**Garbage Collection**

**Key Hotspot Components**

There are three components of the JVM that are focused on when tuning performance. The ***heap*** is where your object data is stored. This area is then managed by the **garbage collector** selected at startup. Most tuning options relate to sizing the heap and choosing the most appropriate garbage collector for your situation. The **JIT compiler** also has a big impact on performance but rarely requires tuning with the newer versions of the JVM.

**What is Automatic Garbage Collection?**

Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused object, or unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed.

The basic process of garbage collection is described below:

**Step 1: Marking**

The first step in the process is called marking. This is where the garbage collector identifies which pieces of memory are in use and which are not.

All objects are scanned in the marking phase to determine which objects are referenced and which are not referenced. This can be a very time consuming process if all objects in a system must be scanned.

**Step 2: Normal Deletion**

Normal deletion removes unreferenced objects leaving referenced objects and pointers to free space. The memory allocator holds references to blocks of free space where new object can be allocated.

**Step 2a: Deletion with Compacting**

To further improve performance, in addition to deleting unreferenced objects, you can also compact the remaining referenced objects. By moving referenced object together, this makes new memory allocation much easier and faster.

**Why Generational Garbage Collection?**

As stated earlier, having to mark and compact all the objects in a JVM is inefficient. As more and more objects are allocated, the list of objects grows and grows leading to longer and longer garbage collection time.

**JVM Generations**

Most of the objects that are used in programs have very short life span. This analysis is used to break the heap in 3 different parts or generations: Young Generation, Old or Tenured Generation, and Permanent Generation

* **Young generation:** The Young Generation is where all new objects are allocated and aged. When the young generation fills up, this causes a minor garbage collection. Minor collections can be optimized assuming a high object mortality rate. A young generation full of dead objects is collected very quickly. Some surviving objects are aged and eventually move to the old generation.

**Stop the World Event** - All minor garbage collections are "Stop the World" events. This means that all application threads are stopped until the operation completes. Minor garbage collections are always Stop the World events.

* **Old generation:** The Old Generation is used to store long surviving objects. Typically, a threshold is set for young generation object and when that age is met, the object gets moved to the old generation. Eventually the old generation needs to be collected. This event is called a major garbage collection.

Major garbage collection are also Stop the World events. Often a major collection is much slower because it involves all live objects. So for Responsive applications, major garbage collections should be minimized. Also note, that the length of the Stop the World event for a major garbage collection is affected by the kind of garbage collector that is used for the old generation space.

* **Permanent Generation**: The **Permanent generation** contains metadata required by the JVM to describe the classes and methods used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here.

Classes may get collected (unloaded) if the JVM finds they are no longer needed and space may be needed for other classes. The permanent generation is included in a full garbage collection.

**The generation garbage collection process**:

* First, any new objects are allocated to the eden space. Both survivor spaces start out empty.
* When the eden space fills up, a minor garbage collection is triggered.
* Referenced objects are moved to the first survivor space. Unreferenced objects are deleted when the eden space is cleared.
* At the next minor GC, the same thing happens for the eden space. Unreferenced objects are deleted and referenced objects are moved to a survivor space. However, in this case, they are moved to the second survivor space (S1). In addition, objects from the last minor GC on the first survivor space (S0) have their age incremented and get moved to S1. Once all surviving objects have been moved to S1, both S0 and eden are cleared. Notice we now have differently aged object in the survivor space.
* At the next minor GC, the same process repeats. However this time the survivor spaces switch. Referenced objects are moved to S0. Surviving objects are aged. Eden and S1 are cleared.
* This slide demonstrates promotion. After a minor GC, when aged objects reach a certain age threshold (8 in this example) they are promoted from young generation to old generation.
* As minor GCs continue to occure objects will continue to be promoted to the old generation space.
* So that pretty much covers the entire process with the young generation. Eventually, a major GC will be performed on the old generation which cleans up and compacts that space.