JVM Performance and Heap Tuning

- Heap Fundamentals
- Garbage Collection Algorithms
- Tuning the Heap

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

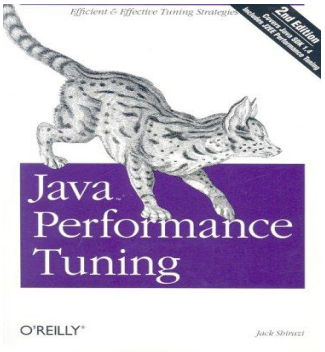
JVM Performance Factors

- There are many factors that influence Java application performance, including (in no particular order):
  - Choice of algorithms
  - I/O, database and network speed
  - CPU speed
  - Amount of physical memory
  - Garbage collection
  - JVM version
  - JVM options, especially heap size

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

A Strategy for Tuning

- From the book **Java Performance Tuning**:
  - Identify the top few bottlenecks
  - Choose the easiest one to fix
  - Repeat
- The reason it's best to fix the easiest is that once you fix a bottleneck, it may have side effects on others



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

How To Find Bottlenecks

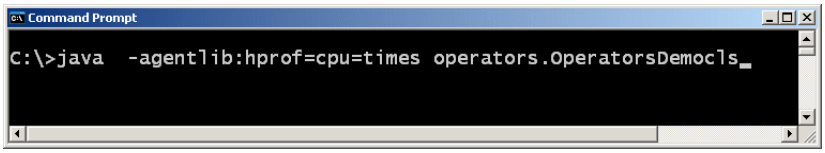
- To discover where the program is using time, you MUST use a profiler of some sort
- Java profilers include:
  - JProfiler
  - JProbe
  - HProf (built into JVM 5 and later)
  - jvisualvm (built into JVM 6 and later)

JProfiler and JProbe are commercial profilers.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using HProf

- **HProf** is a command-line option on the JVM that collects statistics on a program's heap and CPU




```
java.hprof.txt
--------------

. . .

rank   self  accum   count trace method
   1 34.04% 34.04%     298 300317 java.lang.String.charAt
   2 34.04% 68.09%     473 300621 sun.nio.cs.Surrogate.is
   3 31.91% 100.00%      16 300644 java.io.FileOutputStream.write
```

When you use HProf with the options shown in the screen shot, it stores the statistics in a file named "java.hprof.txt".

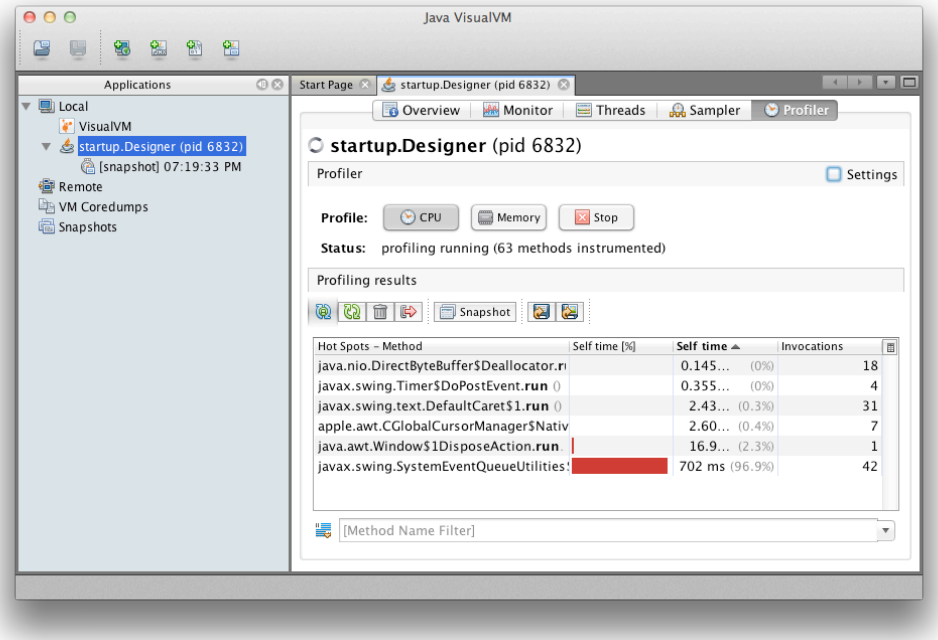
At the bottom of the file, it shows methods and how much time was spent in each, which is invaluable for finding bottlenecks.

