GALWAY-MAYO INSTITUTE OF TECHNOLOGY

Department of Computing & Mathematics

Garbage Collection and JVM Heap Analysis

The Java Virtual Machine abstracts the underlying operating system and hardware components of a computer and provides a universal runtime environment for bytecode instructions. Conceptually, it is convenient to view the JVM as a runtime environment composed of two basic data structures, an object heap and a method stack. When an object is created, through the invocation of a constructor, the instance is given an Object ID by the JVM and placed on the heap. A heap is actually a binary tree and can be implemented as an object array. Methods are executed on one of the JVM stacks with each thread given its own stack. The main thread of execution begins on the main stack through the invocation of the static *main()* method of an application. When the *main()* method returns, the main thread of execution exits and the JVM shuts down.

During the lifetime of an application, many different objects may be instantiated and placed on the JVM object heap. As long as an object remains in scope (is reachable), it will remain on the heap until the *main()* method returns. In contrast with languages like C and C++, memory (heap space) is managed automatically by the Java runtime environment (JVM). An area of heap space can be reclaimed by the garbage collector when an object can no longer be reached by any part of a running programme.

A number of different garbage collection algorithms are available from the JVM based on the idea of generational collection. While a jejune garbage collection algorithm will examine every object on the heap, the algorithms used by the JVM are based on the ***weak generational hypothesis:*** *most objects survive for only a short period of time*. Understanding how the heap space is utilised and how the different garbage collection algorithms function, enable the tuning of a JVM to fit the specific needs of a software application. In addition, JVM monitoring tools such as JVisualVM facilitate the runtime analysis of both CPU and heap usage. As such, JVM analysis tools provide a powerful mechanism for maximising the space and time efficiency of Java applications. The most elementary configuration of the heap is the tuning of the initial and maximum heap size. The *Xms* and *Xmx* switches are used as JVM command line parameters to specify these settings if required.

- **Initial Heap Space (-Xms):** Larger than 1/64th of the physical memory (RAM) available.

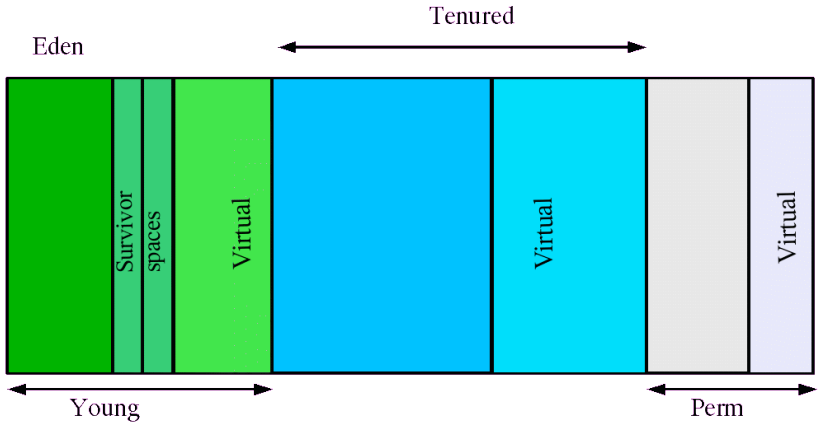
- **Maximum Heap Size (-Xmx):** Smaller than 1/4th of the physical memory (RAM) or 1GB.

JVM Object Heap – The Generation Game

The ***weak generational hypothesis*** is implemented in the object heap as a managed set of generations (memory pools) storing objects of different ages. Garbage collection occurs in each generation when the memory pool fills up. Live objects (those that survive *minor collections*) are promoted through each memory pool until they become tenured. Once an object has tenure, it will only be destroyed in a *major garbage collection*. Note that we can be

notified about when an object has gone out of scope by over-riding the ***finalize()*** method inherited from *java.lang.Object*. In addition, the deprecated *Runtime.runFinalizersOnExit* method can be used to force the JVM to execute all finalisers before it shuts down (***note:*** this method is inherently unsafe and should not be used in a real application).

The JVM heap space is divided into the following generations or memory pools:



1. **Young Generation:** a pool dedicated to young objects representing the vast majority of instances created by an application. A ***minor garbage collection*** is performed when the young generation fills up. Only garbage in the young generation is removed during a minor collection – garbage in the tenured or permanent space is not affected. If the weak generational hypothesis is true, garbage collection can be optimised, as most garbage will be short-lived and exist only in this generation.

    - **Eden Space:** Objects are initially allocated into Eden space. Objects that survive a minor garbage collection in Eden space are copied to one of the survivor spaces.

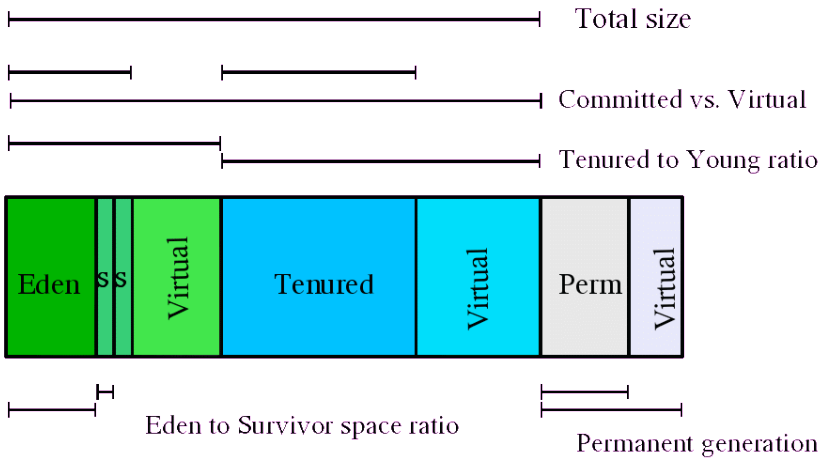
    - **Survivor Space (S0 and S1):** One survivor space is empty at any time, and serves as the destination of any live objects in Eden space and the other survivor space after a garbage collection. Objects are copied between survivor spaces in this way until they are old enough to be moved to the tenured generation.

2. **Tenured Generation (Old):** Live objects that have survived minor garbage collections in the young generation are given tenure. Note that the size of the young generation can be manipulated to tenure objects quickly if it is set low enough. Garbage in the tenured generation is only destroyed during a ***full garbage collection***.

3. **Permanent Generation (Meta):** Only objects that are fundamental to the working of the JVM are kept in the permanent generation. This includes all the reflective data of the JVM itself, such as class and method objects. This generation is divided into read-only and read-write segments.

The size of the young and tenured generations is controlled by a ratio (typically 1:8). Consequently, increasing young generation space will reduce the amount of tenured space and the number of minor collections that occur. The total size of the young generation is controlled by the parameter **NewRatio**. Setting **-XX:NewRatio=3** sets the ratio between the young and tenured generations to 1:3, i.e. the combined size of the Eden and survivor spaces

will be one fourth of the total heap size (the permanent generation pool size is not configurable). The parameter **MaxNewSize** can be used to place a ceiling on the young generation pool size.



An additional parameter, **SurvivorRatio**, can be used to tune the size of the survivor spaces. Specifying **-XX:SurvivorRatio=6** sets the ratio between Eden and a survivor space to 1:6, i.e. each survivor space will be one sixth the size of Eden, and thus one eighth the size of the young generation (not one seventh, because there are two survivor spaces). If survivor spaces are too small, copying collection overflows directly into the tenured generation. If survivor spaces are too large, they will be wastefully empty. At each garbage collection, the JVM chooses a threshold number of times an object can be copied before it is tenured. This threshold is chosen to keep the survivors half full.

Garbage Collection Algorithms

When executing an application, the JVM can be parameterised in numerous different ways, including the specification of one of the following three garbage collection algorithms. Note that different version of Java may use different collectors with different parameters.

- **Serial Collection:** Garbage collection is performed in a single thread of execution and will suspend other threads while it is freeing up memory ("stop the world garbage collection"). The serial collector cannot exploit multi-processor capabilities and is thus better suited for single processor machines with data overheads < 100MB. Specified by the **-XX:+UseSerialGC** option.

- **Parallel Collection:** Executes minor and major collections in parallel (latter since Java 1.6), significantly reducing garbage collection overhead by making minor collection pauses shorter. On a machine with *n* processors the parallel collector uses *n* garbage collector threads (controlled by the *-XX:ParallelGCThreads=<n>* parameter). Designed for medium/large sized data sets that are run on multiprocessor hardware. Specified by the **-XX:+UseParallelGC** option.

- **Concurrent Mark and Sweep Collection:** When dealing with a large amount of data and multiple CPUs, CMS will reduce collection pauses at the expense of more total CPU time. CMS pauses an application twice during a concurrent collection cycle. The first pause is to mark as live the objects directly reachable from the roots (e.g., object references from application thread stacks and registers, static objects and so on) and from elsewhere in the heap (e.g., the young generation). This first pause is referred to as the *initial mark pause*. The second pause comes at the end of the concurrent tracing phase and finds objects that were missed by the concurrent tracing

due to updates by the application threads of references in an object after the concurrent collector had finished tracing that object. This second pause is referred to as the *remark pause*. Specified by the **-XX:+UseConcMarkSweepGC** option.

Exercises

In order to gain a practical understanding of how the heap and garbage collector work, we'll create a large number of short-lived objects and a smaller number of longer living objects to occupy the young and permanent generations respectively. We'll then monitor the heap space after specifying different garbage collectors. Finally, we'll configure the memory pool sizes to exploit our understanding of generational garbage collection.

1. Using Eclipse, create a new class called ***Person***, with a single instance variable of ID (an **int**). Ensure that the constructor is parameterised with an **int**.

2. Override the ***finalize()*** method of ***Object*** in the class ***Person*** and output the value of both the Object ID and the instance variable ID.

3. Create a second class called ***Runner*** that uses a loop to create 10^9 instances of ***Person*** inside the loop. Every 100^{th} instance of the class ***Person*** should be added to an ***ArrayList*** declared at a class level.

4. Open a terminal and start the JVisualVM application with the command ***jvisualvm***. Note that this binary is stored in the *$JAVA_HOME/bin* directory along with the other basic SDK tools.

5. Execute the ***Runner*** class from Eclipse and connect to the JVM using JVisualVM. Use the -Xms and -Xmx switches to specify a maximum and minimum virtual heap allocation space. Select the ***Visual GC*** tab and change the refresh rate to 100ms. You should see the Eden space and survivor spaces filling up and being emptied quickly (minor garbage collection) and the tenured generation filling slowly.

6. Increase and decrease the size of the young generation and survivor space using the **-XX:NewRatio**, **-XX:MaxNewSize** and **-XX:SurvivorRatio** switches. Pay particular attention to how these settings impact memory consumption and garbage collection activity.