OBJECT LIFETIME

TOPICS COVERED:

- Understanding Object Lifetime
- The CIL of "new'
- The Basics of Garbage Collection
- Finalization a Type
- The Finalization Process
- Building an Ad Hoc Destruction Method
- Garbage Collection Optimizations
- The System. GC Type.

Understanding Object Lifetime:

- Every program needs memory. When a program starts, the system allocates some memory for the program to get executed.
- 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.
- **Example -1:**

```
public static int Main(string[] args)
{
//create a local car object     //place an object onto the managed heap
Car c = new Car("zen",200,100);
} // if 'c' is the only reference to the Car object, it may be destroyed when   // Main exits
```

Example -2:

```
public static void Main()
{
Car c=new Car("Ford Ikon");
……
}
```

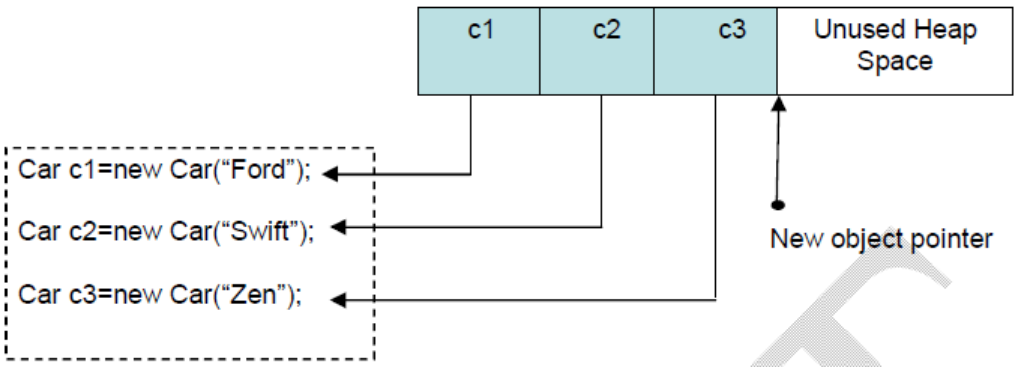
- Once the object is created then allocate the memory for processing. Heap keeps track of your objects
- Programmers never directly de-allocate an object from memory (therefore there is no "delete" keyword).
- Object lifetime is the time when a block of memory is allocated to this object during some process of execution and that block of memory is released when the process ends.

The CIL of *new:*

- When C# encounters the *new* keyword, it will produce a CIL **new object** instruction to the code module.
- **CIL** to allocate region-of-memory for each created object termed as the 'managed-heap'.
- The heap maintains a new-object-pointer(NOP)

  NOP points to next available slot on the heap where the next object will be placed.
- **newobj** instruction informs CLR to perform following sequence of events:

1. CLR calculates total memory required for the new object to be allocated.
   - If this object contains other internal objects, their memory is also taken into account.
   - Also, the memory required for each base class is also taken into account.
2. Then, CLR examines heap to ensure that there is sufficient memory to store the new object. If yes, the object's constructor is called and a reference to the object in the memory is returned (which just happens to be identical to the last position of NOP).
3. Finally, CLR advances NOP to point to the next available slot on the heap.