In the simple program shown here, all of the execution time was spent in libary methods, not in application methods.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using jvisualvm

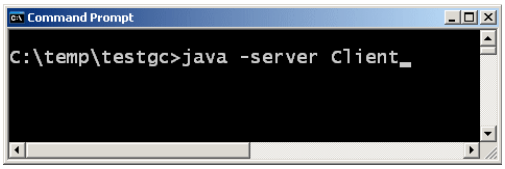
- **jvisualvm** is a command-line tool with a GUI for examining a Java program's memory and CPU statistics



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The -server and -client Options

- The Oracle JVM supports two base modes: **client** and **server**
- The names are misleading - you might choose **-server** for a client-side GUI, for example
- You should profile with both to determine which is best for your application



The main difference between the two options is how they convert Java byte codes into native machine code. The "client" option is designed to speed the initial execution so that GUI applications start faster to give better perceived performance, while the "server" option is designed to better optimize the generated code, leading to better performance over a longer period of time.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is Garbage Collection?

- **Garbage collection** is a service provided by the JVM so that applications do not need to explicitly free storage
- Modern JVMs provide several choices on how collection works and configuration
- All garbage collectors use one or more threads to perform collection - the GC threads run in parallel with application threads

```
1    public void myMethod()
2    {
3       Date d = new Date();
4       int[] array = new int[700];
5       . . .
6
7    }  // "d" and "array" available for GC
```



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

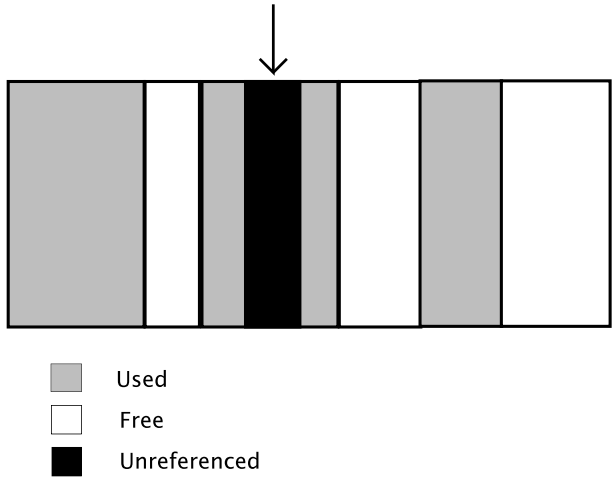
Garbage Collection Fundamentals

- Garbage collection involves three basic operations:
  - **Finding** unreferenced storage
  - **Deleting** unreferenced storage
  - **Compaction**
- All of the JVM garbage collectors perform these three operations in different ways

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Finding Unused Storage

- To find unused storage, garbage collector thread(s) scan the heap and the stack, finding objects and other storage that cannot be reached from the application
- JVMs provide different strategies on how many scanning threads exist



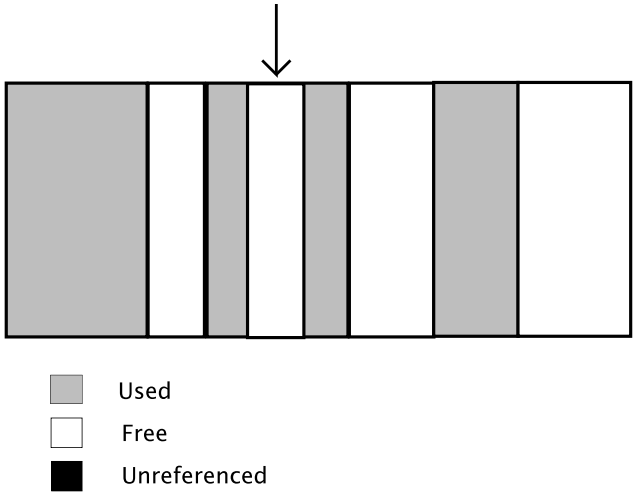
2 - 10

Collectors that use multiple threads to find unused storage run in competition with application threads. This results in higher CPU usage.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Deleting Unused Storage

