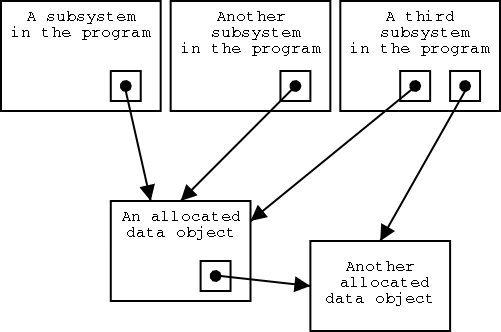
**Java Virtual Machine Memory Management**

**Introduction**

Java is one of the most common programming platforms in use today. There are several key features that have caused the number of devices running Java to soar will into the billions, one of those features is Java’s memory management structure. The Java Virtual Machine (JVM) stores memory in a heap structure, which can be defined by the user exactly how large or small the memory heap should be. The parameters that the user defines for the memory heap will also factor into how frequently the process of garbage collection occurs. Although automatic garbage collection is not required for a platform to run efficiently, a good garbage collection algorithm will provide a much higher throughput than average manual memory management. In this paper we will be exploring the history and motivation behind automatic memory management, as well as looking into how the JVM implements automatic garbage collection in detail.

**Background**

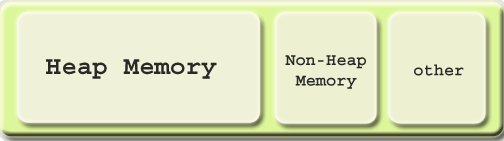
Appropriate memory management has always been an important topic for computer programs to be aware of, and traditionally it has been the responsibility of an application developer to allocate and de-allocate memory as needed for a program. With the rise in popularity of object-oriented programming, memory allocation has become increasingly important. Older programming languages such as C typically use manual memory management. It may seem simple to free allocated data objects once they are no longer needed by the program, however, especially in large systems, this can become very difficult.

Consider a scenario where a user interface allocates a data object, and the newly allocated object needs to be used by several other subsystems after the user interface process has completed. Which subsystem should then be responsible for managing the data object after the originating process completes?

Even the strongest programmers have struggled with situations such as this, and any mistake in manual memory management results in a memory leak. In today’s enterprise computing systems, high availability is of increasing importance. A common reason why a system may have to be restarted is due to memory leaks overrunning the memory space allocated for a program. Perhaps the most effective way to reduce the occurrence of memory leaks is through automatic garbage collection.

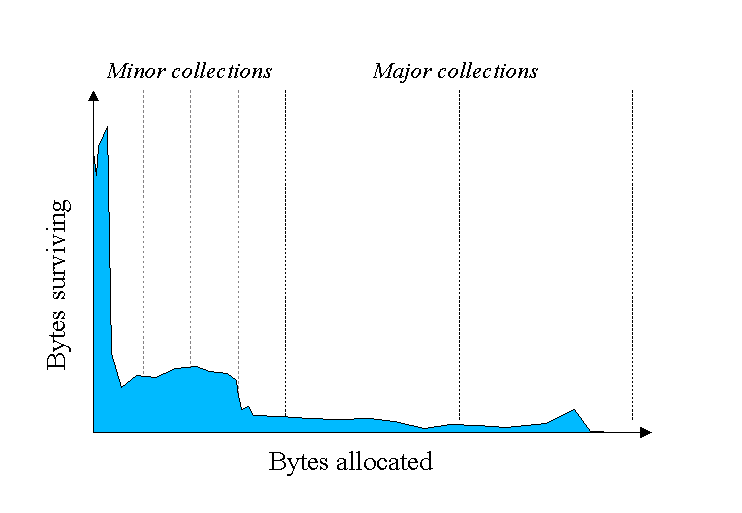
The concept of automatic garbage collection originated with John McCarthy’s 1959 implementation of Lisp. Since then, the Java platform has adopted a garbage collector into the JVM which allows developers to focus efforts on other aspects of coding, rather than worrying about memory management. Drawbacks to automatic garbage collection include non-deterministic scheduling, scattered throughput, and computational overhead. However, in most cases the advantages outweigh the disadvantages, because garbage collection frees the programmer from dealing with manual memory management. This effectively eliminates dangling pointer bugs, double free bugs, and basic memory leaks.

**The Java Virtual Machine Memory Structure**

The JVM memory structure consists of 3 segments: heap memory, non-heap memory, and internal memory. The heap memory is where Java objects are stored and is the most challenging of the three memory segments to manage; it is also the largest segment of the JVM’s memory structure. Non-Heap is used by the JVM to store loaded classes and metadata. The internal memory segment is where the JVM code resides, along with any data structures that the JVM requires in order to operate. All three of these memory segments are created when the JVM starts up, and persist for the full lifecycle of the JVM.

The initial and maximum heap size can be configured using –Xmx<size> and –Xms<size> respectively. By default, the maximum heap size in the JVM is 64Mb.

Since the non-heap and internal memory management is rather non-interesting, we will be looking in depth at the heap memory. However, before talking about how the heap is structured, it is important to note the behaviors of what data will be contained in the heap. From a memory management perspective, it would be useful to understand how long the objects in our heap are surviving for, so garbage collection may be optimized accordingly.



The left side of this graph is indicating that most objects have a very short lifespan, and very few objects survive for long periods of time. The chart above has been divided into different sections, or generations, so that objects of similar age can have their own management policy. This becomes particularly useful when considering the process of garbage collection. Running garbage collection over all objects can incur a significant overhead, but if young objects are grouped together, separate garbage collection timers may be tuned appropriately for each generation.

**Works Cited:**

<https://www.java.com/en/about/>

<http://www.memorymanagement.org/articles/lang.html>

<http://basen.oru.se/kurser/koi/2008-2009-p1/texter/gc/index.html>

<http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)>

<http://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html#wp1085825>

<http://www.ibm.com/developerworks/java/library/j-jtp01274/index.html>

<http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>