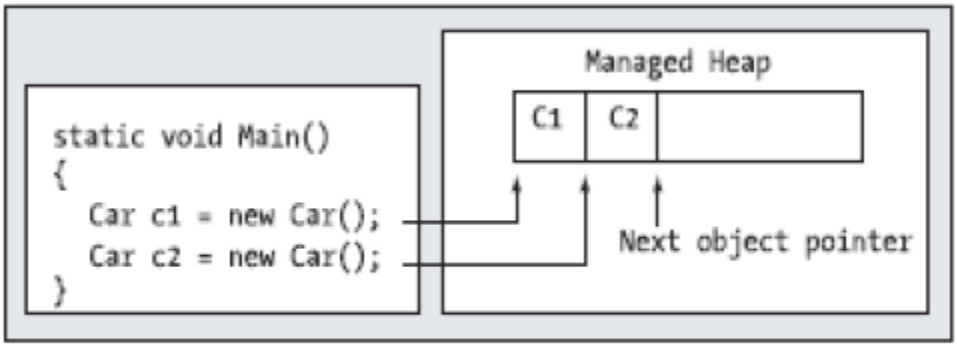
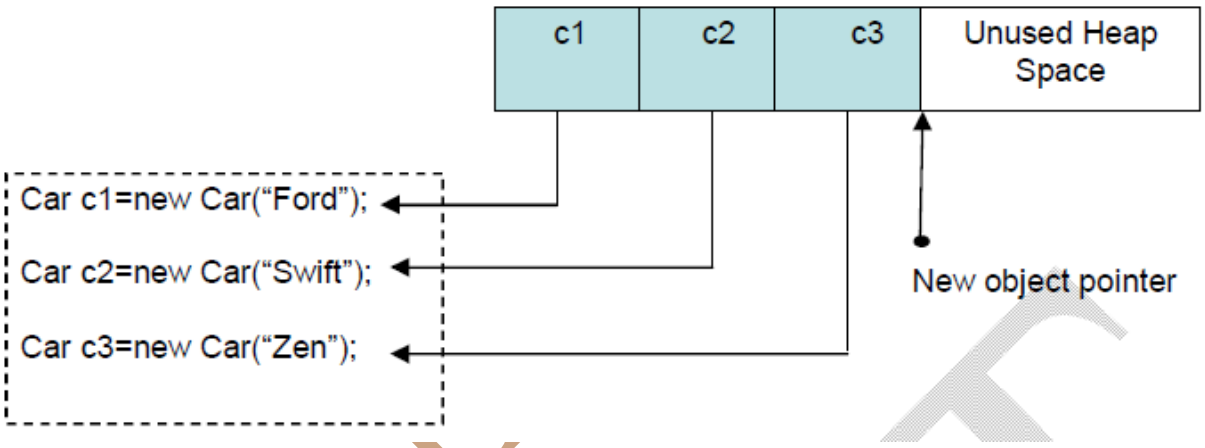


Figure 5-2. *The details of allocating objects onto the managed heap*

The basics of Garbage collection:



- After creating so many objects, the managed heap may become full.
- When the **newobj** instruction is being processed, if the CLR determines that the managed heap does not have sufficient memory to allocate the requested type, it will perform a garbage collection.

What is Garbage Collection and Why We Need It?

When you create any object in C#, CLR (*common language runtime*) allocates memory for the object from heap. This process is repeated for each newly created object, but there is a limitation to everything, Memory is not un-limited and we need to clean some used space in order to make room for new objects,

Here, the concept of *garbage collection* is introduced, *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.

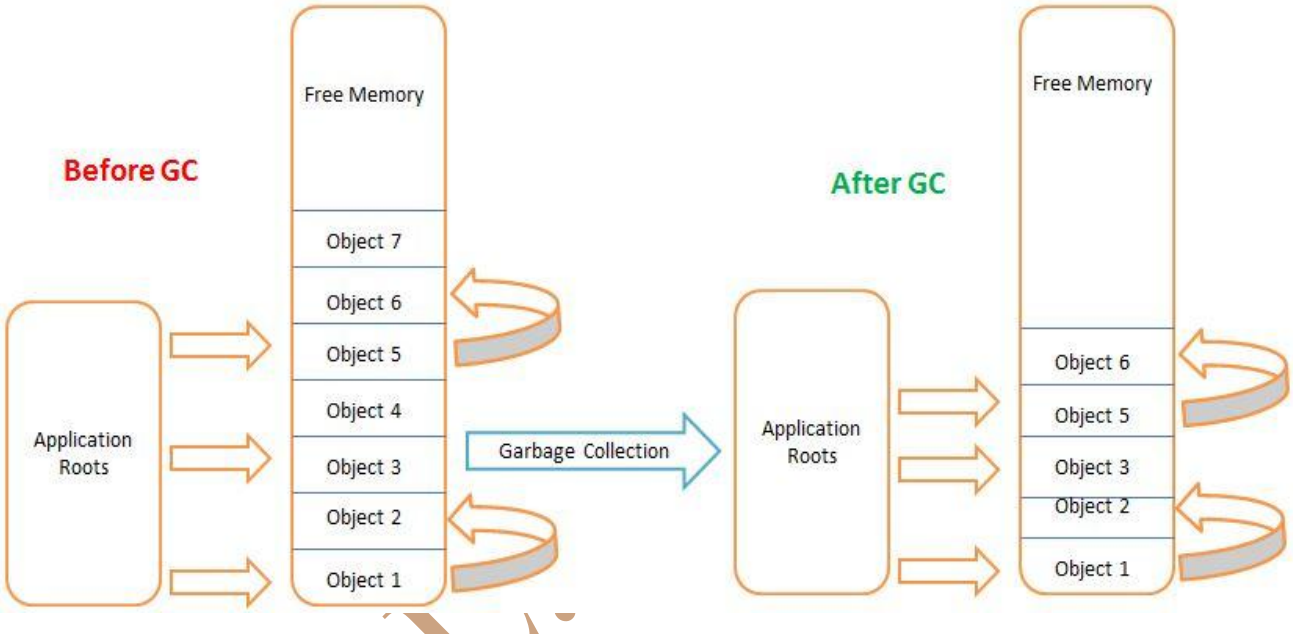
The next rule of garbage collection is: If the managed heap does not have sufficient memory to allocate a new object, a garbage collection will occur.