- The JVM keeps a list of available areas in the heap
- Once the GC has identified unused storage, it can free the storage by adding to to the JVM's "available" (free) list

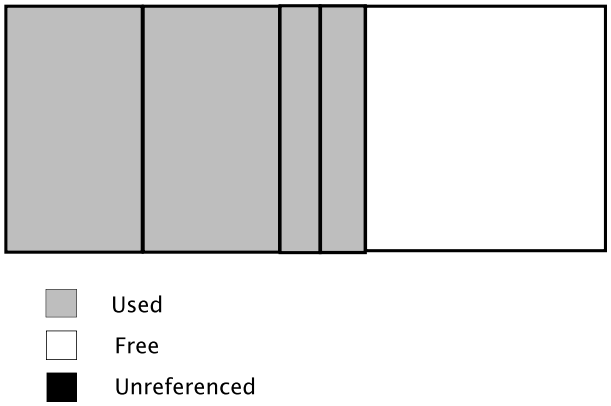


After deletion, note that there could be lots of free storage in the heap, but it's fragmented. In other words, even if there's sufficient TOTAL free space to satisfy a requrest, there might not be enough CONTIGUOUS free space.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Compacting the Heap

- Simply freeing storage is insufficient, since it could result in a **fragmented** heap that has available storage, but none large enough to satisfy an application's request
- During compaction, the garbage collector moves storage to create larger available chunks



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Stop the World (I Want to Get Off)

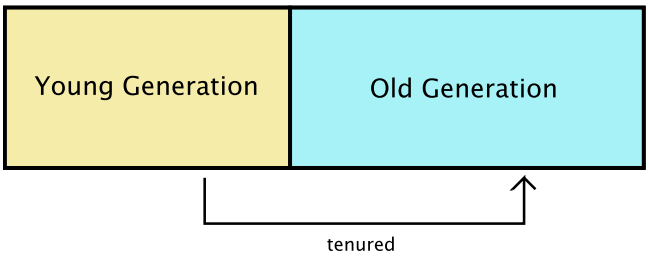
- During compaction, the garbage collection must pause application threads since the memory addresses of objects and other storage may change
- The act of pausing the application during garbage collection is referred to as **stop the world**
- The main goal of tuning is to minimize STW pauses according to the application's needs

Java programs don't use actual addresses of storage - instead, they use "references". During compaction, the collector moves storage and then updates the reference to point to the new address. The collector must stop application threads to guarantee that the application threads cannot use references during the update.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Generations

- To minimize the number and duration of STW pauses, modern JVMs divide the heap into sections referred to as **generations** - the basic categories are called the **young** and **old** generations
- This strategy works well because most programs create many short-lived objects - these reside in the young generation, which is often collected
- The garbage collector moves long-lived objects to the old generation, which it processes less frequently

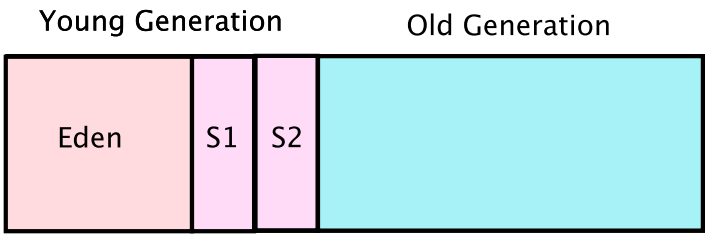


You can configure the sizes of the generations using flags when you start the JVM.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Collecting the Young Generation

- Most JVMs further subdivide the young generation into **eden** and one more **survivor** spaces - new objects and other allocations go into eden
- When eden fills, the GC processes **all** of the allocations there, either freeing them or moving them to a survivor space
- Since the GC completely clears eden every collection, it doesn't need to compact that space
- The act of collecting eden is referred to as a **minor GC** and is typically a fairly quick operation



