**C# Memory Management**

In this article, we will see how memory management is done in .NET environment. Let’s try to keep it simple and short so that people with different levels of knowledge and experience can be benefited. I am planning to write this in three parts in order to make it easy to read.

* Stack and heap
* Value types and reference types

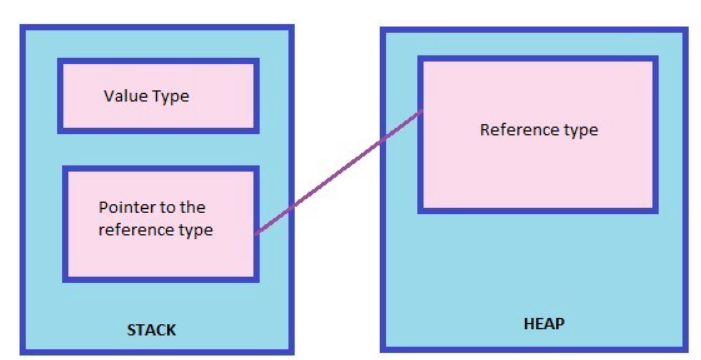
## **Stack and Heap**

Stack and heap are portions of the memory. The **Common Language Runtime (CLR)** allocates memory for objects in these parts.

**Stack** is a simple LIFO(last-in-first-out) structure. Variables allocated on the stack are stored directly to the memory and access to this memory is very fast, and its allocation is done when the program is compiled. When a method is invoked, the CLR bookmarks the top of the stack. The method then pushes data onto the stack as it executes. When the method completes, the CLR resets the stack to its previous bookmark, popping all the method’s memory allocations is one simple operation.

**Heap** can be viewed as a random jumble of objects. It allows objects to be allocated or deallocated in a random order. Variables allocated on the heap have their memory allocated at run time and accessing this memory is a bit slower, but the heap size is only limited by the size of virtual memory. The heap requires the overhead of a garbage collector to keep things in order.

Value type variables are stored in the stack and reference type variables are stored in the heap.



## **Value Type and Reference Type**

A **value type** holds the data within its own memory location.

Value types => bool, byte, char, decimal, double, float, int, long, uint, ulong, ushort, enum, struct

A **reference type** contains a pointer to another memory location that holds the real data.

Reference types => class, interface, delegate, string, object, dynamic, arrays

If you assign a value type variable to another variable, the value is copied directly.

Here is an example for a value type variable copy:

**int x=3; int y=x; x=5;**

If you assign a reference type variable to another, as reference type variables represent the address of the variable, the reference is copied and both variables point to the same location of the heap.

Person obj=new Person();

