

# Excelsior Knowledge Base

## HOWTO: Fine tune application memory consumption (Excelsior JET 3.7 and below)

Article ID: 000002

Last Revised On: 27-Sep-2005

The information in this article applies to:

- Excelsior JET version 3.0 through 3.7

This article **does not** apply to Excelsior JET 4.0 and above.

The similar article for Excelsior JET 4.0 and above can be found here [000025](#).

### SUMMARY

It's quite complicated if at all possible to configure a Java memory manager following the "one size fits all" principle. Applications consume memory in different ways and have different requirements to performance and memory footprint. The behavior of the memory manager and the garbage collector in particular is therefore application-specific.

The first 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 "*heaplimit*" and "*ratio*" parameters of the JET runtime system.

The more complicated second sample help us to assess the effect of heap defragmentation on memory consumption and performance.

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

### DETAILS

Excelsior JET memory manager has the following configuration parameters:

#### **heaplimit**

Specifies the maximum amount of heap memory available to your application at run time (The respective command line parameter of the standard Java launcher is `-Xmx`.)

This parameter is set at compile time, but may be overridden using the `jet.gc.heaplimit` property.

#### **ratio**

The amount of 1/1000ths of total CPU time that may be spent in the garbage collector. The maximum value is 400 (meaning 40%). Setting **ratio** to zero disables periodic garbage collection (the GC is invoked only when the `heaplimit` is reached), which was the only option in Excelsior JET prior to 3.0.

This parameter is set using the `jet.gc.ratio` property and is used to fine tune heap optimization.

## defragment

A boolean flag used to force heap defragmentation after each garbage collection cycle. See Heap defragmentation below for more information.

This flag is set using the `jet.gc.defragment` property.

### Default settings:

- The default value of **ratio** is 11 (meaning 1.1% of CPU time).
- The default **heaplimit** is 0 (meaning adaptive, see below).
- By default, defragmentation after each GC cycle is disabled. It only occurs if the GC fails to free enough memory.

## Heaplimit and ratio

Consider a simple program that constantly creates new heap objects:

```
public class MemUse
{
    String[] roots = new String[100];

    public void createBlocks(){
        int i = 0;
        while (true) {
            i = (i+1) % 100;
            roots[i] = new String("new Object");
        }
    }

    static public void main (String args[])
        throws Exception
    {
        MemUse obj = new MemUse();
        obj.createBlocks();
    }
}
```

An instance of the `MemUse` class contains only one field `roots`, which is an array of type `String`. In the method `createBlocks()`, new objects are allocated and assigned to the array's elements in an infinite loop. Thus, at any time during sample run there are at most 100 alive objects in the heap. Other objects allocated by the sample 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 values of parameters are changed.

For different combinations of the parameters, we monitored memory consumption via the "Mem Usage" column of Windows Task Manager (that includes all memory used by the process, not just the heap). The following table contains some sample results:

**Table: Maximum memory usage, MB**

Heaplimit \ Ratio	0 (disabled)	11 (1,1%)	50 (5%)	200 (20%)
10	13.3	12.2	12.6	6.1
20	22.0	23.0	15.3	5.4
60	63.6	64.6	15.8	5.4

As you can see, if **ratio** is high enough, the garbage collector is invoked so frequently that **heaplimit** is never reached. The default value of 1.1% is definitely too low for this particular application. 5% may be enough, whereas 20% would keep memory usage to a minimum.

Setting **heaplimit** value to 0 switches the memory manager to the “*adaptive*” mode. In this mode, the JET runtime uses not more than 75% of available physical memory. If there’s not enough physical memory, `OutOfMemoryError` will be thrown. So if you want your application to make use of virtual memory, you should specify an explicit value of the **heaplimit** parameter. In this case, however, memory paging would occur if there is no available physical memory left, increasing GC pauses by orders of magnitude.

**Note:** Since the behavior of the memory manager and the garbage collector in particular is application-specific, increasing **ratio** does not necessarily result in lower memory consumption, and decreasing it does not always improve application performance. You have to experiment with your particular application to find the optimum **ratio** for it.

By changing the **heaplimit** value, you may either improve application performance or deteriorate it. In general, it is recommended to use adaptive heaplimit when you know that your application will work “alone” or that the target systems will have enough physical memory. You have to experiment with your particular application to find the best values for heaplimit and ratio with respect to performance and memory consumption.

## Heap defragmentation

Consider the `DemoSort` sample program, which sorts an array of objects using four different sort algorithms.

We intentionally “unoptimized” those algorithms by changing the method `swap()` as follows:

```
void swap(Item a[], int i, int j)
{
    Item swap;
    swap = new Item(a[i]);
    a[i] = new Item(a[j]);
    a[j] = new Item(swap);
}
```

The tables below demonstrate the impact of memory manager settings on performance and memory consumption of this sample:

auto defragmentation									
Heaplimit \ Ratio	11(1,1%)			50 (5%)			200 (20%)		
	used	total	time	used	total	time	used	total	time
10	9.08	9.43	1.922	7.57	7.95	1.913	3.03	3.38	2.153
20	16.17	16.51	1.873	8.06	8.53	1.903	3.12	3.41	2.213
60	23.12	23.59	1.853	9.07	9.54	1.913	3.03	3.38	2.164

**forced defragmentation**

Heaplimit \ Ratio	11(1,1%)			50 (5%)			200 (20%)		
	used	total	time	used	total	time	used	total	time
10	9.17	9.17	1.973	7.53	7.90	2.033	2.62	2.98	2.454
20	16.22	16.51	1.943	7.35	7.72	2.043	2.29	2.64	2.563
60	20.41	20.86	1.913	7.53	7.90	2.033	2.39	2.74	2.553

As can be seen, forcing defragmentation significantly reduces memory consumption but also deteriorates performance.

In most cases, forcing defragmentation will increase GC pauses and thus slow down your application. However, under certain circumstances, e.g. on systems with low RAM capacity, it may actually **improve** overall performance. Also, the changes in memory consumption and performance are application-specific, so it is up to you to experiment and then decide whether frequent heap defragmentation is beneficial to your application.