The .NET Framework's garbage collector manages the allocation and release of memory for your application. Each time you create a new object, the common language runtime allocates memory for the object from the managed heap. As long as address space is available in the managed heap, the runtime continues to allocate space for new objects. However, memory is not infinite. Eventually the garbage collector must perform a collection in order to free some memory. The garbage collector's optimizing engine determines the best time to perform a collection, based upon the allocations being made. When the garbage collector performs a collection, it checks for objects in the managed heap that are no longer being used by the application and performs the necessary operations to reclaim their memory.

Mark and Sweep Algorithm

Marking Phase

When a garbage collection starts, it looks at a set of references called the ‘GC roots’. These are memory locations that are designated to be always reachable for some reason, and which contain references to objects created by the program. It marks these objects as ‘live’ and then looks at any objects that they reference; it marks these as being ‘live’ too. It continues in this manner, iterating through all of the objects it knows are ‘live’. It marks anything that they reference as also being used until it can find no further objects.

Sweep Phase

Once all of these live objects are known, any remaining objects can be discarded and the space re-used for new objects. .NET compacts memory so that there are no gaps (effectively squashing the discarded objects out of existence) – this means that free memory is always located at the end of a heap and makes allocating new objects very fast.

Conditions for a garbage collection

Garbage collection occurs when one of the following conditions is true:

* The system has low physical memory.
* The memory that is used by allocated objects on the managed heap surpasses an acceptable threshold. This threshold is continuously adjusted as the process runs.
* The [GC.Collect](https://msdn.microsoft.com/en-us/library/xe0c2357(v=vs.110).aspx) method is called. In almost all cases, you do not have to call this method, because the garbage collector runs continuously. This method is primarily used for unique situations and testing.

Generation

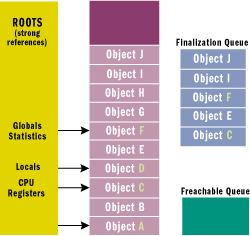
A generational garbage collector collects the short-lived objects more frequently than the longer lived ones.  Short-lived objects are stored in the first generation, generation 0.  The longer-lived objects are pushed into the higher generations, 1 or 2.  The garbage collector works more frequently in the lower generations than in the higher ones.

When an object is first created, it is put into generation 0.  When the generation 0 is filled up, the garbage collector is invoked. The objects that survive the garbage collection in the first generation are promoted onto the next higher generation, generation 1.  The objects that survive garbage collection in generation 1 are promoted onto the next and the highest generation, generation 2.  This algorithm works efficiently for garbage collection of objects, as it is fast.  Note that generation 2 is the highest generation that is supported by the garbage collector.

**SHORT CONCEPTS::**

1. Objects NOT implementing Finalize methods, there Memory is reclaimed immediately,unless of course, they are not reacheable by  
   application code anymore
2. Objects implementing Finalize Method, The Concept/Implementation of Application Roots, Finalization Queue, Freacheable Queue comes before they can be reclaimed.
3. Any object is considered garbage if it is NOT reacheable by Application Code

Assume:: Classes/Objects A, B, D, G, H do NOT implement Finalize Method and C, E, F, I, J implement Finalize Method.

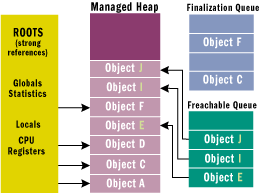
When an application creates a new object, the new operator allocates the memory from the heap. **If the object's type contains a*Finalize*method, then a pointer to the object is placed on the finalization queue**.  
  
therefore pointers to objects C, E, F, I, J gets added to finalization queue.  
  
The **finalization queue** is an internal data structure controlled by the garbage collector. Each entry in the queue points to an object that should have its Finalize method called before the object's memory can be reclaimed. Figure below shows a heap containing several objects. Some of these objects are reachable from the **application's roots**, and some are not. When objects C, E, F, I, and J were created, the .Net framework detects that these objects have Finalize methods and pointers to these objects are added to the **finalization queue**.  
  


When a GC occurs(1st Collection), objects B, E, G, H, I, and J are determined to be garbage. Because A,C,D,F are still reacheable by Application Code depicted through arrows from yellow Box above.

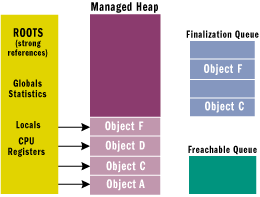
The garbage collector scans the **finalization queue** looking for pointers to these objects. **When a pointer is found, the pointer is removed from the finalization queue and appended to the freachable queue** ("F-reachable").   
  
The **freachable queue** is another internal data structure controlled by the garbage collector. Each pointer in the **freachable queue** identifies an object that is ready to have its Finalizemethod called.

After the collection(1st Collection), the managed heap looks something similar to figure below. Explanation given below::  
1.) The memory occupied by objects B, G, and H has been reclaimed immediately because these objects did not have a finalize method that needed to be called.

2.) However, the memory occupied by objects E, I, and J could not be reclaimed because their*Finalize*method has not been called yet. Calling the Finalize method is done by **freacheable queue.**

3.) A,C,D,F are still reacheable by Application Code depicted through arrows from yellow Box above, So they will NOT be collected in any case  
  


There is a special runtime thread dedicated to calling Finalize methods. When the freachable queue is empty (which is usually the case), this thread sleeps. But when entries appear, this thread wakes, removes each entry from the queue, and calls each object's Finalize method. The garbage collector compacts the reclaimable memory and the special runtime thread empties the***freachable***queue, executing each object's*Finalize*method.***So here finally is when your Finalize method gets executed***

The next time the garbage collector is invoked(2nd Collection), it sees that the finalized objects are truly garbage, since the application's roots don't point to it and the freachable queue no longer points to it(it's EMPTY too), Therefore the memory for the objects (E, I, J) are simply reclaimed from Heap.See figure below and compare it with figure just above  
  


The important thing to understand here is that two GCs are required to reclaim memory used by***objects that require finalization***. In reality, more than two collections cab be even required since these objects may get promoted to an older generation

**NOTE::** The***freachable queue***is considered to be a root just like global and static variables are roots. Therefore, if an object is on the freachable queue, then the object is reachable and is not garbage.

As a last note, remember that debugging application is one thing, Garbage Collection is another thing and works differently. So far you can't FEEL garbage collection just by debugging applications, further if you wish to investigate Memory get [started here.](http://www.diaryofaninja.com/blog/2014/03/20/investigating-aspnet-memory-dumps-for-idiots-like-me)

<https://msdn.microsoft.com/en-us/library/498928w2(v=vs.110).aspx>

in most cases, the garbage collector can determine the best time to perform a collection, and you should let it run independently. There are rare situations when a forced collection might improve your application's performance. In these cases, you can induce garbage collection by using the [GC.Collect](https://msdn.microsoft.com/en-us/library/xe0c2357(v=vs.110).aspx) method to force a garbage collection.