**Garbage Collection And The Dispose Pattern**

**Introduction :**

Experts have written hundreds of articles on garbage collection and dispose pattern. This is NOT one of them. I had so many questions around this topic. While understanding this concept I came across many explanations, many of them were very good. I thought of collecting all of them in points so it would help another newbie like me. This article targets .NET Framework and sample codes are written in c# language.

What is garbage collection?

* When we create any object in C#, CLR (common language runtime) allocates memory for the object from heap. Garbage collector manages allocation and reclaiming of memory.
* GC (Garbage Collector) makes a trip to the heap and collects all objects that are no longer used by the application and then makes them free from memory.
* GC works on managed heap, which is nothing but a block of memory to store objects.

How it works?

* Heap is managed by different **'Generations'**, it stores and handles long-lived and short-lived objects.
* 0 Generation (Zero): This generation holds short-lived objects, e.g., Temporary objects. GC initiates garbage collection process frequently in this generation.
* 1 Generation (One): This generation is the buffer between short-lived and long-lived objects.
* 2 Generation (Two): This generation holds long-lived objects like a static and global variable that needs to be persisted for a certain amount of time.
* Objects which are not collected in generation Zero, are then moved to generation 1, such objects are known as survivors, similarly objects which are not collected in generation One, are then moved to generation 2 and from there onwards objects remain in the same generation.

How GC decides if objects are live?

* It collects all handles of an object that are allocated by user code or by CLR.
* Keeps track of static objects, as they are referenced to some other objects.
* Use stack provided by stack walker and JIT.

When GC gets triggered?

* When virtual memory is running out of space.
* When allocated memory is surpassed acceptable threshold (when GC found if the survival rate (living objects) is high, then it increases the threshold allocation).
* When we call **GC.Collect()** method explicitly.
* GC.Collect() forces an immediate garbage collection of all generations.
* Objects that are referenced in managed code are not collected.
* GC.Collect() has nothing to do with unmanaged resources.

What are the phases of GC?

* **Mark phase**: Walk the GC roots to find all live objects and mark them as live.
* **Sweep phase**: Walk all of the objects in the GC heap
* Dead objects are either freed or moved to the finalizer queue if finalization is required.
* Note that at this point, we may be able to allocate GC objects on the GC heap IFF enough contiguous memory was freed; however, no memory was returned to the operating system.
* **Compaction phase**: Move the objects in the heap to reduce the total amount of memory used.
* The goal of the compaction is to remove all fragmentation in the heap.
* The GC can not move objects that are pinned. This can be caused by the application explicitly pinning an object or if the object was passed as a parameter to a p/invoke call that is occurring during the GC
* At this point, the GC will return blocks of free memory to the operating system.
* Compactions are not performed on every GC; however, they are performed based on a heuristic of heap fragmentation.
* **Pitching JIT compiled code phase:** This phase occurs as a last resort and results in all JIT compiled code except for the methods on the current thread stacks to be released. The methods will be recompiled on a demand basis.

What are the problems with GC.Collect()?

* Calling GC.Collect() forces the CLR to do a stack walk to see if each object can be truly be released by checking references.
* The time it takes the CLR to do a GC is a function of both the number of live and dead objects.
* Applications main thread (and any child threads it created) are frozen so the GC can walk the stack. The more time app spends in GC, the more time it spends frozen.

When is it acceptable to call GC.Collect()?

* Consider calling GC.Collect() if some non-recurring event has just happened and this event is highly likely to have caused a lot of old objects to die.
* A classic example of this is if you’re writing a client application and you display a very large and complicated form that has a lot of data associated with it.  Your user has just interacted with this form potentially creating some large objects… things like XML documents, or a large DataSet or two.  When the form closes these objects are dead and so GC.Collect() will reclaim the memory associated with them.

What are unmanaged resources?

* Managed objects are created, managed and under scope of CLR.
* Unmanaged objects are created outside the control of .NET libraries and are not managed by CLR.
* An allocation of any memory block to unmanaged resource is invisible to the garbage collector.
* Example of such unmanaged code is COM objects, file streams, connection objects, Interop objects.

What is the role of **Finalizer** or **Destructor**?

* MSDN says: Finalize method allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.
* In other words the Finalize method is used to perform cleanup operations on unmanaged resources held by the current object before the object is destroyed.
* The C# compiler does not allow you to override the Finalize method. Instead, you provide a finalizer by implementing a destructor for your class. A C# destructor automatically calls the destructor of its base class.
* Syntax of destructor

~SampleClass()

{

//release unmanaged resources here

}

* The exact time when the finalizer executes is undefined.

When to use **Destructor**?

* You should use destructor for a class that uses unmanaged resources such as file handles or database connections that must be released when the managed object that uses them is discarded during garbage collection. Garbage collector cleans up managed resources automatically.
* The objects that have finalizers (and if a call to GC.SuppressFinalize method hasn’t been made) would not be collected when a call to GC.Collect() method is made. Rather, such objects would be placed in the finalization queue. If you would like to collect those objects as well, you would need to make a call to the GC.WaitForPendingFinalizers() method so that those objects are cleaned up when the next GC cycle runs. In essence, reclaiming the memory occupied by objects that have finalizers implemented requires two passes since such objects are placed in the finalization queue rather than being reclaimed in the first pass when the garbage collector runs.
* Never implement finalizers for objects that only contain other managed objects. Finalizers will delay when the GC can actually free the memory associated with an object.

