To help optimize the process, each object on the heap is assigned to a specific “generation.” The idea behind generations is simple: the longer an object has existed on the heap, the more likely it is to stay there.

For example, the class that defined the main window of a desktop application will be in

memory until the program terminates. Conversely, objects that have only recently been placed on the heap (such as an object allocated within a method scope) are likely to be unreachable rather quickly.

Given these assumptions, each object on the heap belongs to one of the following generations:

• *Generation 0*: Identifies a newly allocated object that has never been marked for collection.

• *Generation 1*: Identifies an object that has survived a garbage collection (i.e., it was

marked for collection but was not removed due to the fact that the sufficient heap

space was acquired).

• *Generation 2*: Identifies an object that has survived more than one sweep of the garbage collector.

The garbage collector will investigate all generation 0 objects first. If marking and sweeping (or said more plainly, getting rid of) these objects results in the required amount of free memory, any surviving objects are promoted to generation 1.

To see how an object’s generation affects the collection process,

If all generation 0 objects have been evaluated, but additional memory is still required, generation 1 objects are then investigated for reachability and collected accordingly. Surviving generation 1 objects are then promoted to generation 2. If the garbage collector *still* requires additional memory, generation 2 objects are evaluated.

At this point, if a generation 2 object survives a garbage collection, it remains a generation 2 object, given the predefined upper limit of object generations.

The bottom line is that by assigning a generational value to objects on the heap, newer objects (such as local variables) will be removed quickly, while older objects are

not “bothered” as often.

The System.GC Type

The mscorlib.dll assembly provides a class type named System.GC that allows you to programmatically interact with the garbage collector using a set of static members. Now, do be very aware that you will seldom (if ever) need to make use of this class directly in your code. Typically, the only time you will use the members of System.GC is when you are creating classes that make internal use of *unmanaged resources*.

Forcing a Garbage Collection

Again, the whole purpose of the .NET garbage collector is to manage memory on our behalf. However, in some very rare circumstances, it may be beneficial to programmatically force a garbage collection using

**GC.Collect().**

**Finalize() will (eventually) occur during a “natural” garbage collection or** possibly when you programmatically force a collection via GC.Collect().

1. You should not implement the Finalize method until it is extremely necessary.
2. At runtime C#, C++ destructors are automatically Converted to Finalize method. But in VB.NET you need to override Finalize method, since it does not support destructor.
3. You should not implement a Finalize method for managed objects, because the garbage collector cleans up managed resources automatically.