- Garbage collection is a process of reclaiming memory occupied by unreachable objects in the application.
- **Now the question is,** How CLR is able to determine an object on the heap that it is no longer needed and destroy it?
- To answer this question, we should know **application roots.**
- A **root** is a storage location containing a reference to an object on the heap. In other words, a **root** is a variable in our application that points to some area of memory on the managed heap.
- **The root can fall into any of the following categories:**
    - Reference to global objects
    - Reference to static objects
    - References to local objects within a given method
    - References to object parameters passed into a method
    - Any CPU register that references a local object
- When a garbage-collection occurs, the CLR will inspect all objects on the heap to determine if it is still in use in the application.
- To do so, the CLR will build an object-graph. Object-graph represents each object on the heap that is still reachable.
- When garbage-collector determines that a given root is no longer used by a given application, the object is marked for termination.
- When garbage-collector searches the entire heap for orphaned-root, the underlying memory is reclaimed for each unreachable object.

- Then, garbage-collector compacts empty block of memory on the heap (which in turn will cause CLR to modify the set of application roots to refer to the correct memory location).
- Finally, the NOP is re-adjusted to point to the next available slot.

How the Garbage Collector works?

- Consider the below diagram to easily understand garbage collector works:



When Garbage collector runs:

Marking Phase:

- It marks all the heap memory as not in use
- Then examines all the programs reference variables, parameters that has object reference, CPU registers and other items that point to heap objects

Relocating Phase:

- For each references, the garbage collector marks the object to which the reference points as in use

Compact Phase:

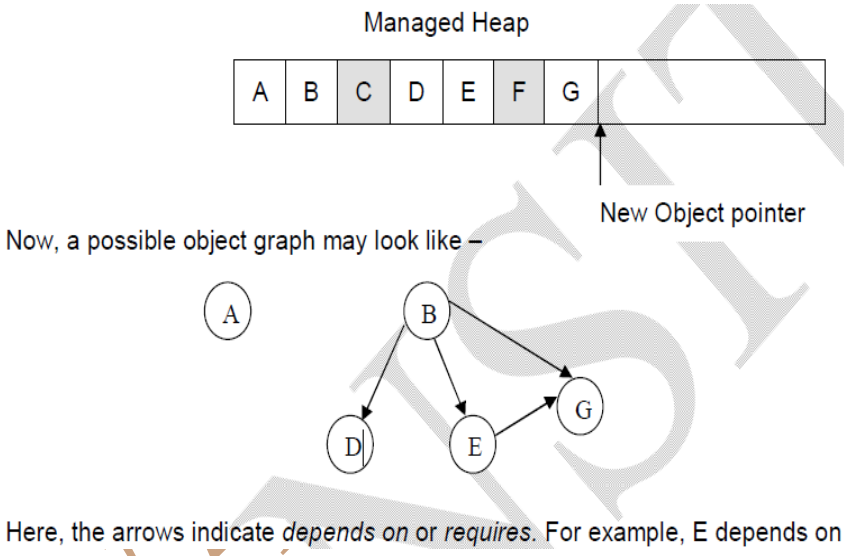
- Then it compacts heap memory that is still in use and updates program reference
- Garbage collector updates the heap itself so that the program can allocate memory from unused portion