What is the role of **Idisposable.Dispose** method?

* The exact time when finalizer will be called is nondeterministic.
* Framework provides the [**System.IDisposable**](https://msdn.microsoft.com/en-us/library/system.idisposable(v=vs.110).aspx) interface that should be implemented to provide the developer a manual way to release unmanaged resources as soon as they are not needed
* Dispose is used to deterministically free up unmanaged resources.
* It doesn't collect the object's memory (that still belongs to GC) - but is used for example to close files, database connections, etc
* Dispose method is NOT called automatically unless the object is instantiated inside using block. In other cases we should call it manually.

What is the **Dispose pattern**?

* A design pattern is a general reusable solution to a commonly occurring problem in software design.
* The **Dispose Pattern** is intended to standardize the usage and implementation of finalizers and the IDisposable interface.
* Examples for dispose pattern
* When implementing a sealed class that doesn't use unmanaged resources, we simply implement a Dispose method as with normal interface implementations.

namespace Garbage\_Collection

{

public sealed class SampleClass : IDisposable

{

public void Dispose()

{

// get rid of managed resources, call Dispose on member variables...

}

}

}

* When implementing an unsealed class we should only implement a finalizer if you have actual unmanaged resources to dispose.but we give inheritors of our class a hook to call our Dispose and implement a finalizer themselves if they use unmanaged resources directly.

namespace Garbage\_Collection

{

public class BaseSampleClass : IDisposable

{

public void Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

protected virtual void Dispose(bool disposing)

{

if (disposing)

{

// get rid of managed resources

}

// get rid of unmanaged resources

}

// only if you use unmanaged resources directly in BaseSampleClass

//~BaseSampleClass()

//{

// Dispose(false);

//}

}

}

namespace Garbage\_Collection

{

class ChildSampleClass : BaseSampleClass

{

private IntPtr m\_Handle;

protected override void Dispose(bool disposing)

{

if (disposing)

{

// get rid of managed resources

}

ReleaseHandle(m\_Handle);

base.Dispose(disposing);

}

~ChildSampleClass()

{

Dispose(false);

}

}

}

What is the use of ‘using’ statement?

* using statement provides a convenient syntax that ensures the correct use of IDisposable objects.
* As a rule, when you use an IDisposable object, you should declare and instantiate it in a using statement
* When you are using an object that encapsulates any resource, you have to make sure that when you are done with the object, the object's Dispose method is called.
* The using statement calls the Dispose method on the object in the correct way, and it also causes the object itself to go out of scope as soon as Dispose is called.
* The using statement ensures that Dispose is called even if an exception occurs while you are calling methods on the object.
* Remember that the objects you instantiate must implement the System.IDisposable interface otherwise we will get "type used in a using statement must be implicitly convertible to 'System.IDisposable'"
* Example:

namespace Garbage\_Collection

{

class Program

{

static void Main(string[] args)

{

using (var obj = new ChildSampleClass())

{

//work on the object here

//once the control leaves this block Dispose method on

//this object is automatically called

}

}

}

}

**Conclusion :**

I believe the points here help you understand garbage collection and the dispose pattern. Please refer to the links provided below to explore the topic even more. This document is located at https://github.com/shreesharao/Articles/Garbage Collection And The Dispose Pattern.doc and is always open for improvements.

**References :**

* [Garbage Collection in .Net framework](http://www.c-sharpcorner.com/uploadfile/skumaar_mca/garbage-collection-in-net-framework/)
* [The perils of GC.Collect (or when to use GC.Collect)](https://blogs.msdn.microsoft.com/scottholden/2004/12/28/the-perils-of-gc-collect-or-when-to-use-gc-collect/)
* [Two things to avoid for better memory usage](https://blogs.msdn.microsoft.com/ricom/2003/12/02/two-things-to-avoid-for-better-memory-usage/)
* [My two cents on GC.Collect method in C#](http://www.infoworld.com/article/3099113/application-development/my-two-cents-on-gc-collect-method-in-c.html)
* [Object.Finalize Method ()](https://msdn.microsoft.com/en-us/library/system.object.finalize(v=vs.110).aspx)
* [Dispose Pattern](https://msdn.microsoft.com/en-us/library/b1yfkh5e(v=vs.110).aspx)
* [Finalize/Dispose pattern in C#](http://stackoverflow.com/questions/898828/finalize-dispose-pattern-in-c-sharp)
* [Proper use of the IDisposable interface](http://stackoverflow.com/questions/538060/proper-use-of-the-idisposable-interface)
* [using Statement (C# Reference)](https://msdn.microsoft.com/en-us/library/yh598w02.aspx)
* [Understanding the 'using' statement in C#](https://www.codeproject.com/Articles/6564/Understanding-the-using-statement-in-C)
* [Uses of “using” in C#](http://stackoverflow.com/questions/75401/uses-of-using-in-c-sharp)