



# Explore Java Memory Management

IBM Cognos BI 10.2.2

Business Analytics software

© 2015 IBM Corporation





## Objectives

- At the end of this module, you should be able to:
  - describe Java memory layout
  - manage Java memory
  - use tools to monitor Java memory



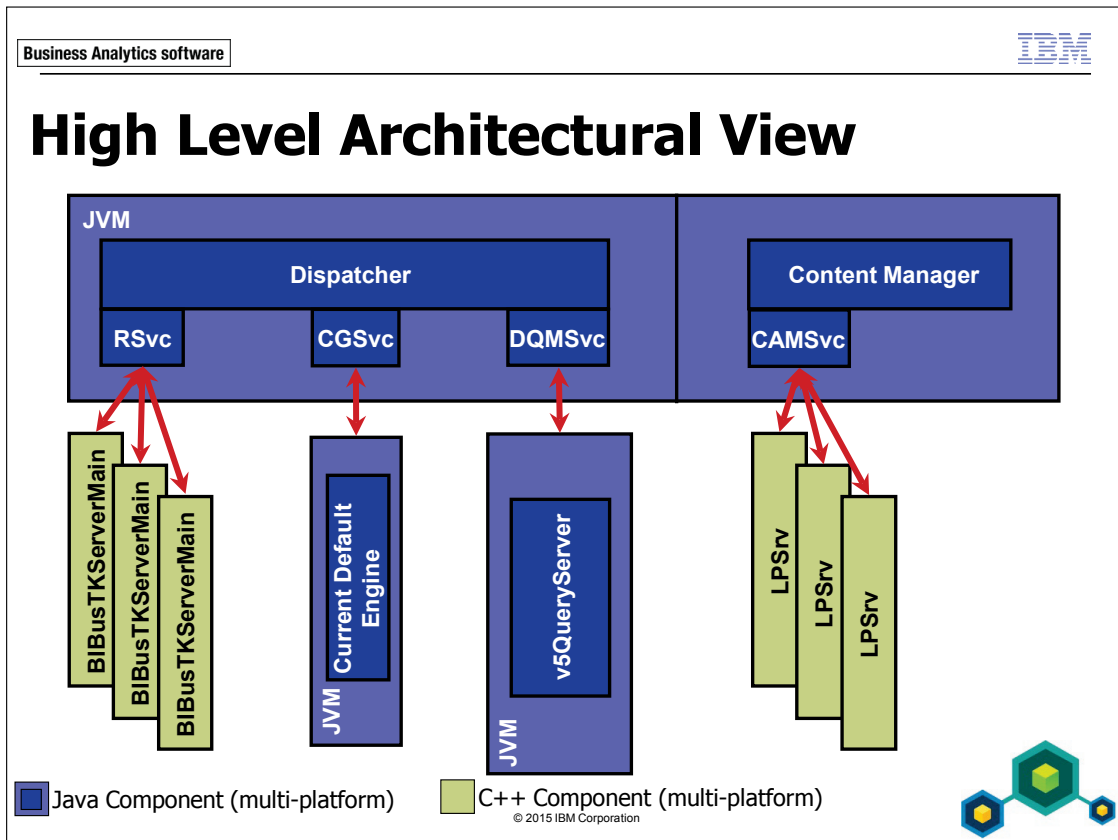
## Review Java Components in IBM Cognos

- IBM Cognos 10 is a mix of Java and C++
- new as of 10.2.0: Cognos Graphic Service (CGS), Dynamic Query Mode (DQM)
- changed as of 10.2.0: Cognos Access Manager (CAM), Content Manager
- legacy: Content Manager (CM), Dispatcher (DISP)

© 2015 IBM Corporation



IBM Cognos is moving to more Java components over time.



This high level example shows that even a simple install runs three instances of JRE with IBM Cognos 10. This is one reason why Java is important to understand.

## What is Java?

- an interpreted, object oriented programming language
- develop once, deploy anywhere
- low cost code maintenance, single code stream for all platforms
- standardized
- optimizable at JRE level with Just-In-Time compilation (JIT)

© 2015 IBM Corporation



Java offers a modular structure of code, with objects and classes, where the same source code can be used for all targets. It can be compiled to platform independent bytecode, such as .class files which are sometimes zipped into .jar files. Bytecode is not executable, and requires a run-time environment. There is no direct control of hardware, and Java needs more resources than machine code.