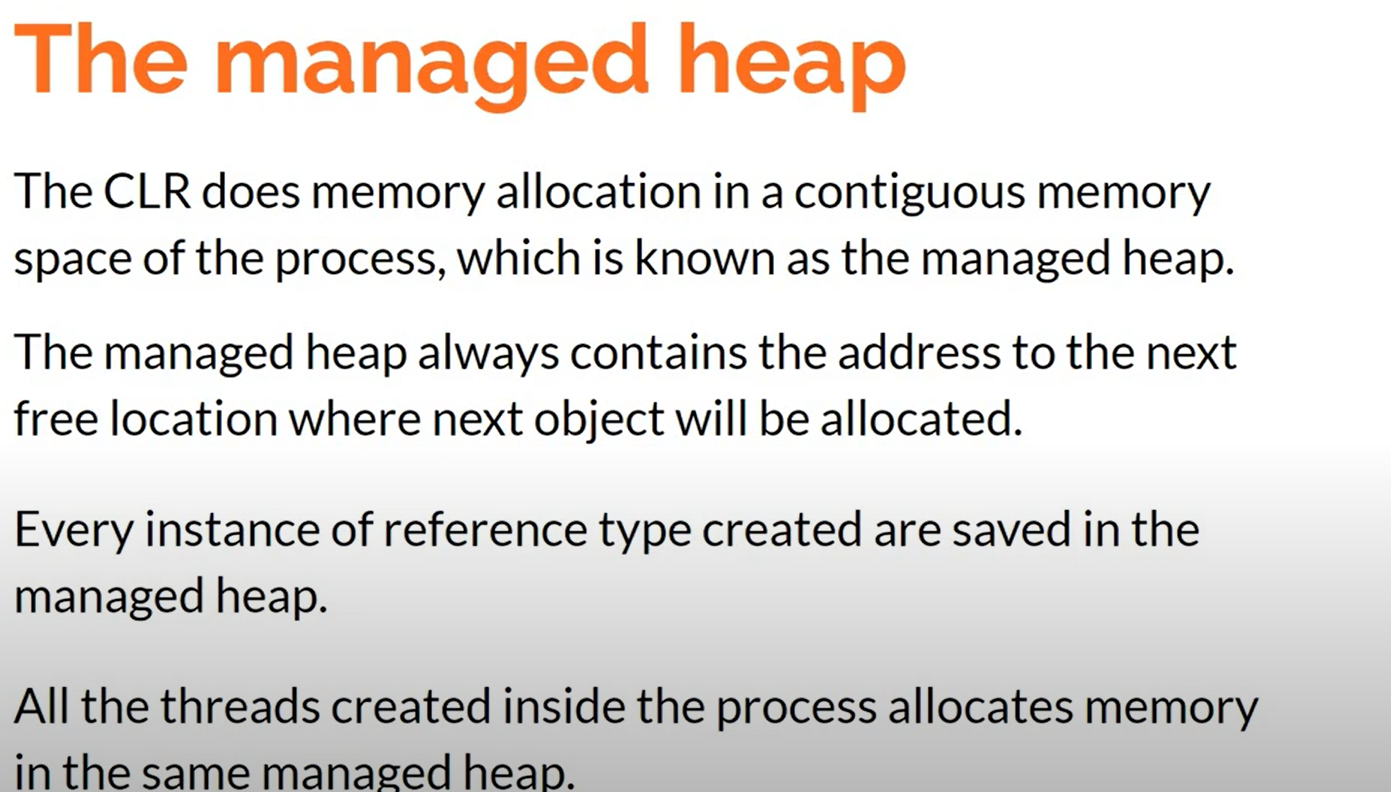
Person obj1=obj;

# Garbage Collection in C# | .NET Framework

Automatic memory management is made possible by **Garbage Collection in .NET Framework**. When a class object is created at runtime, certain memory space is allocated to it in the heap memory. However, after all the actions related to the object are completed in the program, the memory space allocated to it is a waste as it cannot be used. In this case, garbage collection is very useful as it automatically releases the memory space after it is no longer required.   
