CHAPTER 8

Managing an Object's Lifetime

IN THIS CHAPTER

- Understanding Memory Allocation
- Understanding the Garbage Collection
- Understanding the Finalize Method
- Understanding Dispose and the IDisposible Interface
- Object Resurrection
- Advanced Garbage Collection

Real life is often a great place to get programming examples. Think of life: Humans are born; they grow up; they live their lives; they do tons of things; and, at a certain point of life, they die. Managing objects in programming environments works similarly. You give life to an object by creating an instance; then you use it in your own application while it is effectively useful to the application. But there is a point at which you do not need an object anymore, so you need to destroy it to free up memory and other resources, bringing an object to "death." Understanding how object lifetime works in .NET programming is fundamental because it gives you the ability to write better code; code that can take advantage of system resources to consume resources the appropriate way or return unused resources to the system.

Understanding Memory Allocation

Chapter 4, "Data Types and Expressions," discusses value types and reference types, describing how both of them are allocated in memory: Value types reside in the stack whereas reference types are allocated on the managed heap. When you create a new instance of a reference type, via the `New` keyword, the .NET Framework reserves some memory in the managed heap for the new object instance. Understanding memory allocation is fundamental, but an important practice is to also release objects and resources when they are unused or unnecessary. This returns free memory and provides better performances. For value types the problem is of easy resolution: Being allocated on the stack, they are simply removed from memory when you

assign the default zero value. The real problem is about reference types. Basically you need to destroy the instance of an object; to accomplish this you assign `Nothing` to the instance of a reference type.

When you perform this operation, the .NET Framework marks the object reference as no longer used and marks the memory used by the object as unavailable to the application, but it actually does not immediately free up the heap, and the runtime cannot immediately reuse such memory. Memory previously used by no-longer-referenced objects can be released by the .NET Framework after some time because the .NET Framework knows when it is the best moment for releasing resources. Such a mechanism is complex, but fortunately it is the job of the *garbage collector*.

Understanding Garbage Collection

In your applications you often create object instances or allocate memory for resources. When you perform these operations, .NET Framework checks for available memory in the heap. If available memory is not enough, .NET Framework launches a mechanism known as *garbage collection*, powered by an internal tool named *garbage collector*. The garbage collector can also be controlled by invoking members of the `System.GC` class, but the advantage is leaving.NET Framework the job of handling the process automatically for you. Basically the garbage collector first checks for all objects that have references from your applications, including objects referenced from other objects, enabling to keep alive object graphs. Objects having any references are considered as used and alive, so the garbage collector marks them as in use and therefore will not clean them up. With that said, any other objects in the heap are surely considered as unused and therefore the garbage collector removes the object and references to the object. After this, it compresses the heap and returns free memory space that can be reallocated for other new objects or resources. In all this sequence of operations, you do nothing. The garbage collector takes care of anything required. You can also decide to release objects when you do not need them any more setting their reference to `Nothing` so that you can free up some memory. The following snippet shows how you can logically destroy an instance of the `Person` class:

```
Dim p As New Person
p.FirstName = "Alessandro"
p.LastName = "Del Sole"

p = Nothing
```

When you assign an object reference with `Nothing`, the CLR automatically invokes the destructor (the `Finalize` method is covered in next section) that any class exposes because the most basic implementation is provided by `System.Object`. At this point there is a problem. The garbage collection behavior is known as *nondeterministic*, meaning that no one can predict the moment when the garbage collector is invoked. In other words, after you set an object reference to `Nothing`, you cannot know when the object will be effectively released. There can be a small delay, such as seconds, but also a long delay, such as minutes or hours. This can depend on several factors; for example, if no additional

memory is required during your application's lifetime, an object could be released at the application shut down. This can be a problem of limited importance if you have only in-memory objects that do not access to external resources, such as files or the network. You do not have to worry about free memory because, when required, the garbage collector will kick in. Anyway, you can force a garbage collection process by invoking the `System.GC.Collect` method. The following is an example:

```
p = Nothing
'Forces the garbage collector
'so that the object is effectively
'cleaned up
System.GC.Collect()
```

Forcing the garbage collection process is not a good idea. As you can easily imagine, frequently invoking a mechanism of this type can cause performance overhead and significantly slow down your application performances. (Although .NET 4.0 introduces some improvements that are covered in last section of this chapter.) When you work with in-memory objects that do not access external resources, such as the `Person` class, leave the .NET Framework the job of performing a garbage collection only when required. The real problem is when you have objects accessing to external resources, such as files, databases, and network connections that you want to be free as soon as possible when you set your object to `Nothing` and therefore you cannot wait for the garbage collection process to kick in. In this particular scenario you can take advantage of two methods that have little different behaviors: `Finalize` and `Dispose`.