## Describe the Java Runtime Environment

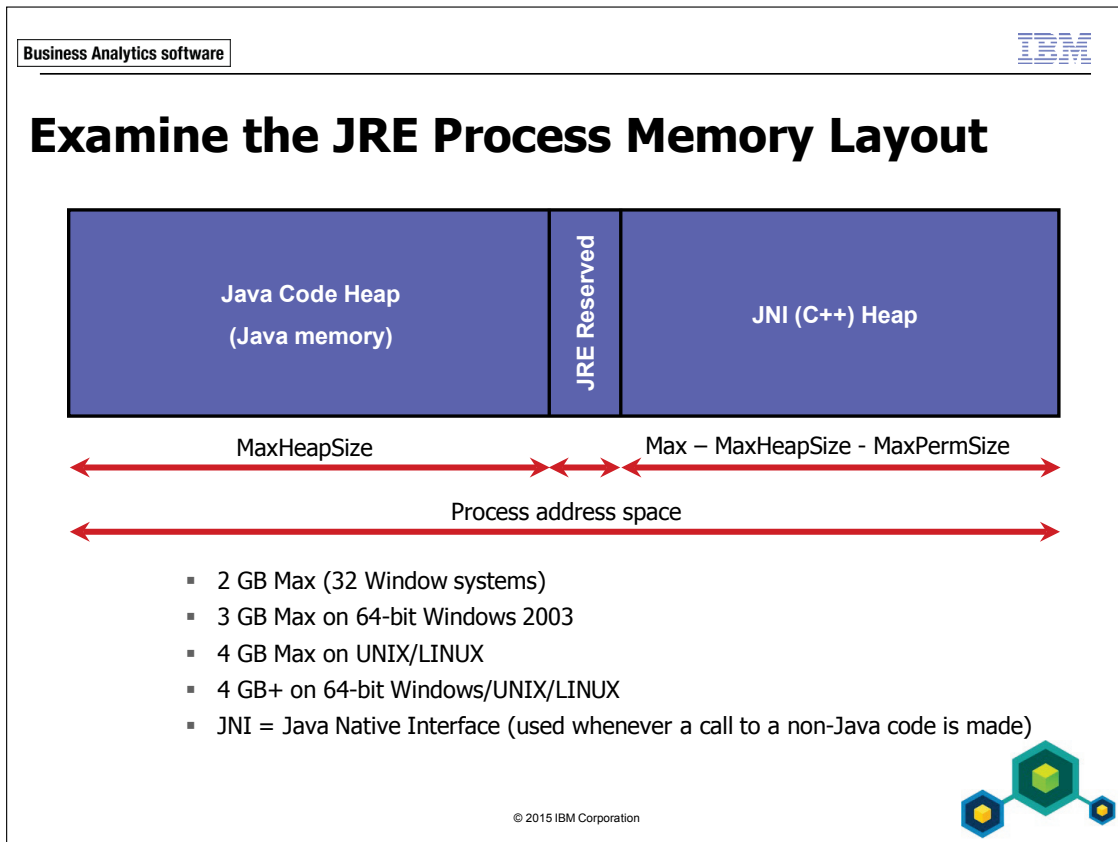
- platform dependent, usually machine code
- implemented by different OS vendors typically
- abstracts hardware specifics to developer
- implements memory management
- responsible for optimizations (JIT)
- enables parallelizing threads by mapping to CPU architecture

© 2015 IBM Corporation



Memory management (allocation and garbage collection) can be the most critical configuration of a JRE. The quality and configuration of the JRE directly impacts Java code performance and stability. Good optimization and memory management can overlay bad code design, resulting in increased performance for the Java applications.

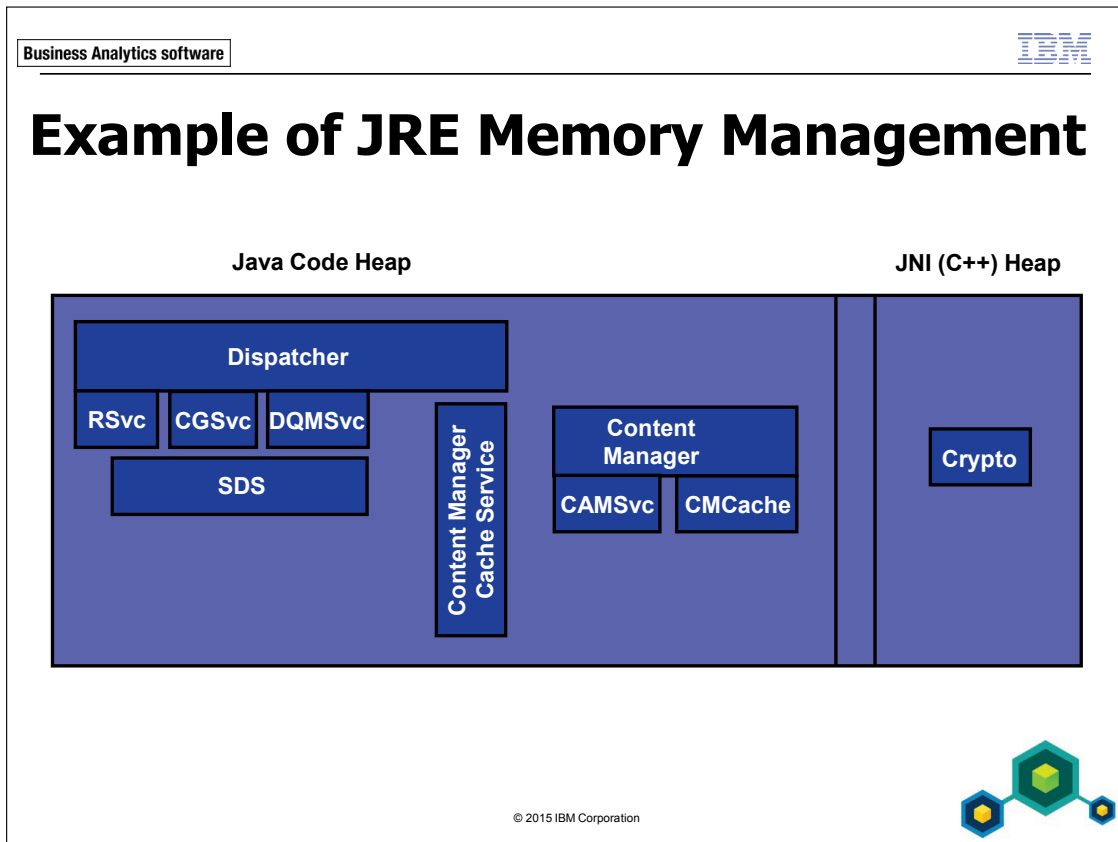
Bad memory management will negate good code design.



A JRE uses a configurable amount of memory within the allowed address space for a single process on the particular hardware and operating system. This amount is passed to the JRE upon start in a parameter called `-Xmx` (`MaxHeapSize`) and is usually less than the theoretical maximum. Additionally, a smaller amount is reserved for the JRE itself.

The `MaxHeapSize` works much like a slider control; more Java heap means less JNI heap.