Before Garbage Collector Runs:

- In the above diagram, Before Garbage collector runs, the application root has dependency on object 1, object 3 and object 5.
- Object 1 is dependent on object 2 and Object 5 is dependent on object 6. So the application root does not have any dependency on object 4 and object7.

After the Garbage collector runs:

- It discards Object 4 and Object 7 since there is no dependency exists and compact the heap memory.
- When it destroys an object, the garbage collector frees the object's memory and any unmanaged resource it contains.



Here, the arrows indicate *depends on* or *requires*. For example, E depends on G, B depends on E

- To illustrate this concept, assume that the heap contains a set of objects named A, B, C, D, E, F, and G (Figure 5.3).
- Let C and F are unreachable objects which are marked for termination as shown in the following diagram:

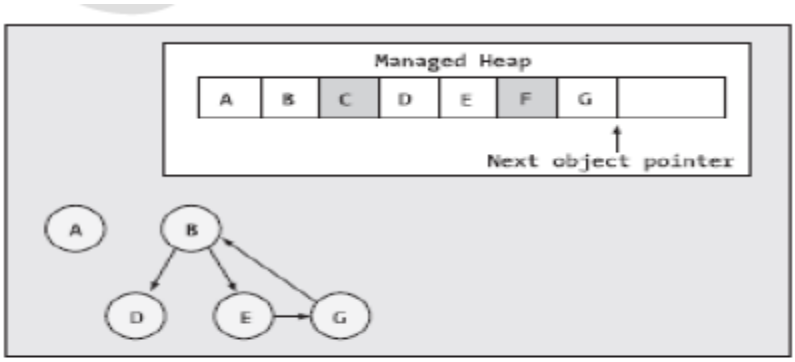


Figure 5.3: Before garbage collection

After garbage-collection, the NOP is readjusted to point to the next available slot as shown in the following diagram (Figure 5.4):

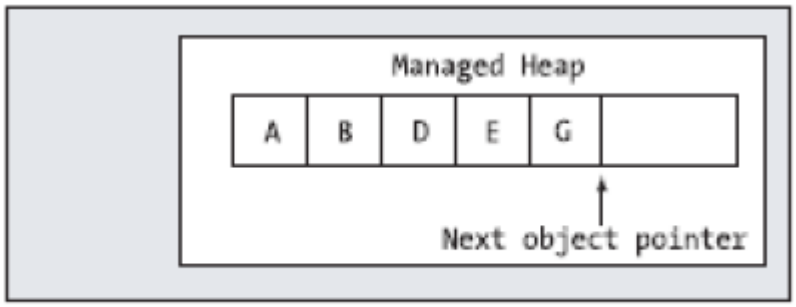


Figure 5.4: After garbage collection

Finalizing a Type:

- The .NET garbage collection scheme is ***non-deterministic*** in nature.[In other words, we cannot determine exactly when an object will be de-allocated from the memory.]
- This approach seems to be quite good because, we, the programmers need not worry once the object has been created.
- But, there is a possibility that the objects are holding *unmanaged resources* (Win32 files etc) longer than necessary.

Unmanaged resources like network connection, file handle, window handle etc..

- When we build .Net types that interact with unmanaged resources, we like to ensure that this resource is released in-time rather than waiting for .NET garbage collector.
- To facilitate this, the C# provides an option for overriding the virtual System.Object.Finalize() method.
- Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.
- The Finalize method is used to perform cleanup operations on unmanaged resources held by the current object before the object is destroyed. The method is protected and therefore is accessible only through this class or through a derived class.
- C# will not allow the programmer to directly override the Finalize() method.

```
public class Test
{
Protected override void Finalize ()     // Compile Time error!!!
{
…….
}
}
```

- we need to use a C++ -type destructor syntax:

```
public class Test
{
~Test()
{
……
}
}
```