Garbage collection will always work on **Managed Heap** and internally it has an Engine which is known as the **Optimization Engine**.

Garbage Collection occurs if at least one of multiple conditions is satisfied. These conditions are given as follows:

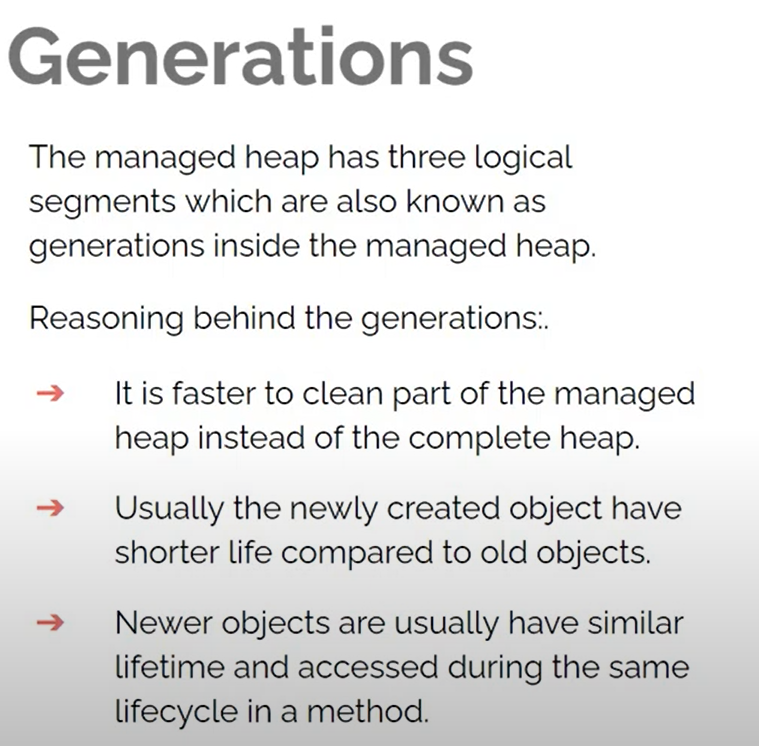
* If the system has low physical memory, then garbage collection is necessary.
* If the memory allocated to various objects in the heap memory exceeds a pre-set threshold, then garbage collection occurs.
* If the *GC.Collect* method is called, then garbage collection occurs. However, this method is only called under unusual situations as normally garbage collector runs automatically.