IBM Cognos 10 uses both heaps, therefore the MaxHeapSize still has a direct impact, however more Java heap is favorable.

Separating install components to separate instances of application servers helps to manage memory requirements for Java heap, and in particular for Content Manager. Having multiple instances of JRE will help to leverage large RAM pools on 32-bit JREs, and using 64-bit JREs enables even larger Java heaps.

## Why is Java Memory Management Important?

- defining amount of memory is not sufficient configuration
- JRE memory management handles allocation and freeing memory in Java heap
- JRE interfaces with OS level memory management for optimum performance

© 2015 IBM Corporation



It is important to understand how memory is managed in a JRE in detail because this significantly impacts the stability and performance of both the JRE and the IBM Cognos 10 components.

In Java, memory is allocated implicitly when instantiating an object, and there is no concept of freeing memory or releasing objects explicitly. In addition, the JRE has to interface with OS level memory management for optimum performance. Allocating and freeing memory is expensive, and OS level memory management concepts like virtual memory (paging) can interfere with the JRE memory management.

## Explore Concepts of Memory Management

- Garbage Collection (GC)
  - allocates and frees memory for Java objects within the Java heap but not in the JNI heap
  - GC implementation details are not part of any specification, only the concept is
  - each JRE vendor implements GC differently
- Virtual Memory
  - memory is reserved from the OS but not committed
  - committed memory dynamically changes due to GC
  - reserved memory is the maximum JRE can allocate
  - enable by specifying `-Xms (MinHeapSize)` parameter to JRE
  - disabled if `-Xms >= -Xmx`.

© 2015 IBM Corporation



A JRE implements two concepts for memory management, garbage collection and virtual memory. The combination of the two concepts leads to things like a JRE process reserving 786MB on Win32 but depending on load and components running in that JRE, the committed size may be considerably less.

For more information on GC and memory, go to <http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnosis/GCandMemory-042005.pdf>.

For more information on IBM JVM, go to <http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnosis/diag142.pdf>.


Business Analytics software

IBM

## JVM Settings

- starting points for 50+ concurrent users include:
  - set gcpolicy=gencon to minimize the long pauses for gc

© 2015 IBM Corporation

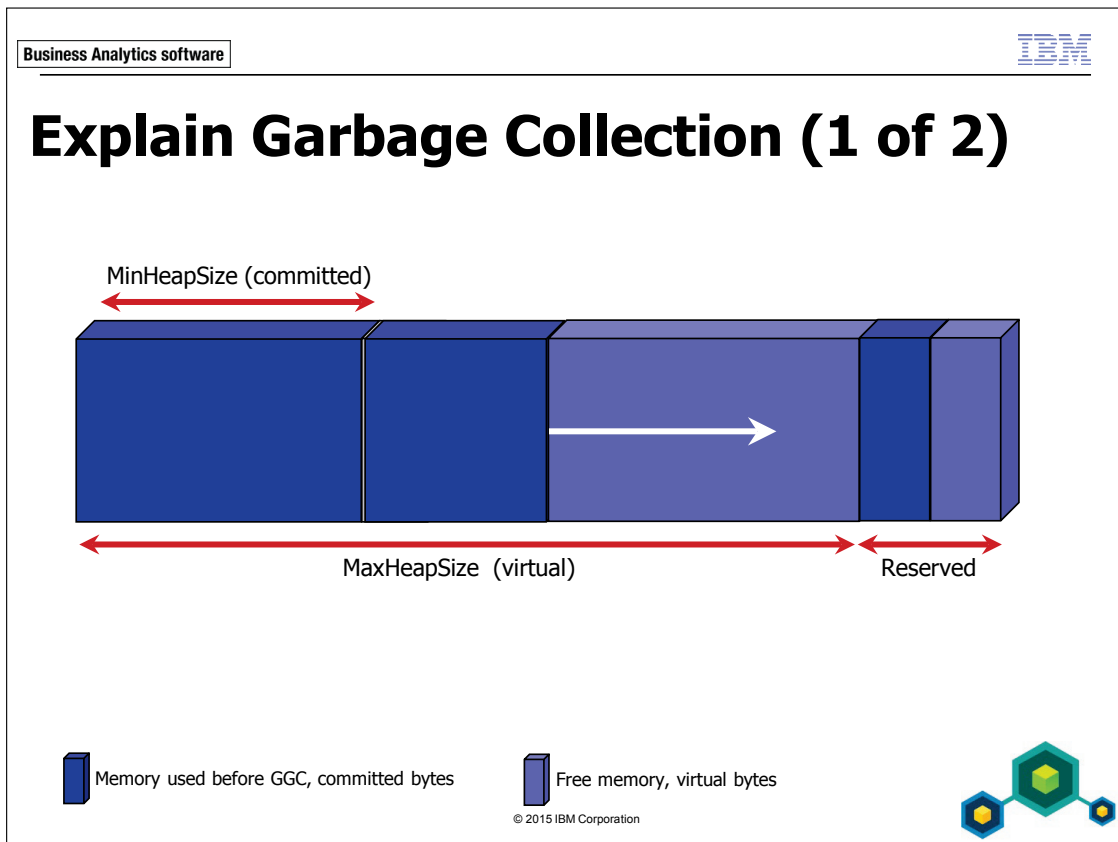


Along with the setting in the slide, consider the following component-specific starting point settings for JVM:

- Dispatcher IBM JVM (Report Server)
  - 64 bit: xms=xmx= 4GB and xmn set to approx. 1/3 of xmx, or 1286MB
  - 32 bit: xms=xmx= 1024MB and xmn set to approx. 1/3 of xmx, or 341MB
- Dispatcher IBM JVM (Content Manager, or combined CM+RS)
  - 64 bit: xms=xmx= 6 GB and xmn set to approx. 1/3 of xmx, or 2048MB
  - 32 bit: xms=xmx= 768MB and xmn set to approx. 1/3 of xmx, or 256MB

- Graphics Service IBM JVM (IBM Cognos BI v10 only)
  - Unix: <cog\_install>/bin/cgsServer.sh
  - Change "Xmx1g" (default) to "Xmx2g" for 32 bit or "Xmx4g" for 64 bit
- Dynamic query data server IBM JVM (IBM Cognos BI v10 only)
  - <cog\_install>/configuration/xqe.config.xml
  - set -Xmx and -Xms to the same value, set -Xmn to 1/2 -Xmx
  - 64 bit: -Xmx4096m -Xms4096m -Xmn512m (default settings)
  - 32 bit: -Xmx2048m -Xms2048m -Xmn512m (default settings)

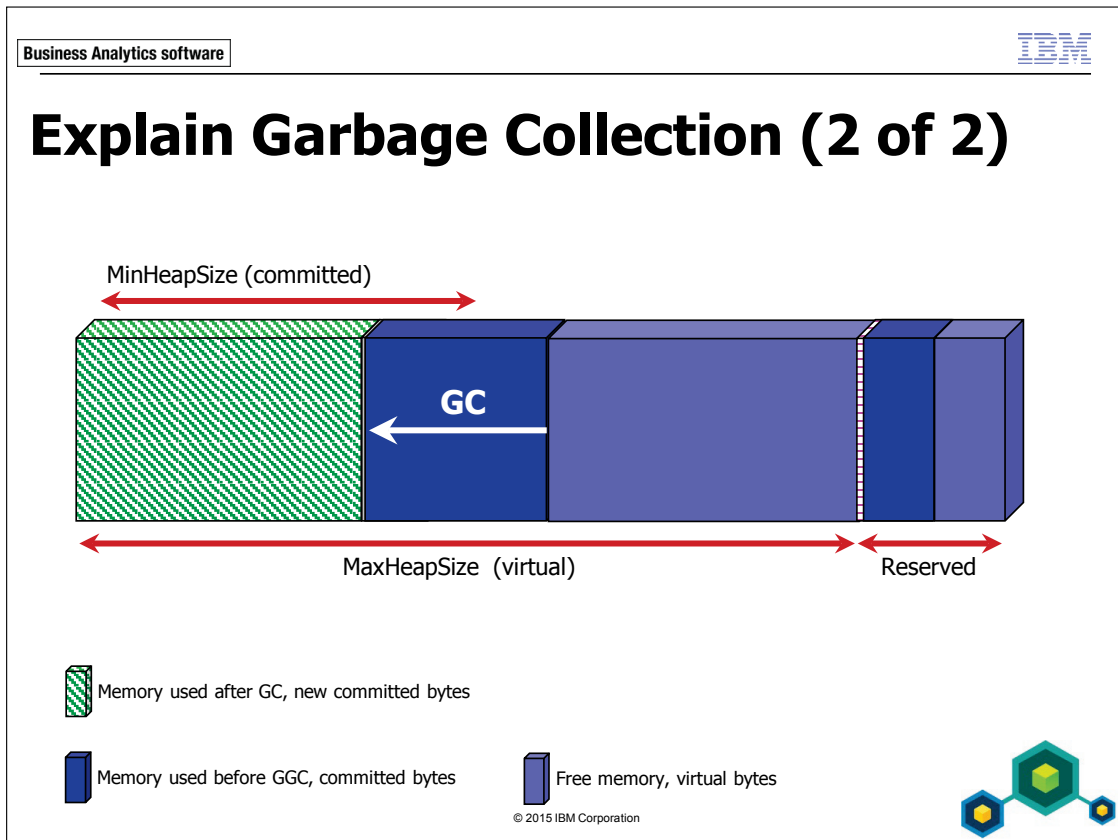
For DQM, this can be changed in IBM Cognos Administration, in the properties of the Query Service.



This example is to introduce the general concept of GC only, and does not provide implementation details.

During runtime, Java objects get created in the Java heap, thereby filling it up. If the Java heap is running low, more committed bytes are requested from the OS, up to the MaxHeapSize if virtual memory is enabled.

GC is triggered by memory allocation and/or after a period of time. GC leverages empirical observations about Java memory usage (infant mortality) and is cost intensive (lowers performance).



To avoid running out of memory, GC must occur in cycles triggered by criteria based on memory occupation (depending on the vendor and the GC implementation). GC identifies the objects not referenced by any other objects as garbage. GC will collect and dump the garbage, freeing up the memory again. The amount of committed bytes can decrease down to MinHeapSize.

## Considerations of Garbage Collection

- can cause problems when the JRE does not process code
- is cost intensive
- having an optimal GC is key to good performance
- different GC implementation strategies exist
- finding the right strategy can be challenging

© 2015 IBM Corporation



Identifying dead objects for GC is expensive and depends on the size of Java heap and the number of objects.

Different GC implementation strategies exist; depending on the JRE, several strategies are supported, and can be configured by passing parameters to the JRE upon initialization.