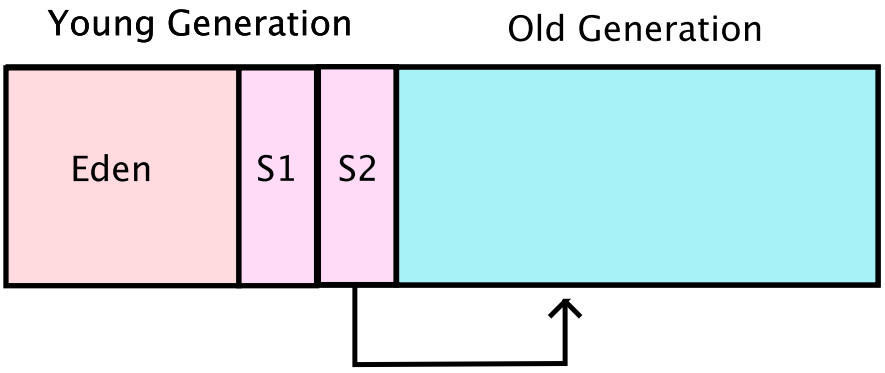
Minor GC is relatively quick, since the collector doesn't need to perform compaction.

Typically, minor GC does require STW since the collector needs to move storage.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Tenured Storage

- During each minor GC, the JVM **ages** storage in the survivor space(s)
- Once a given storage reaches a threshold age, the GC **tenures** it by moving it to the old generation

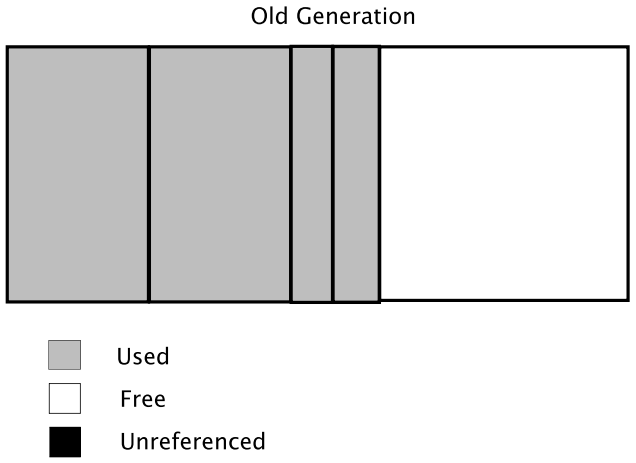


As always, "storage" consists of objects, arrays and so forth.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Collecting the Old Generation

- When the old generation fills, the JVM performs a **full GC**, collecting and compacting it
- Full collections may take a relatively long time and stop-the-world resulting in poor application response
- JVMs provide multiple **collection algorithms** so you can tune GC to your application's needs



Note that a full GC might occur even if there's plenty of available space in the OG – if it's fragmented, there might not be a large enough CONTIGUOUS space for the requested allocation.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Garbage Collection Algorithms

- The Oracle JVM (Java 8) provides four different GC algorithms:
  - **Serial** - Uses a single thread; only appropriate for single-core CPUs
  - **Throughput (aka Parallel)** - Uses multiple threads to GC the young and old generations; appropriate for (and default) for multi-core CPUs
  - **CMS** - Multi-thread "low-pause"; uses more CPU than **Throughput**
  - **G1** - Multi-thread "low-pause"; more CPU; appropriate for large (>4GB) heaps

CMS is an acronym for Concurrent Mark and Sweep.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Choosing a GC Algorithm

- JVMs running on single-core CPUs use **Serial** by default, while on multi-core, **Throughput** is default
- You can override the default using JVM options:

Algorithm	JVM Option(s)
Serial	-XX:+UseSerialGC
Throughput	-XX:+UseParallelOldGC
CMS	-XX:+UseConcMarkSweepGC -XX:+UseParNewGC
G1	-XX:+UseG1GC



The defaults may well be different on newer JVMs.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Serial Algorithm