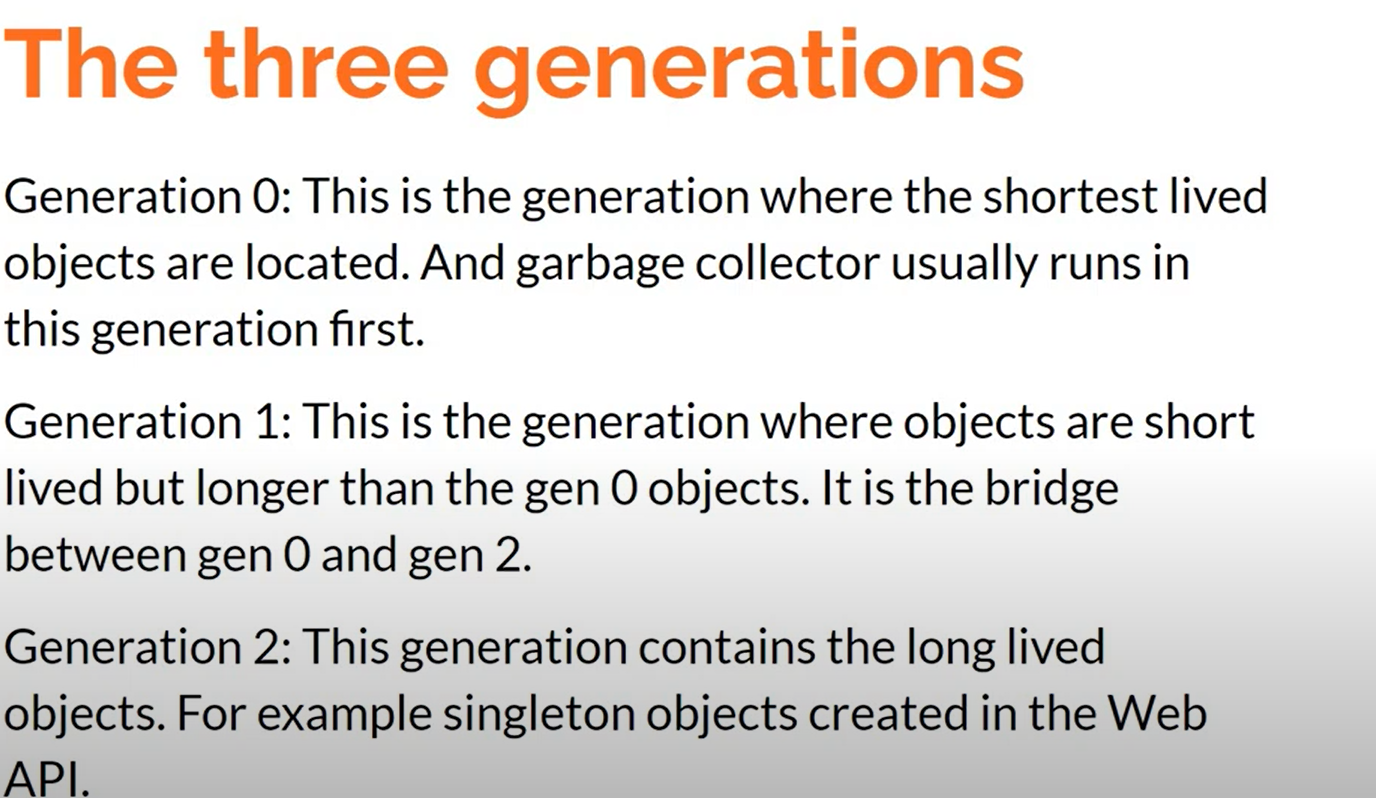


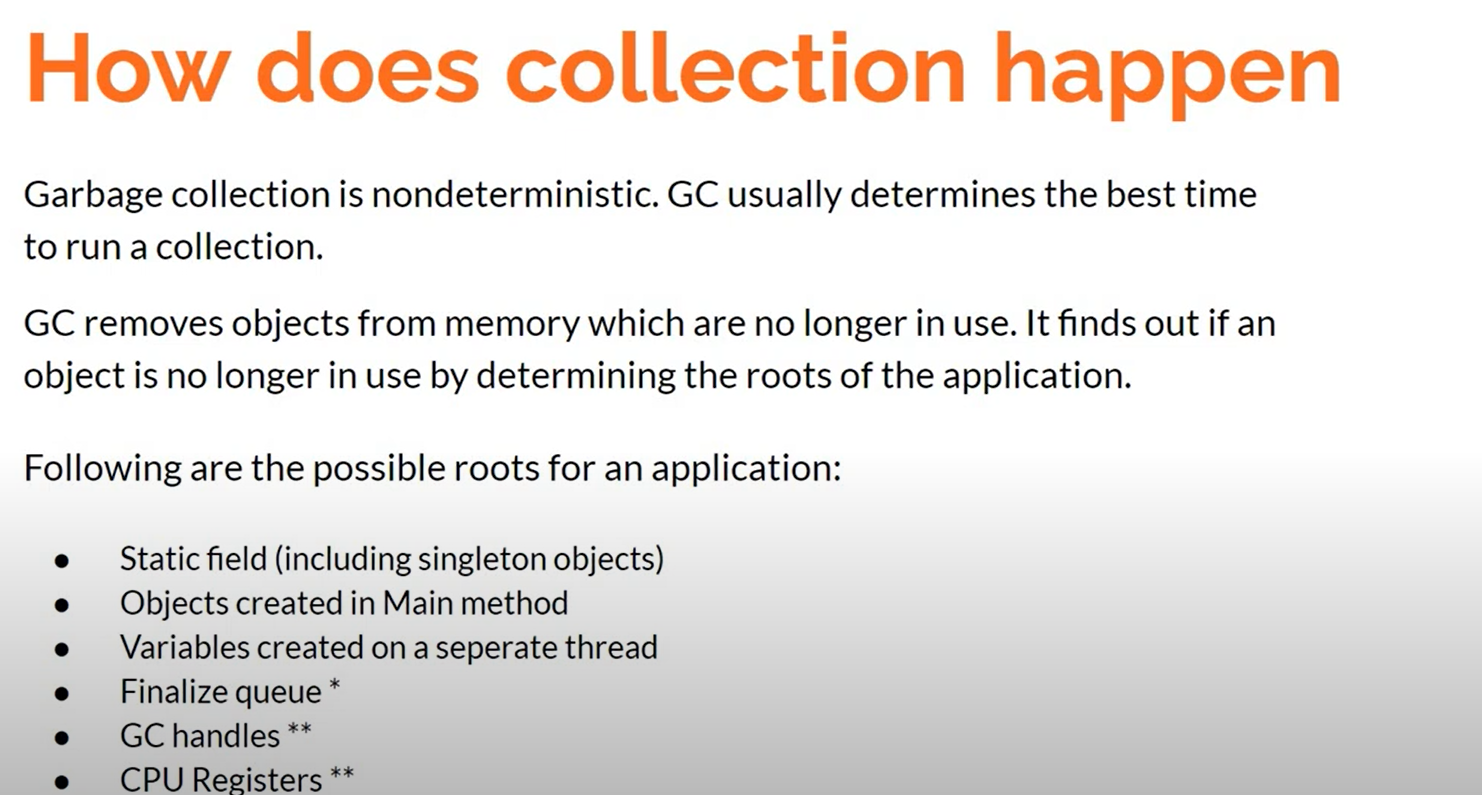
### **How GC works in C#**

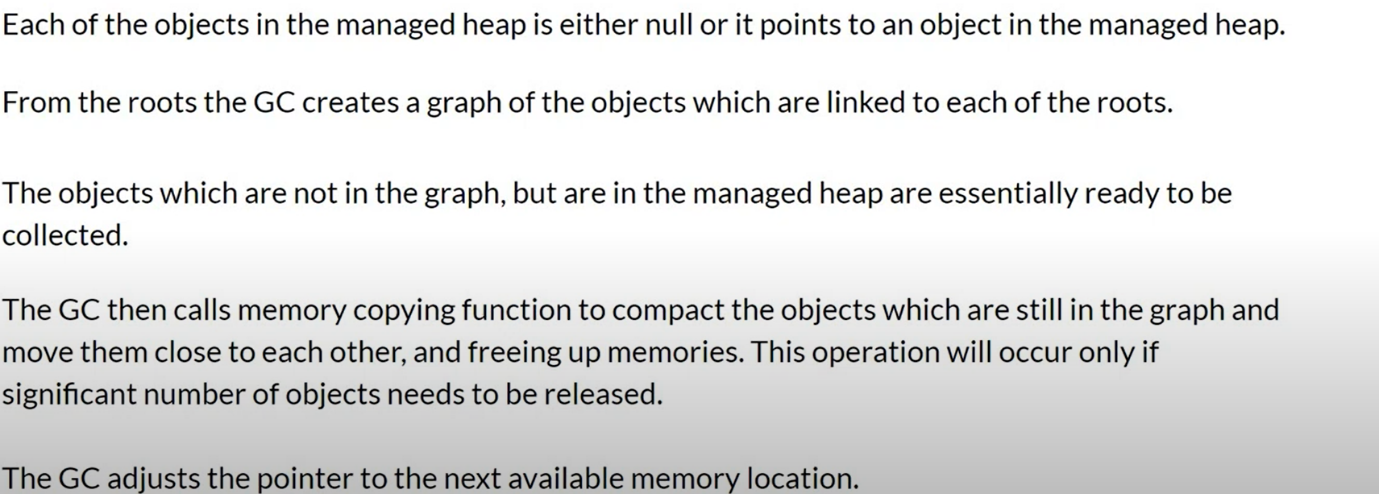
Memory is divided into spaces called generations. The collector starts claiming objects in the youngest generation. Then it promotes the survivors to the next generation. The C# garbage collection uses three generations in total:

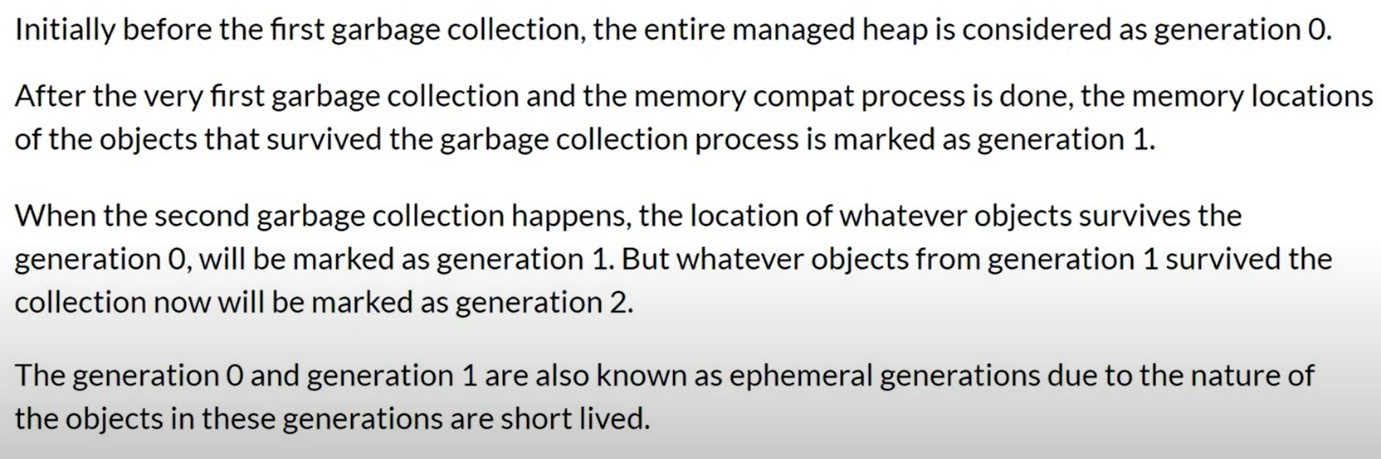
* Generation 0—This generation holds short-lived objects. Here’s where the collection process happens most often. When you instantiate a new object, it goes in this generation by default. The exceptions are objects whose sizes are equal to or greater than 85,000 bytes. Those objects are large, so they go straight into generation 2.
* Generation 1—This is an intermediate space between the short-lived and long-lived layers.
* Generation 2—Finally, this is the generation that holds objects that live the longest in the application—sometimes as long as the whole duration of the app. GC takes place here less frequently.

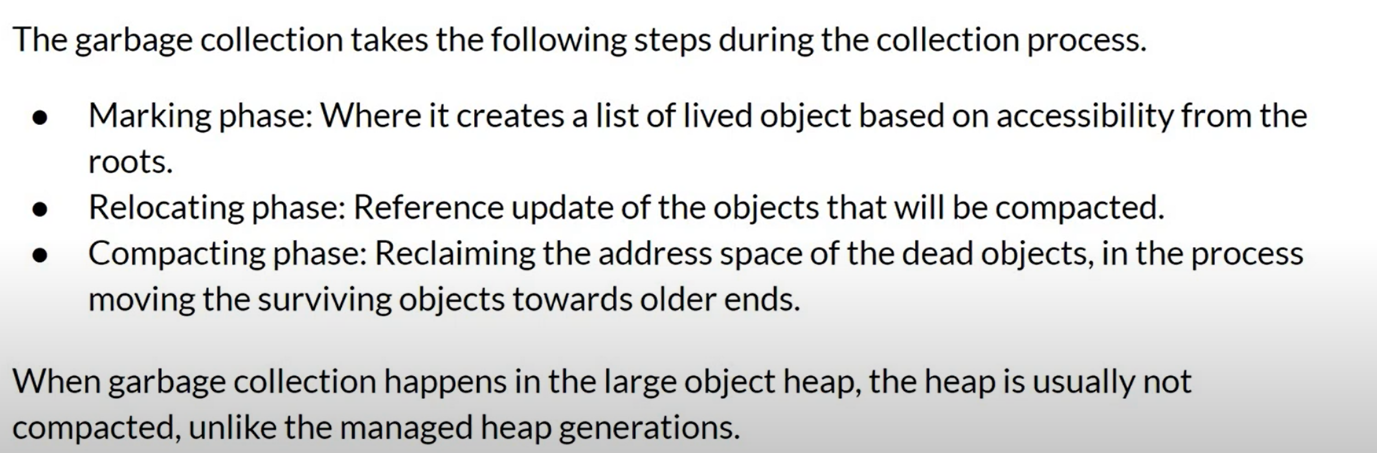












# Finalizers

Finalizers are used to perform any necessary final clean-up when a class instance is being collected by the garbage collector.

* Finalizers cannot be defined in structs. They are only used with classes.
* A class can only have one finalizer.
* Finalizers cannot be inherited or overloaded.
* Finalizers cannot be called. They are invoked automatically.
* A finalizer does not take modifiers or have parameters.