## What is IBM JRE Garbage Collection?

- 3 step approach to identify dead objects
  - Mark
  - Sweep
  - Compact

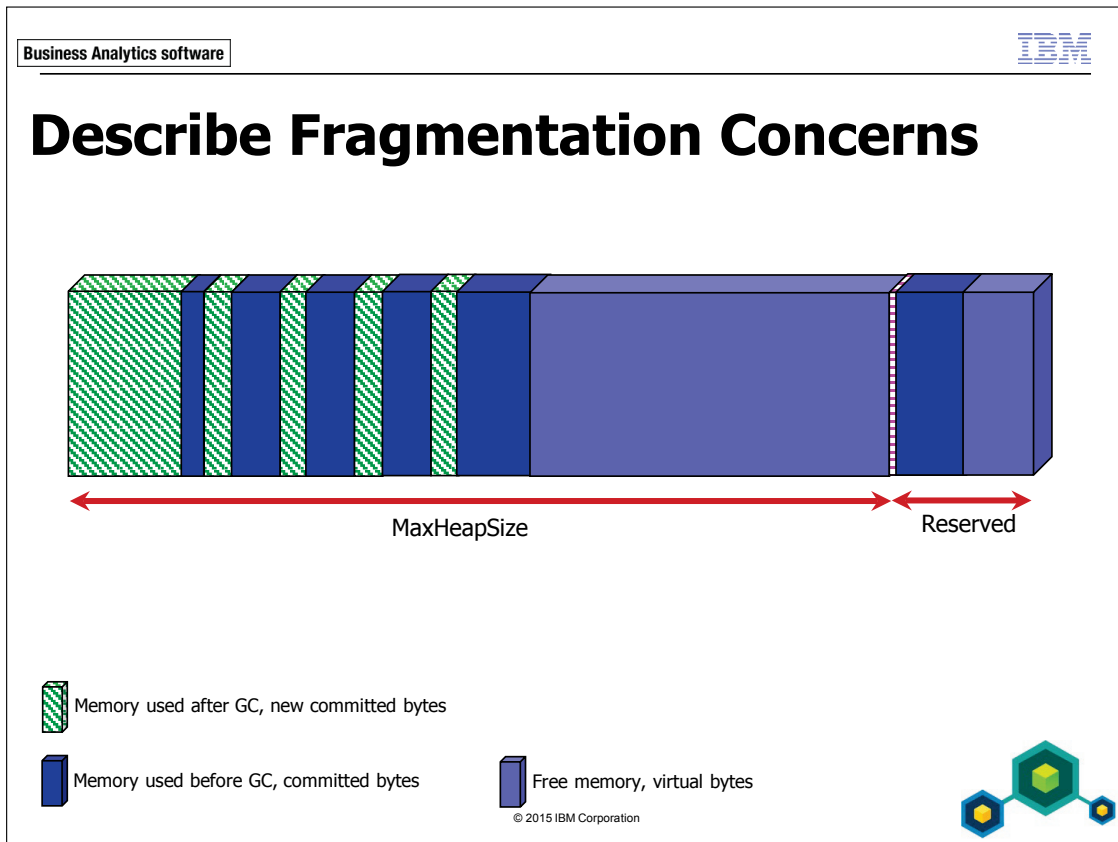
© 2015 IBM Corporation



Mark occurs most frequently, and will mark only the dead objects. This is a quick process and is not very expensive in terms of processing time.

Sweep happens less frequently, and marked objects will get freed. This is more expensive than a Mark.

Compact will defragment the memory by collecting all live objects and relocate them to a contiguous chunk. Compact is the least frequent, as it can be very expensive.



If GC is configured to run too few compacts, fragmentation can occur. It is impossible to allocate large objects and OutOfMemory errors can occur even though the sum of free bytes is sufficient.

## Log Garbage Collection

- log will at least show most crucial information about a GC run
- from a verbose GC log you can deduct if the configured MinHeapSize of 768M is a good fit

```
[GC 203376K->45888K(227220K) , 0.0279463 secs]
[GC 203200K->44599K(227220K) , 0.0173486 secs]
[GC 201910K->50273K(227220K) , 0.0462307 secs]
[GC 207585K->50437K(227220K) , 0.0490736 secs]
...
[GC 203376K->45888K(227220K) , 0.0279463 secs]

GC <oldHeapSize> -> <NewHeapSize> (<totalHeapSize>) , <timeTakenforGC>
```

© 2015 IBM Corporation



GC can be logged for basically any JRE by specifying the Verbose GC parameter. The resulting log will at least show most crucial information about a GC run, including duration, result (how many bytes have been freed), and some GC implementation specific detail (this varies by vendor).

## Verbose GC Output of IBM JRE: Example 1

### System.gc call

```
<GC(3): GC cycle started Tue Mar 10 08:24:34 2015  
<GC(3): freed 58808 bytes, 27% free (1163016/4192768), in 14 ms>  
<GC(3): mark: 13 ms, sweep: 1 ms, compact: 0 ms>  
<GC(3): refs: soft 0 (age >= 32), weak 0, final 0, phantom 0>
```

© 2015 IBM Corporation



All the lines start with GC(3), which indicates that this was the 3rd garbage collection in this JVM. The first line shows the date and time of the start of the collection. The second line shows that 58808 bytes were freed in 14 ms, resulting in 27% free space in the heap. The figures in parentheses show the actual number of bytes that are free and the total bytes that are available in the heap. The third line shows the times for the mark, sweep, and compaction phases. In this case, no compaction occurred, so the time is zero. The last line shows the reference objects that were found during this garbage collection, and the threshold for removing soft references. In this case, no reference objects were found.

Business Analytics software



## Verbose GC Output of IBM JRE: Example 2

### Allocation failure

```
<AF[5]: Allocation Failure. need 32 bytes, 286 ms since last AF>
<AF[5]: managing allocation failure, action=1 (0/6172496) (247968/248496)>
<GC(6): GC cycle started Tue Mar 10 08:24:46 2015
<GC(6): freed 1770544 bytes, 31% free (2018512/6420992), in 25 ms>
<GC(6): mark: 23 ms, sweep: 2 ms, compact: 0 ms>
<GC(6): refs: soft 1 (age >= 4), weak 0, final 0, phantom 0>
<AF[5]: completed in 26 ms>
```

© 2015 IBM Corporation



An allocation failure does not mean that an error has occurred in the code; it is the name that is given to the event that triggers when it is not possible to allocate a large enough chunk from the heap.

The output contains the same four lines that are in the `System.gc()` verbose output, and some additional lines.

The lines that start with `AF[5]` are the allocation failure lines and indicate that this was the fifth AF collection in this JVM. The first line shows how many bytes were required by the allocation that had a failure and how long it has been since the last AF. The second line shows what action the Garbage Collector is taking to solve the AF, and how much free space is available in the main part of the heap, and how much is available in the wilderness. The last line shows how long the AF took, including the time taken to stop and start all the application threads.

## Enable Verbose GC in IBM Cognos 10: Example using Tomcat with Command Line

- any JRE, log to console only

```
set CATALINA_OPTS=-Xmx768m -XX:MaxNewSize=384m -XX:NewSize=192m -
XX:MaxPermSize=128m -verbose:gc %DEBUG_OPTS%
```

- IBM JRE, log to file

```
set CATALINA_OPTS=-Xmx768m -XX:MaxNewSize=384m -XX:NewSize=192m -
XX:MaxPermSize=128m -Xverbosegclog:<path+filename> %DEBUG_OPTS%
```

- Sun JRE (as of 1.4), log to console

```
set CATALINA_OPTS=-Xmx768m -XX:MaxNewSize=384m -XX:NewSize=192m -
XX:MaxPermSize=128m -verbose:gc -XX:+PrintGCTimeStamps -
XX:+PrintGCDetails %DEBUG_OPTS%
```

- Sun JRE (as of 1.4), log to file

```
set CATALINA_OPTS=-Xmx768m -XX:MaxNewSize=384m -XX:NewSize=192m -
XX:MaxPermSize=128m -Xloggc:<path+filename> %DEBUG_OPTS%
```

© 2015 IBM Corporation



To enable verbose garbage collection, edit the `startup_tomcat.bat` located in the `..\bin64` directory and add a command line argument to the call. The examples provided are specific to using Tomcat. If you are not using Tomcat, then other steps apply.

## Enable Verbose GC in IBM Cognos 10: Example using Tomcat as a Service

- edit bootstrap\_xxxx.xml in the ..\bin64 folder and add a new `<param>` element directly after

```
<param>-Xmx${dispatcherMaxMemory}m</param>
```

- IBM JRE

```
<param>-Xmx${dispatcherMaxMemory}m</param>
```

```
<param> -Xverbosegclog:../logs/gc.log</param>
```

```
...
```

- Sun JRE

```
<param>-Xmx${dispatcherMaxMemory}m</param>
```

```
<param> -Xloggc:../logs/gc.log</param>
```



This example requires a restart of the IBM Cognos 10 service.

## Enable Verbose GC: CGS and DQM

- Cognos Graphic Service
  - additional JRE parameters can be put into
    - cgsserver.sh (UNIX)
    - ..\webapps\p2pd\WEB\_INF\services\cgsService.xml (Windows)
- Dynamic Query Mode
  - additional JRE parameters can be put into
    - ..\webapps\p2pd\WEB\_INF\services\queryService.xml (Windows and UNIX)





## Enable GC Logging in WAS

1. In **Administrative Console**, expand **Servers**, click **Application Servers**, and then click the desired server.
2. On **Configuration** tab, under **Server Infrastructure**, expand **Java and Process Management**, and click **Process Definition**.
3. Under **Additional Properties** section, click **Java Virtual Machine**.
4. Select the **Verbose garbage collection** check box, and then click **Apply**.
5. At the top of the **Administrative Client**, click **Save** to apply changes to the master configuration.
6. Stop and then restart the **Application Server**.
7. The verbose garbage collection output is written to either `native_stderr.log` or `native_stdout.log` for the Application Server, depending on the SDK operating system as follows:

**For AIX®, Microsoft® Windows®, or Linux®:**

`native_stderr.log`

**For Solaris™ or HP-UX:**

`native_stdout.log`

© 2015 IBM Corporation



WAS is Websphere Application Server.

## Tuning GC: IBM JRE

- **garbage collection frequency is too high until the heap reaches a steady state**
  - use `verbosegc` to determine the size of the heap at a steady state and set `-Xms` to this value
- **heap is fully expanded and the occupancy level is greater than 70%**
  - increase `-Xmx` value so that the heap is not more than 70% occupied
  - for best performance try to ensure that the heap never pages
  - maximum heap size should, if possible, be able to be contained in physical memory
- **at 70% occupancy the frequency of garbage collections is too great**
  - change the setting of `-Xminf`
  - default is 0.3, which tries to maintain 30% free space by expanding the heap
  - a setting of 0.4, for example, increases this free space target to 40%, and reduces the frequency of garbage collections
- **pause times are too long**
  - try using `-Xgcpolicy:optavgpause`
  - this reduces the pause times and makes them more consistent when the heap occupancy rises
  - it does, however, reduce throughput by approximately 5%, although this value varies with different applications

© 2015 IBM Corporation



These are some examples of tuning GC for IBM JRE.

## JVM Tuning and Dynamic Cubes


- dynamic cubes have large, long-lived caches, partially pre-loaded
- require much larger heap sizes (may be >120 GB)
- occupy more of the tenured space in a generational garbage collector
- may require reduced nursery size

© 2015 IBM Corporation












Dynamic Query is a Java-based engine and can be tuned to optimize JVM performance. The most common setting that users need to adjust is the maximum java heap size, but there are many configurables that can be used to tune the JVM for the specific usage patterns experienced in an enterprise environment. The default settings generally work well for traditional DQM implementations which tend to use smaller heap sizes (< 10GB) and have usage consisting of many short-lived transactions with modest caching.

