

Excelsior Knowledge Base

HOWTO: Fine tune application memory footprint

Article ID: 000025

Last Revised On: 27-Sep-2005

The information in this article applies to:

- Excelsior JET 4.0 and above

The similar article for Excelsior JET 3.7 and below can be found here [000002](#).

SUMMARY

Excelsior JET lets you define the policy of memory management to be used for your Java application at run time. This article describes a few simple mechanisms that you can employ to set an optimal balance between performance of the application and its memory footprint.

The sample cited in this article illustrates behavior of a model application with respect to memory consumption. We examine that behavior and show how to change it by adjusting the *Maximum heap size* and *GC ratio* parameters of the JET Runtime. The behavior of memory manager when the running application lacks enough physical memory is also discussed.

Source files and build/run scripts for this article may be found [here](#).

DETAILS

Excelsior JET memory manager has the following configuration parameters.

GC ratio

Ideally, an application should work fast enough and consume as few memory as possible. In practice, there are always trade-offs between application performance (or throughput) and its memory footprint. What's especially important is that performance and memory requirements are application-specific. For one program, performance is critical, whereas low memory footprint has more value for the other. Put simply, one size does not fit all.

Fortunately, the JET Runtime lets you fine tune the performance/memory balance for your Java application. For that, you just set a ratio in which the Runtime will portion out execution time between the two tasks: *execution of the application's code* and *garbage collection*. This technique is simple and effective:

- By assigning GC a greater time share, you increase the frequency of GC invocations which does not allow the Java heap to excessively grow in size.
- And vice versa, by lowering the ratio of GC in execution time, you reduce the GC overhead thereby improving performance at the cost of higher memory consumption.

You can do that by specifying the *GC ratio* setting to the JET Runtime. Its value is the tenths of percent of total CPU time that may be spent for garbage collection. For example, the default GC ratio is 1.1% which means that GC will normally take the percentage in total execution time. You can vary the GC ratio as follows.

- Using the JET Control Panel: adjust the **GC ratio** control on the **Options** page; or
- Set the system property `jet.gc.ratio`. For example, to set the ratio to 5.0%, specify `-Djet.gc.ratio:50`

The maximum GC ratio you can set is 40.0%.

Note: changing the default GC ratio is available only in the Professional Edition of Excelsior JET.

The default setting (1.1%) is biased to performance and, therefore, may cause excessive memory consumption in some cases. You can experiment with your Java application by fine tuning the GC ratio. Often, increasing the value by several units may noticeably reduce memory footprint. That's why the unit of measure is 1/10th of percent not a whole percent.

One might question: what reduction in memory footprint can I expect when increasing the GC ratio by X%? In general, it depends on two factors: *how much garbage* your application produces and *how fast* it does that. Thus, the answer is specific to a particular application.

Note that under certain circumstances, the JET Runtime is unable to meet the GC ratio requirement. For example, if a Java application allocates objects very intensively and amount of available physical memory is low, more CPU time may be spent in the garbage collector regardless of the ratio specified.

Finally, you may set the GC ratio to 0 to entirely disable the mechanism (it's not recommended, though.) In such case, GC is automatically invoked only if the JET Runtime is unable to serve a regular request for object allocation due to memory constraints as described in the following sections.

Maximum heap size

You may specify the maximum amount of heap memory that your application is allowed to allocate at run time¹:

- Using the Control Panel: adjust the **Maximum Heap Size** control on the **Options** page
- In the project file: use the HEAPLIMIT equation:

```
-HEAPLIMIT=30m    % maximum heap size is 30MB
```

To override the compile-time setting, you may set the system property `jet.gc.heaplimit`, e.g.

```
-Djet.gc.heaplimit:35m
```

You need to specify the maximum heap size only if you do not want the Java heap to exceed certain upper bound in any case. On the one hand, it may be useful to reduce application memory footprint. On the other hand, it may compromise performance of your application. For example, it might work more effectively when using large Java heaps on the target machines that have high RAM capacity.