Indirectly Invoking Finalize ()

- The .NET runtime will activate garbage collector when it requires more memory that what is available at heap.
- But also note that, the finalization will automatically take place when an ***application domain*** or ***AppDomain*** is unloaded by CLR.
- Application domain can be assumed to be Application itself. Thus, once our application is about to shut down, the finalize logic is activated.

```
using System;
class Test
{
public Test()
{ }
~Test()
{
Console.WriteLine("Finalizing!!!");
}
public static void Main()
{
Console.WriteLine(“Within Main()");
Test t=new Test();
Console.WriteLine(“Exiting Main()");
}
}
```

Output:

Within Main()

Exiting Main()

Finalizing!!!

The Finalization Process:

- When an object is placed on a heap using *new*, the CLR automatically determines whether this object supports a user-defined Finalize() method.

- If yes, the object is marked as ***finalizable*** and a pointer to this object is stored on an internal queue names as ***the finalization queue.***
- The finalization queue is a table maintained by the CLR that points to every object that must be finalized before it is removed from the heap.
- When the garbage collector starts its action, it checks every entry on the finalization queue and copies the object from the heap to another CLR-managed structure termed as ***finalization reachable table (f-reachable).***
- At this moment, a separate thread is produced to invoke the Finalize() method for each object on the f-reachable table at the *next garbage collection.*
- Thus, when we build a custom-type (user-defined type) that overrides the System.Object.Finalize() method, the .NET runtime will ensure that this member is called when our object is removed from the managed heap.
- But this will consume time and hence affects the performance of our application.

Building an Ad Hoc Destruction Method:

- The objects holding unmanaged resources can be destroyed using Finalize() method.
- But the process of finalization is time consuming.
- C# provides an alternative way to avoid this problem of time consumption.
- The programmer can write a custom ad hoc method that can be invoked manually before the object goes out of the scope.
- This will avoid the object being placed at finalization queue and avoid waiting for garbage collector to clean-up.
- The user-defined method will take care of cleaning up the unmanaged resources.

```
public class Car
{ ……..
public void Kill()      //name of method can be anything
{
//clean up unmanaged resources
}
}
```

The *IDisposable* Interface:

- IDisposable is an interface defined in the System namespace. The interface defines a single method, named "Dispose", that is used to perform clean-up activities for an object when it is no longer required. The method may be called manually, when an object's work is finished, or automatically during garbage collection.

```
public interface IDisposable
{
public void Dispose();
}
```

- Our application can implement this interface and define *Dispose()* method. Then, this method can be called manually to release unmanaged resources. Thus, we can avoid the problems with finalization.

```
public class Car: IDisposable
{ ……
public void Dispose()
{
//code to clean up unmanaged resources
}
}
class Test
{ ……
public static void Main()
{
Car c=new Car("Ford");
……
c.Dispose();
……
} //c still remains on the heap and may be collected by GC now
}
```

Another rule for working with garbage collection is:
Always call Dispose() for any object in the heap. The assumption is : if there is a Dispose() method, the object has some clean up to perform.

```
using System;
```

```
namespace UnmangedMemoryExample
{
    class DataConnectionManager : IDisposable
    {
        public DataConnectionManager() { }

        //Dispose method can be called explicitly or invoked implicitly at
the end of a using block
        public void Dispose()
        {
            //Clean up unmanaged resources
            Console.WriteLine("Here is where you clean up unmanaged
resources");
        }
    }
}
```

- Dispose() method can be called manually before the object goes out of scope.
- This method will take care of cleaning up the unmanaged resources.
- This method will

  → Avoid object being placed at finalization-queue &

  → Avoid waiting for garbage collector to clean-up

- C# provides another way of doing this with the help of using keyword:

```
public void Test()
{
using(Car c=new Car())
{
//Do something
//Dispose() method is called automatically when this block exits
}
}
```

- One good thing here is, the Dispose() method is called automatically when the program control comes out of using block.
- But there is a disadvantage: If at all the object specified at using does not implement IDisposable, then we will get compile-time error.