- **Serial** is the default on 32-bit JVMs and single-core machines
- It uses a single thread to find, delete and compact, fully compacting the old generation during a full GC (stop-the-world)

```
-XX:+UseSerialGC
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Throughput Algorithm

- **Throughput** aka **parallel** is the default on 64-bit and multi-core machines
- It uses multiple threads to collect both the old and new generations, fully compacting the old generation during a full GC
- Both minor and full GC are stop-the-world events
- This collector uses more CPU than the serial collector, but makes minor GC much faster

```
-XX:+UseParallelOldGC
```


In older (pre JDK7u4) JVMs, this collector only used multiple threads for the young generation

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The CMS Algorithm

- The **Concurrent Mark and Sweep** (CMS) collector is designed to minimize long stop-the-world pauses
- CMS does not pause application threads during a full GC - tradeoff is that the old generation can become fragmented
- Once fragmentation occurs, CMS may require a lengthy STW pause to compact the old generation
- CMS tends to use significantly more CPU than the throughput and serial collectors

```
-XX:+UseConcMarkSweepGC -XX:+UseParNewGC
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The G1 Algorithm

- The **Garbage First** (G1) collector is designed to efficiently process large heaps
- It subdivides the old generation into regions and collects them in a manner similar to how all collectors process the young generation (i.e. survival spaces)
- Since it completely clears at least a portion of the old generation during a full GC, it is less subject to fragmentation than CMS
- Like CMS, it can use a significant amount of CPU

```
-XX:+UseG1GC
```


G1 can be faster than CMS for large heaps because during a full GC, it doesn't need to scan the entire old generation - it can process the subregions. On the other hand, CMS must process the entire old generation during a full GC - if the heap is large, that can cause significant slowdowns.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Choosing a GC Algorithm

- There is no easy metric for deciding which algorithm is best for your application - you need to profile and stress test to see which one works best
- However, here are a few general guidelines:
  - For large heaps (> 4GB), G1 will probably work best
  - For interactive applications where response time is important, the throughput collector will often exhibit shorter **average** response time, but may have significantly longer **99th percentile** pauses than the CMS or G1 collectors
  - For batch applications where throughput (shortest execution time) is important, if the app itself is CPU bound, the throughput collector is best, but if there's available CPU, CMS or G1 is likely the best choice

These guidelines are from the excellent Java Performance: Definitive Guide book.

For interactive apps, the 99th percentile difference is likely due to the fact that CMS needs to do less work during a full GC. However, if the old generation becomes fragmented, CMS may incur a very large pause - hopefully this occurs far less often than 1 % of the requests. And as always, CMS/G1 tend to use more CPU than throughput, so if the server is CPU limited, throughput might still be the better choice, even if you are attempting to minimize the maximum pause (slowest response).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

PermGen: Java 7 and Earlier

- Older JVMs use **Permanent Generation (PermGen)** space to hold classes and other metadata items
- The PermGen is a separate space from the "normal" heap
- If you see "OutOfMemoryError: PermGen space" in logs, you can increase the maximum PermGen space

```
-XX:PermSize

-XX:MaxPermSize
```


PermGen only includes metadata, not any actual classes themselves - the Class objects reside in the normal heap.

Java 7 started refactoring PermGen as a step to Java 8, where it is replaced by "metaspace". Some of the changes in Java 7 include moving certain parts of the metadata to the normal heap, thus reducing the amount needed by PermGen (but increasing use of the normal heap).

The name "permanent generation" is a bit of a misnomer, since the data stored there isn't actually permanent - the JVM may need to garbage collect it if it fills. In fact, if you notice an app incurring a lot of GC as it loads, it might be that the initial PermGen size is too small and you can decrease load time by increasing it.

Note that you can also set the initial size of PermGen with the XX:PermSize flag.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Metaspace: Java 8

- In Java 8 and later, classes and other metadata is stored in **metaspace**
- Unlike the PermGen space of earlier Java versions, the JVM allocates metaspace from the native heap
- You can configure the metaspace using JVM flags

```
-XX:MetaspaceSize

-XXMaxMetaspaceSize
```


Metaspace replaces the Permanent Generation (PermGen) of earlier versions of Java.