The memory allocation pattern of dynamic cubes changes over time. Because dynamic cubes preload the member and aggregate caches when the cube is started, they exhibit different patterns while starting/refreshing than while queries are running against them. During a start/refresh, the cube loads a large number of objects into the JVM which will stay alive for quite a long time. In a generational garbage collector, these objects must first survive a number of nursery collections before they are promoted to tenured space. Since the nursery is optimized to copy small numbers of live objects to survivor spaces during collections, this behavior causes long pauses for large nurseries. So, during the load/refresh phase, it is ideal to have a smaller nursery, to reduce such pauses. However when queries are executing (especially after the caches are sufficiently warm), it is good to have a larger proportion of nursery space because the objects created during a query are short lived, so they play well into the nursery collection policy. Further, a larger nursery is even more useful the higher the user concurrency because higher concurrency usually causes a higher proportion of the JVM heap is being used for short-lived objects.

Business Analytics software


## Configure the JVM for Query Service

<input type="checkbox"/>		Initial JVM heap size for the query service (MB) (Requires QueryService restart)	<input type="text" value="1024"/>	Yes
<input type="checkbox"/>		JVM heap size limit for the query service (MB) (Requires QueryService restart)	<input type="text" value="1024"/>	Yes
<input type="checkbox"/>		Initial JVM nursery size (MB) (Requires QueryService restart)	<input type="text" value="0"/>	Yes
<input type="checkbox"/>		JVM nursery size limit (MB) (Requires QueryService restart)	<input type="text" value="0"/>	Yes
<input type="checkbox"/>		JVM garbage collection policy (Requires QueryService restart)	Generational ▾	Yes
<input type="checkbox"/>		Additional JVM arguments for the query service (Requires QueryService restart)	<input type="text"/>	Yes
<input type="checkbox"/>		Number of garbage collection cycles output to the verbose log (Requires QueryService restart)	<input type="text" value="1000"/>	Yes
<input type="checkbox"/>		Disable JVM verbose garbage collection logging (Requires QueryService restart)	<input type="checkbox"/>	Yes



© 2015 IBM Corporation

In IBM Cognos Administration, there are eight JVM configurable values available for tuning the Query Service.

The first four properties allow you to adjust the minimum and maximum sizes of the overall JVM respective nursery sizes within the overall heap.

For more information on the individual properties, refer to the product documentation. For more details on heap sizing, refer to

[http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/index.jsp?topic=%2Fcom.ibm.java.doc.diagnostics.60%2Fdiag%2Funderstanding%2Fmm\\_heapsizing\\_initial.html](http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/index.jsp?topic=%2Fcom.ibm.java.doc.diagnostics.60%2Fdiag%2Funderstanding%2Fmm_heapsizing_initial.html).

## Tips on Tuning Settings

- set initial heap size large enough for normal operation
- set the maximum heap size large enough for peak operation plus 15% for overhead to prevent unnecessary GCs
- set initial nursery size to 0

© 2015 IBM Corporation



Set the initial heap size large enough for normal operation. It should be large enough to fit all the caches and run most reports at the required level of concurrency. For more information on hardware requirements, see the document entitled *Understanding Hardware Requirements for Dynamic Cubes* in the Business Analytics Proven Practices Web location at <http://www.ibm.com/developerworks/analytics/practices.html>.

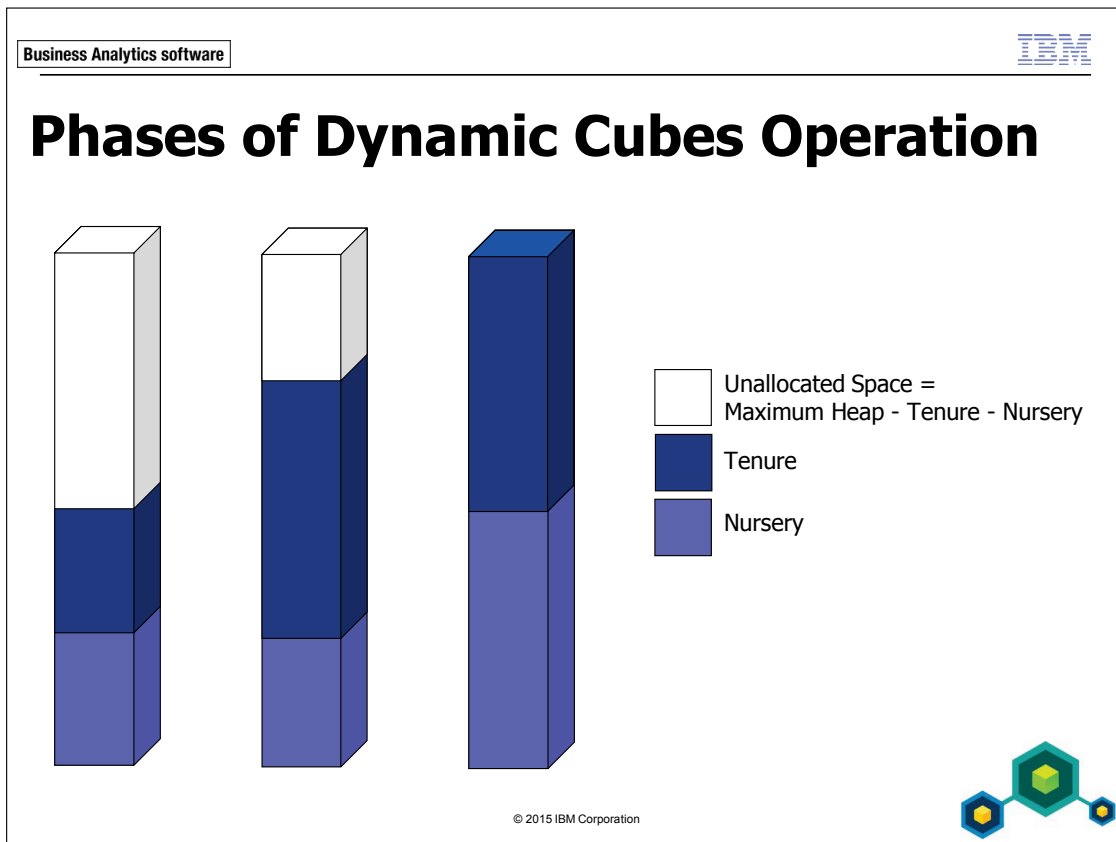
Set the maximum heap size large enough for peak operation plus a small (15%) overhead to prevent unnecessary GCs. It should be large enough to absorb the memory spikes that can occur when more data volume-intensive queries are executed or a higher concurrency is experienced.