Garbage Collection Optimization:

- Till now we have seen two methodologies for cleaning user-defined types. they are Dispose method and Finalize method.
- When CLR is attempting to locate unreachable objects, it does not literally search through every object placed on the managed heap looking for orphaned roots.
- Because, doing so will consume more time for larger applications.
- To optimize the collection process, each object on the heap is assigned to a given ***generation***. [Basically, heap is managed by different '*Generations*', it stores and handles long-lived and short-lived objects][The managed heap is organized into three generations so that it can handle short lived and long lived objects efficiently. Garbage collector first reclaim the short lived objects that occupy a small part of the heap.]

The idea behind generation is as follows:

- If an object is on the heap since long time, it means, the object will continue to exist for more time. For example, application-level objects.
- Conversely, if an object has been recently placed on the heap, it may be dereferenced by the application quickly. For example, objects within a scope of a method.

Generation 0

This is the youngest generation and contains the newly created objects. Generation 0 has short-lived objects and collected frequently. The objects that survive the Generation 0 are promoted to Generation 1. [Identifies a newly allocated object that has never been marked for collection.]

Example: A temporary object.

Generation 1

This generation contains the longer lived objects that are promoted from generation 0. The objects that survive the Generation 1 are promoted to Generation 2. Basically this generation serves as a buffer

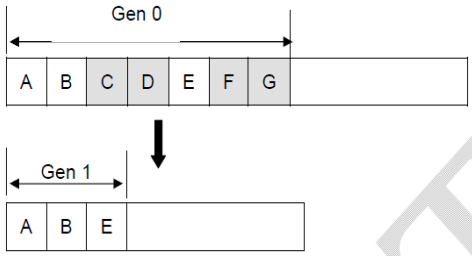
between short-lived objects and longest-lived objects.[Identifies an object that has survived a garbage collection sweep (i.e. it was marked for collection, but was not removed due to the fact that the heap had enough free space)]

Generation 2

This generation contains the longest lived objects that are promoted from generation 1 and collected infrequently.[Identifies an object that has survived more than on sweep of the garbage collector.]

Example : An object at application level that contains static data which is available for the duration of the process.

- Now, when garbage collection occurs, the GC marks and sweeps all generation-0 objects first.
- If this results in the required amount of memory, the remaining objects are promoted to the next available generation (G0->G1 & G1->G2).
- If all generation-0 objects have been removed from the heap, but more memory is still necessary, generation-1 objects are marked and swept, followed(if necessary) by generation-2 objects.



- If all generation 0 objects have been removed from heap and still more memory is necessary, generation 1 objects are checked for their 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 checked for their reachability.
- At this point, if generation 2 objects survive a garbage collection, they remain at that generation only.
- Thus, the newer objects (local variables) are removed quickly and older objects (application level variables) are assumed to be still in use.
- This is how, the GC is able to quickly free heap space using the generation as a baseline.

The System.GC Type:

The programmer can interact with the garbage collector using a base class ***System.GC***. This class provides following members:

- **Collect ():** This forces GC to call the Finalize() method for every object on managed-heap.
- **GetTotalMemory():** This returns the estimated amount of memory currently being used by all objects in the heap.
- **GetGeneration():** This returns the generation to which an object currently belongs.
- **MaxGeneration():** This returns the maximum generations supported on the target system.
- **ReRegisteredForFinalize():** This sets a flag indicating that the suppressed-object should be reregistered as finalize.
- **SuppersFinalize():** This sets a flag indicating that a given object's Finalize() method should not be called.
- **WaitForPendingFinalizers():** This suspends the current thread until all finalizable objects have been finalized. This method is typically called directly after invoking GC.Collect().

Building Finalization and Disposable Types:

Consider an example to illustrate how to interact with .NET garbage collector.

```
using System;
class Car:IDisposable
{
string name;
public Car(string n)
{
name=n;
}
~Car()
{
Console.WriteLine("Within destructor of {0}", name);
}
public void Dispose()
{
Console.WriteLine("Within Dispose() of {0}", name);
GC.SuppressFinalize(this);
}
```

```
}
class Test
{
public static void Main()
{
Car c1=new Car("One");
Car c2=new Car("Two");
Car c3=new Car("Three");
Car c4=new Car("Four");
c1.Dispose();
c3.Dispose();
}
}
```

Output:

```
Within Dispose() of One
Within Dispose() of Three
Within destructor of Four
Within destructor of Two
```

• Both Dispose() and Finalize() (or destructor) methods are used to release the unmanaged resources.