For example, the following is a declaration of a finalizer for the Car class

class Car

{

~Car() // finalizer

{

// cleanup statements...

}

}

Example

class IntWrapper{

public int x;

}

Void main(){

var obj1=new IntWrapper();

.. // create two more objects of same type

obj1.x=1; obj2.x=2; obj3.x=3;

GC.collect();

Obj2=null;

GC.collect();

}

The GC.GetGeneration() method can be used to know the generation to which an object belongs.

List<string> obj = new List<string>() { "Nandini", "S" };

           Console.WriteLine(System.GC.GetGeneration(obj));

           System.GC.Collect();

           Console.WriteLine(System.GC.GetGeneration(obj));

           System.GC.Collect();

           Console.WriteLine(System.GC.GetGeneration(obj));

**Example.** First, this example invokes the GC.Collect method. Three calls to get the total memory usage on the system are present. They occur before the allocation, after the allocation, and after the forced garbage collection.

static void Main()

{

long mem1 = GC.GetTotalMemory(false);

{

*// Allocate an array and make it unreachable.*

int[] values = new int[50000];

values = null;

}

long mem2 = GC.GetTotalMemory(false);

{

*// Collect garbage.*

GC.Collect();

}

long mem3 = GC.GetTotalMemory(false);

{

Console.WriteLine(mem1);

Console.WriteLine(mem2);

Console.WriteLine(mem3);

}

}

}

Dispose and Finalize in .Net

Managed Resources

* Created and handled by CLR

Unmanaged Resources

* Not Controlled by CLR

By default Garbage collector disposes an object when it is not being referenced anywhere.

Dispose Unmanged resources as soon as we are done using them.

1. Calling Dispose Method
2. Creating a Using Block{}

Syntax

Finalize is created using the class destructor

Dispose is used by creating the Dispose().

Execution

Finalize is called in a non-deterministic way.

Dispose can be explicitly called to release resources.

Examples

**namespace** FinalizeDemo

{

**class** Program

    {

**static** **void** Main(**string**[] args)

        {

            FinalizeDemo d = **new** FinalizeDemo();

            d = **null**;

            Console.ReadLine();

        }

    }

**class** FinalizeDemo

    {

**public** FinalizeDemo()

        {

            Console.WriteLine("Object Created");

        }

        ~FinalizeDemo()

        {

            Console.WriteLine("Destructor Called.");

        }

    }

}

Dispose ()

**namespace** FinalizeDemo

{

**class** Program

    {

**static** **void** Main(**string**[] args)

        {

            FinalizeDemo d = **new** FinalizeDemo();

            d.Dispose();

            d = **null**;

            Console.ReadLine();

        }

    }

**class** FinalizeDemo:IDisposable

    {

**private** **bool** Disposed = **false**;

**public** FinalizeDemo()

        {

            Console.WriteLine("Object Created");

        }

        ~FinalizeDemo()

        {

            Console.WriteLine("Destructor Called.");

            Dispose(**false**);

        }

**public** **void** Dispose()

        {

            Dispose(**true**);

            GC.SuppressFinalize(**this**);

        }

**public** **void** Dispose(**bool** disposing)

        {

**if**(!Disposed)

            {

**if** (disposing)

                {

                    Console.WriteLine("Called From Dispose");

                   //Clear all the managed resources here

                }

**else** {

                    //Clear all the unmanaged resources here

                }

                Disposed = **true**;

        }

    }

}

Here, we have taken a Boolean variable Disposed = false. Now, we have two scenarios

1) If a user calls the Dispose method: Here, we are checking whether an object has been disposed of or not. Now, when we call this method from the Dispose method of IDisposable interface, we pass true. In if block, we will write all the cleanup code, and then outside it, we will set the Disposed variable to true.

2) If a user forgets to call the Dispose method: In this case Destructor, we will call the Dispose Method with false and control will go to else block inside the Dispose method. Here, we will write all the cleanup code.

.NET introduced the using block to take care of calling the Dispose method if a class is implementing the IDisposable interface. Hence, it’s a good practice to create an object within the using block.