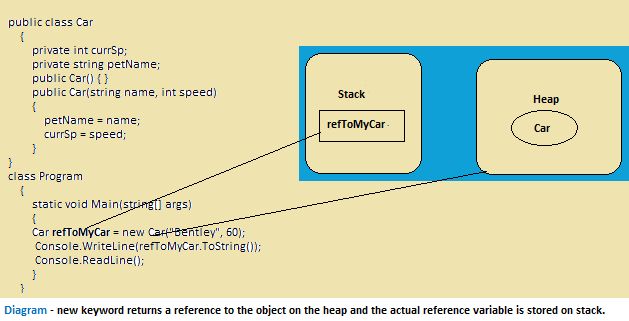
[](https://www.c-sharpcorner.com/technologies/dot_net_2015)

Object Lifetime in .NET Framework

In this article you will learn about object lifetime in .NET framework.

[Download Free .NET & JAVA Files API](https://www.c-sharpcorner.com/AdRedirector.aspx?AdId=133&target=https://www.e-iceblue.com/download/free-products.html?aff_id=100" \t "_blank)

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. Once the object is allocated with memory, it is necessary to release that memory so that it is used for further processing, otherwise it would result in memory leaks. We have a class in .Net that releases memory automatically for us when the object is no longer used. We will try to understand the entire scenario thoroughly of how objects are created and allocated memory and then deallocated when the object is out of scope.  
  
The class is a Blueprint that describes how an instance of this type will look and feel in memory. This instance is the object of that class type. A block of memory is allocated when the **new** keyword is used to instantiate the new object and constructor is called. This block of memory is big enough to hold the object. When we declare a class variable it is allocated on the stack and the time it hits a new keyword and then it is allocated on the **heap**. In other words, when an object of a class is created it is allocated on the heap with the C# **new** keyword operator. However a new keyword returns a reference to the object on the heap, not the actual object itself. This reference variable is stored on the stack for further use in applications.  
  
  
  
When the new operator is used to create an object, memory is taken from the managed heap for this object and the managed heap is more than just a random chunk of memory accessed by the CLR. When the object is no longer used then it is de-allocated from the memory so that this memory can be reused.  
  
The key pillar of the .NET Framework is the automatic garbage collection that manages memory for all .NET applications. When an object is instantiated the garbage collector will destroy the object when it is no longer needed. There is no explicit memory deal location since the garbage collector monitors unused objects and does a collection to free up memory that is an automatic process. The Garbage Collector removes objects from the heap when they are unreachable by any part of your code base. The .Net garbage collector will compact empty blocks of memory for the purpose of optimization.  
  
The heap is categorized into three generations so it can handle long-lived and short-lived objects. Garbage collection primarily occurs with the reclamation of short-lived objects that typically occupy only a small part of the heap.  
  
**Generations**  
There are the following three generations of objects on the heap:

* **Generation 0:** Newly created objects are at Generation 0. These objects on Generation 0 are collected frequently to ensure that short-lived objects are quickly collected and the memory is released. Objects that survive Generation 0, the collections are promoted to Generation 1. Most objects are reclaimed for garbage collection in Generation 0 and do not survive to the next generation.
* **Generation 1:** Objects that are collected less frequently than Generation 0 and contains longer-lived objects that were promoted from Generation 0. Objects that survive Generation 1, collection are promoted to Generation 2.
* **Generation 2:** Objects promoted from Generation 1 that are the longest-lived objects and collected infrequently. The overall strategy of the garbage collector is to collect and move longer-lived objects less frequently.

The garbage collector cleans up **managed resources** automatically since managed code is directly targeted by the CLR. But when the object uses **unmanaged resources** like database connections or file manipulation, that needs to be released manually and this can be done by a finalize method.  
  
We use the destructor method using the (~) sign in our code to destroy the objects and this destructor is converted into a finalize method (check in the compiled code). This is known as a finalization process. If we are implementing Finalize(), we do not have control since when this method should be called the garbage collector takes care of this on its own. In the finalization process there are a two collection cycles to completely release the object's memory. During the first collection pass, the object is flagged for finalization. After the finalization occurs, the garbage collector can reclaim the object's memory and the memory is released.  
  
There is another method, Dispose(), that releases managed and unmanaged resources explicitly. This method is the single method in an IDisposable interface and can be used to release unmanaged resources manually.  
  
I have written another article on the IDisposable pattern in which we would have a clear picture of the difference between Finalize() and Dispose ().