• As we can see in the above example, when Dispose() method is invoked through an object, we can prevent the CLR from calling the corresponding destructor with the help of SuppressFinalize() method of GC class.

• By manually calling Dispose() method, we are releasing the resources and hence there is no need to call finalizer.

• Calling the Dispose() function manually is termed as explicit object de-allocation and making use of finalizer is known as implicit object de-allocation.

Forcing Garbage Collection:

- We know that, CLR will automatically trigger a garbage collection when a managed heap is full.
- We, the programmers, will not be knowing, when this process will happen.

- However, if we wish, we can force the garbage collection to occur using the following statements: GC.Collect(); GC.WaitForPendingFinalizers();
- The method WaitForPendingFinalizers() will allow all finalizable objects to perform any necessary cleanup before getting destroyed. Though, we can force garbage collection to occur, it is not a good programming practice.

Programmatically Interacting with Generations

- We can investigate the generation of an object currently belongs to using GC.GetGeneration().
- GC.Collect() allows to specify which generation should be checked for valid application roots.
- Consider the following code:

```
class Car : IDisposable
{
string name;
public Car(string n)
{
name = n;
}
~Car()
{
Console.WriteLine("Within destructor of {0}", name);
}
public void Dispose()
{
Console.WriteLine("Within Dispose() of {0}", name);
GC.SuppressFinalize(this);
}
}
class Test
{
public static void Main()
{
```

```
            Car c1 = new Car("One");
            Car c2 = new Car("Two");
            Car c3 = new Car("Three");
            Car c4 = new Car("Four");
            Console.WriteLine("c1 is Gen {0}", GC.GetGeneration(c1));
            Console.WriteLine("c2 is Gen {0}", GC.GetGeneration(c2));
            Console.WriteLine("c3 is Gen {0}", GC.GetGeneration(c3));
            Console.WriteLine("c4 is Gen {0}", GC.GetGeneration(c4));
            c1.Dispose();
            c3.Dispose();
            GC.Collect(0);
            Console.WriteLine("c1 is Gen {0}", GC.GetGeneration(c1));
            Console.WriteLine("c2 is Gen {0}", GC.GetGeneration(c2));
            Console.WriteLine("c3 is Gen {0}", GC.GetGeneration(c3));
            Console.WriteLine("c4 is Gen {0}", GC.GetGeneration(c4));
        }
        }
```

Output: C1 is Gen 0

C2 is Gen 0

C3 is Gen 0

C4 is Gen 0

Within Dispose() of One

Within Dispose() of Three

C1 is Gen 1

C2 is Gen 1

C3 is Gen 1

C4 is Gen 1

Within Destructor of Four

Within Destructor of Two

Model Questions in this Chapter:

1. Explain the keywords: 1) finally, 2) using.
2. What is meant by object lifetime? Explain the Garbage Collection optimization process in C#. or What is meant by object life time? Describe the role of .NET garbage collection, finalization process and Ad-Hoc destruction method, with examples.
3. Explain the process of finalizing objects in .NET environment. Given the members of System.GC and explain their usage, with examples.
4. Explain the different methods of file System.GC type.
5. Explain the concept of finalizing types.
6. Explain in detail garbage collection optimization.
7. Explain how garbage collection(g.c) works?
8. How do destructors and garbage collection work in C#?
9. What do you mean by object life time? Explain the garbage collection optimization process in c#?
10. When does garbage collection occur. Explain the process of garbage collection and how it is optimized in .NET?
11. Describe the role of .NET garbage collection, finalization, and Ad Hoc Destruction method with examples
12. Explain the process of finalizing objects in .NET environment. Give the members of system. GC and explain their usage, with examples.
13. When do you override the virtual system.object.Finalize() method? How do you implement using destructor? Why do we avoid using destructor in C# applications?
14. Explain finalization process. How do you build ADHOC destruction method?