory, both created when it starts.

Heap memory is the runtime data area from which the JVM allocates memory for all class instances and arrays. The heap may be of a fixed or variable size. The garbage collector is an automatic memory management system that reclaims heap memory for objects.

Non-heap memory includes a method area shared among all threads and memory required for the internal processing or optimization for the JVM. It stores per-class structures such as a runtime constant pool, field and method data, and the code for methods and constructors. The method area is logically part of the heap but, depending on implementation, a JVM may not garbage collector compact it. Like the heap, the method area may be of fixed or variable size. The memory for the method area does not need to be contiguous.

In addition to the method area, a JVM 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 JVM code for high performance.

### Memory Pools and Memory Managers

Memory pools and memory managers are key aspects of the JVM memory system.

A memory pool represents a memory area that the JVM manages. The JVM has at least one memory pool and it may create or remove memory pools during execution. A memory pool can belong to either heap or non-heap memory.

A memory manager manages one or more memory pools. The garbage collector is a type of memory manager responsible for reclaiming memory used by unreachable objects. A JVM may have one or more memory managers. It may add or remove memory managers during execution. A memory pool can be managed by more than one memory manager.

### Garbage Collection

Garbage collection (GC) is how the JVM 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 un-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.

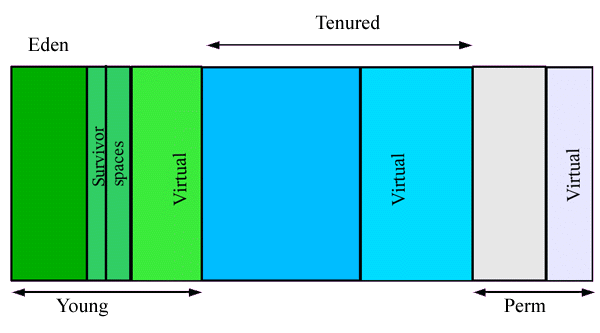
The HotSpot VM garbage collector uses generational garbage collection. Generational GC takes advantage of the observation that, in practice, most programs create:

* Many objects that have short lives (for example, iterators and local variables).
* some objects that have very long lifetimes (for example, high level persistent objects)

So, generational GC divides memory into several generations, and assigns each a memory pool. When a generation uses up its allotted memory, the VM performs a partial garbage collection (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 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 this:



As explained in the following documents, if the garbage collector has become a bottleneck, you can improve performance by customizing the generation sizes. Using jconsole, explore the sensitivity of your performance metric to the garbage collector parameters

## 3c. Thread tab:

The Threads tab provides information on thread use.

The Threads list in the lower left lists all the active threads. If you enter a string in the Filter field, the Threads list will show only those threads whose name contains the string you enter. Click on the name of a thread in the Threads list to display information about that thread to the right, including the thread name, state, and stack trace.

The chart shows the number of live threads versus time. Three lines are shown:

* Magenta: total number of threads
* Red: peak number of threads
* Blue: number of live threads.

## 3d. Classes tab:

The Classes tab displays information on class loading.

The graph plots the number of classes loaded versus time:

* Red line is the total number of classes loaded (including those subsequently unloaded).
* Blue line is the current number of classes loaded.

The Details section at the bottom of the tab displays the total number of classes loaded since the JVM started, the number currently loaded and the number unloaded.

## 3e. MBeans tab:

The MBean tab displays information on all the MBeans registered with the platform MBean server.

The tree on the left shows all the MBeans, organized according to their objectNames. When you select an MBean in the tree, its attributes, operations, notifications and other information is displayed on the right.