The default value of the maximum heap size is 0 which means *Adaptive*. In this mode, the Java heap is only bounded by the size of Virtual Address Space that varies from 2 to 4GB, depending on the underlying operating system. But do not be afraid. It does not mean that your application will take gigabytes of memory at run time. First, the JET Runtime checks the GC ratio which effectively damps inflation of the heap. Moreover, the runtime constantly monitors the amount of *available physical memory* as described in the next section.

¹The respective command line parameter of the standard Java launcher is `-Xmx`

Handling low memory

As a rule, it's hard to determine the "right" application memory settings because they are specific to a particular target system. For instance, you consider a box equipped with 256MB of RAM as a "typical target system" but some customers are proved to be still using 128MB machines. Another example is a system under heavy workload: imagine that your application runs simultaneously with a couple of other programs hungry for memory. Anyway, your application must work, more or less efficiently.

To address these problems, the JET Runtime considers the amount of *available* physical memory in the system to schedule garbage collection. If low memory is detected when allocating an object, GC is activated to avoid unnecessary swapping RAM pages to the hard disk. Of course, it leads to more frequent GC invocations and penalizes application performance. But it's a good alternative to tremendous overheads of swapping memory to and fro.

However, if the runtime detects that application execution has degraded to sole garbage collection or it's unable to allocate an object even after GC invocation, the physical memory boundary is crossed. The JET Runtime augments the Java heap with the amount of memory enough to serve the allocation request plus some decent gap necessary for further operation. Often, this move helps to clear the air. The operating system reacts to the event and fetches memory, e.g. from the system cache, so that the application continues running smoothly.

You have control over the JET Runtime with respect to crossing the physical memory threshold. To override the default behavior

- Using the Control Panel: On **Options** page, select **throw OutOfMemoryError** from the **Handling low memory** combo box; or
- Specify the system property `-Djet.gc.no.swap`.

As a result, `OutOfMemoryError` will be thrown instead of using virtual memory to increase the heap. This mode is useful if you suspect that your application has performance issues due to not enough memory in the system. You may easily prove or disprove that by disabling the default policy and handling `OutOfMemoryError` exceptions, if any.

Default settings:

- The default value of **ratio** is 11 (meaning 1.1% of execution time).
- The default **maximum heap size** is 0 (meaning "adaptive").
- By default, application uses virtual memory if there is not enough physical memory.

Tweaking Maximum heap size and GC ratio

Let us consider an extreme case. This simple program constantly creates new heap objects most of which become unreachable (to be garbage collected):

```
public class MemUse {  
  
    static String[] roots = new String[100];  
  
    static public void main (String args[]) {  
        int i = 0;
```

```

    while (true) {
        roots[i % 100] = new String("new Object");
        i++;
    }
}

```

In the main method, new objects are allocated and assigned to elements of the `roots` array in an infinite loop. Thus, at any time during sample run there are at most 100 alive objects created by the program in the heap. Other allocated objects are considered garbage and should be reclaimed by the GC when it starts.

Simplicity of this sample does not prevent revealing the changes in the JET memory manager behavior when varying values of the parameters. For different combinations of the parameters, we monitored memory consumption via the "Peak Mem Usage" column of Windows Task Manager (that includes all memory used by the process, not only the heap.) Here go the results.

First, we set Maximum heap size to adaptive and experiment with different values of GC ratio.

Ratio, %	11 (1,1%)	50 (5%)	60 (6%)	300 (30%)
Memory usage, MB	404.6	75.1	36.5	10.1

Table 1: Maximum heap size is adaptive

The "Memory usage" row of Table ?? shows that the default value of 1.1% is definitely too low for this particular application. 6% may be enough, whereas 20% would keep memory usage to a minimum. Note that the memory usage value 404.6MB in the second column is close to the amount of *available* physical memory in the system (we used a 512MB box for experimenting.)

Then, we set the maximum heap size to 50MB, and tried the same values of GC ratio as shown in Table ??.

Ratio, %	11 (1,1%)	50 (5%)	60 (6%)	300 (30%)
Memory usage, MB	56.9	56.9	36.5	10.1

Table 2: Maximum heap size is set to 50MB

It is easy to see that the restriction effectively reduced memory footprint (compare second and third columns of the two tables). Note also that if GC ratio is set to 6% or 20%, the specified maximum heap size is never reached so the results in both tables are the same.