By default, there is no upper bound on metaspace size, so a program that "leaks" classloaders (for example) could exhaust native memory. For an example that shows how that might happen, see this article: https://dzone.com/articles/java-8-permgen-metaspace

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Tuning the Heap

- JVMs provide several options to let you specify heap settings, including:
    - Min and max size of the whole heap
    - The min and max size of the young generation
    - The ratio of the young generation to the old generation
- Before and during tuning, you can use **jvsiualvm** or JVM **garbage-collection logs** to analyze the heap

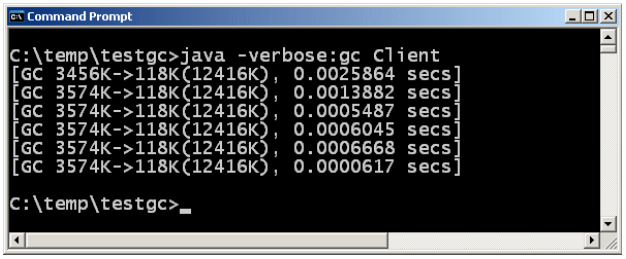
This and the following pages are about tuning the "normal" heap, not PermGen or metaspace, which have their own JVM flags to configure.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Verbose GC Logging

- During tuning, you can enable heap statistic logging so you can analyze the heap and how garbage collection is performing

```
-verbosegc
```



By default, this swtich sends GC output to "stderr", but you can redirect it using a different JVM option:

```
java -Xloggc:test.txt Client
```

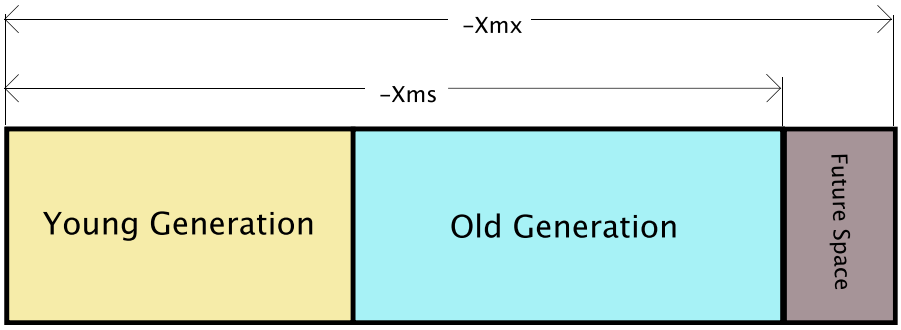
The -Xloggc option replaces the -verbose:gc option.

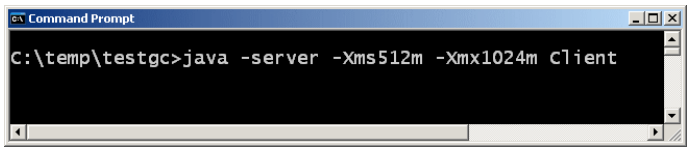
Note that jvisualjvm also provides some of the same statistics (as well as a profiler).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sizing Total Heap Size

- You can use the **-Xms** and **-Xmx** flags to configure the initial and maximum size total heap size





You can create a fixed-size heap by specifying the same values for initial and max.

Note that you should NEVER set the heap size to a value greater than the total amount of physical memory - most operating systems will swap part of the heap to disk in that case, which has a large impact on performance. The performance hit would be especially bad during a full GC where the JVM needs to access the entire heap. The Java Performance book suggests: "A good rule of thumb is to size the heap so that it is 30% occupied after a full GC."

Note also that in the absence of these flags, JVMs use (hopefully) sensible defaults. For example, most JVMs use some fraction (e.g. one quarter) of the total available machine memory as the default for -Xmx.

In addition to, or instead of configuring the total heap size, you can configure the ratio of the new generation to the total heap using the -XX:NewRatio flag. The default value is 2, which yields a new generation size of 33% - the formula used is:

```
YG size = total size / (1 + NewRatio)
```

The remaining storage goes to the old generation.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- How to profile Java code to find performance bottlenecks
- How garbage collection works and the various GC algorithms
- How JVS partition the heap into generations
- About tuning heap sizes

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum