A Survey of Finalizers used for Garbage Collection

Bhavneet Soni

Harrisburg University of Science and Technology

Author’s Note

Review of Literature

Submitted to Dr. Majid Shaalan as part of First deliverable for the Course Project

Abstract

Purpose of this exercise is to understand how various programming languages deal with Garbage collection. Finalizers also known as destructors are called in to garbage collect redundant objects and variables. For the survey, we have reviewed multiple resources of information about the use of Garbage collectors, principles behind the automatic garbage collecting techniques, how different programming languages implement or not implement these techniques. So far, we are reviewed the work done by various researchers on how the process is handled and optimized by various software’s.

Keywords: Finalizer, destructor, garbage collection, semantics, memory management

A Survey of Finalizers used for Garbage Collection

Performance of a program or an application is dependent on many factors such as how efficient algorithms are implemented, how much memory is available, what’s the stack size etc. However, no matter how efficient an algorithm is or how much computing power is available the efficiency of the program will diminish if it is not properly managed. One of the most important methods of managing resources is removing entities that are no longer required by the program.

Earlier languages such as C++ didn’t offer any garbage collection and the user had to write their own collectors and do memory management. This lead to run away memory issues and put the pressure on the programmer to keep track of all the memory objects created and when to destroy them when no longer needed. More younger languages like JAVA, C# have provisions for addressing these issues and do meaningful memory management for freeing up the programmer to focus on other tasks, have come up with automatic garbage collection [1].

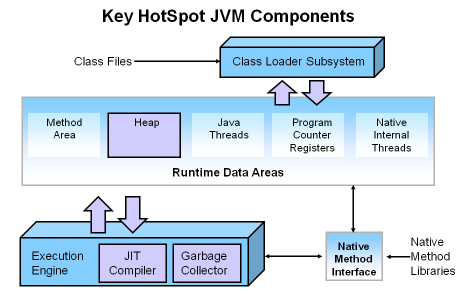


Figure 1Memory Space in a JVM

To understand how memory can be optimized we need to understand how it is designed and how it operates. A memory space of a Java virtual machine (JVM) is called heap. Heap is managed by Execution engine, Figure 1 Memory Space in a JVM, shows how memory heap is managed by java. Memory heap consists of Eden space and survivor space. When a new object is created in JVM allocates it to Eden space. JVM runs a Mark and sweep cycle which will go over all the objects in the heap space, all the objects that are not reachable are removed from the memory. Objects that survive the first sweep are moved to survivor space. This process of Mark-And-Sweep is called minor garbage collection step and has minor impact on the overall performance. However, Garbage collection brings in a serious drawback of freeze the world syndrome, where everything is stopped by the machine until GC is complete, which makes it very inefficient or undesirable for banking transactions. Another issue with the GC is Reference counting, is caused because of the additional overhead more so relevant in Multi-Threaded environments.

There are additional research work carried out and have Eiffel implementers [2] that try to have benefits of Reference counting and Mark and sweep without the “Freezing” drawback.

We will try to understand the semantics and sequence of finalizing steps, understand how the objects are generally stored in the program memory heap and how these are discarded when not in use by the program. We will review the specifications of the afore mentioned languages along with a few papers written by Leal and Ierusalimschy [3], Blackburn [4] to name a few.

Garbage collector helps in managing allocation and reclaiming of memory,It makes a trip to the heap and collects all objects that are no longer needed by application and its makes than free from memory.Thats why when you create any object in c# CLR allocates memory for the object from heap .This process goes on for every newly created object.As memory is limited so we need to clean some used space in order to make room for other new objects.So we use here garabage collector freeing of space for other resources.

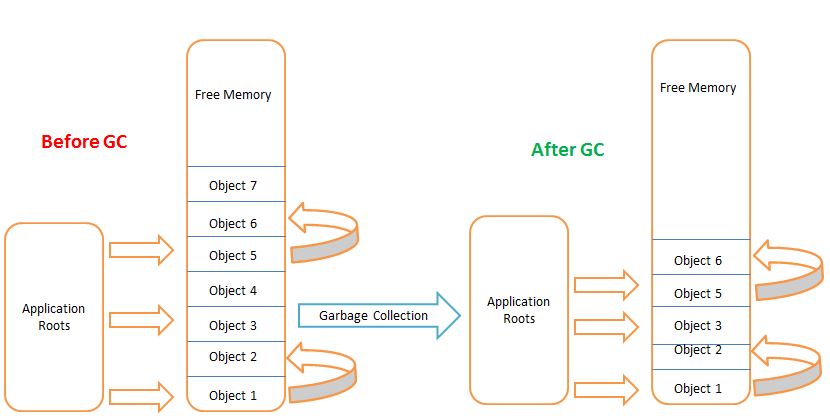
Garbage collection consists of the following steps:

The garbage collector searches for managed objects that are referenced in managed code.

The garbage collector attempts to finalize objects that are not referenced.

The garbage collector frees objects that are not referenced and reclaims their memory. n?

Garbage Collection diagram:-

[](http://www.csharpstar.com/wp-content/uploads/2016/03/Garbage_Collection.jpg?189db0)

When a program starts, the system allocates some memory for the program to get executed. When a C# program instantiates a class, it creates an object.The program manipulates the object, and at some point the object may no longer be needed.When the object is no longer accessible to the program and becomes a candidate for garbage collection.There are two places in memory where the CLR stores items while your code executes.

Stack

Heap

The stack keeps track of what’s executing in your code (like your local variables), and the heap keeps track of your objects.Value types can be stored on both the stack and the heap.For an object on the heap, there is always a reference on the stack that points to it.The garbage collector starts cleaning up only when there is not enough room on the heap to construct a new object.

The stack is automatically cleared at the end of a method. The CLR takes care of this and you don’t have to worry about it.The heap is managed by the garbage collector.In unmanaged environments without a garbage collector, you have to keep track of which objects were allocated on the heap and you need to free them explicitly. In the .NET Framework, this is done by the garbage collector.

References

|  |  |
| --- | --- |
| [1] | "JAVA SE 9 Specifications," Oracle, [Online]. Available: https://docs.oracle.com/javase/specs/jls/se9/html/jls-12.html#jls-12.6. |
| [2] | "Eiffel Software," [Online]. Available: https://www.eiffel.org/doc/eiffelstudio/Release%20notes%20for%20EiffelStudio%2017.05. |
| [3] | M. A. Leal and R. Ierusalimschy, "A Formal Semantics for Finalizers," *Journal of Universal Computer Science, vol. 11, no. 7,* pp. 1198-1214, 2005. |
| [4] | J. N. Z. a. S. M. Blackburn, "Java Finalize Method, Orthogonal Persistence and Transactions," *Department of Computer Science Australian National University Canberra ACT 0200 Australia.* |
| [5] | "Microsoft Develoer Network," Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/ms235315.aspx. |
| [6] | T. Ramananandro, "Mechanized Formal Semantics and Verified Compilation for C++ Objects". |