Set the initial nursery size to 0 unless there is a problem. This will allow the system to calculate the initial nursery size as a proportion of the initial heap size. The calculated proportion has been shown to achieve better performance, especially during and immediately after the priming phase.

Set the maximum nursery size to an appropriate level, given your expected amount of caching, reporting style, and concurrency requirements. The idea is for all the temporary objects created during report execution to fit in the nursery so they aren't promoted to tenured space. That way they can take advantage of the nursery's collection policy. A larger proportion of nursery space to tenured space is required for higher concurrency and/or data volume-intensive reports with modest caching, than for lower concurrency and/or simple reports with aggressive caching. For the former, you may choose to set the maximum nursery size to 50% of the maximum heap. For the latter, you could leave it set to 0 and the system will calculate it as a smaller proportion of the maximum heap.

Set the maximum tenured size using the `-Xmox` setting in the additional arguments to the maximum heap - maximum nursery. This should generally be large enough to fit all the caches, plus a 10-20% buffer. Since tenured space is generally increased before nursery space, this maximum allow the tenured space to expand large enough to fit all the caches and then allow the nursery space to increase to accommodate queries, which generally only need short term space.

Set minimum tenured size using the `-Xmos` setting in the additional arguments to the same value as maximum tenured space? This will prevent unnecessary GCs for increasing the size.



The tuning setting tips should allow the heap proportions to be adjusted automatically by the JVM so that they are optimal for the distinct phases of dynamic cubes operation.

While there is no theoretical limit on the maximum heap size, performance starts to degrade as more long-lived objects are added to memory. The theory is that tenured GC is scanning all the objects to find dead ones and that gets more expensive. Also, it maybe more expensive to allocate objects when there are many objects alive, due to overhead. GC pauses may become prohibitively long when the heap usage grows beyond ~120GB, so the initial heap size generally should not be set larger than that value, and the maximum heap size should not be much larger. Of course, pause time can also vary due to hardware variances and enterprise environment patterns, and so this is not a hard limit.



## Analysis of JVM Garbage Collection

- Out of Memory (OOM) messages
- Heapdumps
- Verbose GC logs

© 2015 IBM Corporation



Out of memory (OOM) messages: Severe out of memory conditions can affect the stability of the Query Service. In such cases, the Query Service will become unavailable in the portal. In this case, the maximum JVM heap size (and probably the three others) should be increased, and the Query Service should be restarted.

In the case where the Query Service is able to remain stable, the Query Service will report the `OutOfMemoryError` to every user whose action was affected by the error. This includes report execution as well as admin operations (such as cube cache refreshes). The error message also includes the current setting for the maximum JVM heap size, so you can cross reference it to the setting in the portal. For example: "Insufficient memory. The current JVM maximum heap size of the QueryService is 512MB and may need to be increased".

Heapdumps and MAT tool: When an `OutOfMemoryError` occurs, one or more heap dump files are generally created in the `bin` or `bin64` directory. These are useful for looking at the contents of the heap at the time of the error to find the root cause of issues and can be analyzed with the Memory Analyzer Tool. Information on the Memory Analyzer Tool is available at:

<http://www.ibm.com/developerworks/java/jdk/tools/mat.html>.

Verbose GC logs and ISA tool: Verbose GC logging is particularly useful for analyzing and tuning JVM garbage collection. A detailed set of instructions for how to interpret the log and use it to tune the JVM is beyond the scope of this course, but it should be noted that:

- Verbose GC logging is enabled by default
- You can disable it or adjust the size of the log with the dedicated parameters in the portal
- The IBM Support Assistant tool is useful for analyzing the log (Information on the ISA tool is available at: <http://www-01.ibm.com/software/support/isa/>.)
- Large collection times generally indicate insufficient heap/nursery sizes, suboptimal proportions of nursery and tenured space

## What Happens in a JRE Crash?

- dump file called a Java core created
- depending on the JRE version one of these files is in the ..\bin directory of the IBM Cognos 10 install
  - HS\_err\_(timestamp).txt
  - Javacore\_(timestamp).txt

© 2015 IBM Corporation



A JRE can crash like any other process, and when that happens, a dump file is created. Depending on the version of your JRE you will get one of two files in the ..\bin directory of the IBM Cognos 10 install.

Depending on the JRE, the file will contain local environment info, JRE information, thread dumps (per thread) in a Stack trace, GC history, memory information, and ClassLoader information. This file may be required by IBM Cognos Support to help troubleshoot issues.

## What Should You Do if Java Hangs?

- force a Java dump or Threaddump
  - Windows: Ctrl+Break (only works in CMD window)
  - UNIX: Kill -3 <Java\_PID>

© 2015 IBM Corporation



Sometimes a Java process churns or hangs, and is unresponsive. To troubleshoot, force a Java dump or Threaddump to obtain similar information as a Java core file.

## Use Tools to Monitor Java

- Process Explorer:
  - <http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx>
- IBM Support Assistant:
  - <http://www-01.ibm.com/software/support/isa/>



## Summary

- At the end of this module, you should be able to:
  - describe Java memory layout
  - manage Java memory
  - use tools to monitor Java memory