OUT OF SCOPE OBJECTS

Objects that go out of scope will also be marked for garbage collection if they have no external reference to them, even if you don't set them explicitly to `Nothing`.

Understanding the `Finalize` Method

The `Finalize` method can be considered as a destructor that executes code just before an object is effectively destroyed, that is when memory should be effectively released. This method is inherited from `System.Object`; therefore any class can implement the method taking care of declaring it as `Protected Overrides` as follows:

```
Protected Overrides Sub Finalize()
    'Write your code here for releasing
    'such as closing db connections,
    'closing network connections,
    'and other resources that VB cannot understand

    MyBase.Finalize() 'this is just the base implementation
End Sub
```

If you need to destroy an object that simply uses memory, do not invoke `Finalize`. You need instead to invoke it when your object has a reference to something that Visual Basic cannot understand because the garbage collector does not know how to release that reference, so you need to instruct it by providing code for explicitly releasing resources. Another situation is when you have references to something that is out of the scope of the object, such as unmanaged resources, network connections, and files references different than .NET streams. If an object explicitly provides a `Finalize` implementation, the CLR *automatically* invokes such a method just before removing the object from the heap. This means that you *do not* need to invoke it manually. Notice that `Finalize` is not invoked immediately when you assign `Nothing` to the object instance you want to remove. Although `Finalize` enables you to control how resources must be released, it does not enable you to control when they are effectively released. This is due to the nondeterministic behavior of the garbage collector that frees up resources in the most appropriate moment, meaning that minutes or hours can be spent between the `Finalize` invocations and when resources are effectively released, although invoking `Finalize` marks the object as no longer available. Of course you could force the garbage collection process and wait for all finalizers to be completed, if you want to ensure that objects are logically and physically destroyed. Although this is not a good practice, because manually forcing a garbage collection causes performance overhead and loss of .NET Framework optimizations, you can write the following code:

```
'Object here is just for demo purposes
Dim c As New Object
c = Nothing
GC.Collect()
GC.WaitForPendingFinalizers()
```

There are also a few considerations on what `Finalize` should contain within its body. Here is a simple list:

- Do not throw exceptions within `Finalize` because the application cannot handle them and therefore it would crash.
- Invoke only shared methods except when the application is closing; this will avoid invocations on instance members from objects that can be logically destroyed.
- Continuing the previous point, do not access external objects from within `Finalize`.

To complete the discussion, it is worth mentioning that destruction of objects that explicitly provide `Finalize` require more than one garbage collection process. The reason why destroyed objects are removed from the heap at least at the second garbage collection is that `Finalize` could contain code that assigns the current object to a variable, keeping a reference still alive during the first garbage collection. This is known as *object resurrection* and is discussed later in this chapter. As a consequence, implementing `Finalize` can negatively impact performances and should be used only when strictly required.

Understanding Dispose and the IDisposable Interface

One of the issues of the Finalize destructor is that you cannot determine whether resources will be freed up and when an object will be physically destroyed from memory. This is because of the nondeterministic nature of the garbage collector, meaning that unused references will still remain in memory until the GC kicks in, and generally this is not a good idea. It is instead a good approach providing clients the ability to immediately release resources (such as network connections, data connections, or system resources) just before the object is destroyed setting it to Nothing. The .NET Framework provides a way for releasing resources immediately and under your control, which is the Dispose method. Implementing Dispose avoids the need of waiting for the next garbage collection enabling cleaning up resources immediately, right before the object is destroyed. Differently from Finalize, Dispose must be invoked manually. To provide a Dispose implementation, your class must implement the IDisposable interface. Visual Studio 2010 provides a skeleton of IDisposable implementation when you add the Implements directive. The code in Listing 8.1 shows the implementation.

LISTING 8.1 Implementing the IDisposable Interface

```
Class DoSomething
    Implements IDisposable

#Region "IDisposable Support"
    Private disposedValue As Boolean ' To detect redundant calls

    ' IDisposable
    Protected Overridable Sub Dispose(ByVal disposing As Boolean)
        If Not Me.disposedValue Then
            If disposing Then
                ' TODO: dispose managed state (managed objects).
            End If

            ' TODO: free unmanaged resources (unmanaged objects) and
            ' override Finalize() below.
            ' TODO: set large fields to null.
        End If
        Me.disposedValue = True
    End Sub

    ' TODO: override Finalize() only if Dispose(ByVal disposing As Boolean) above
    'has code to free unmanaged resources.
    'Protected Overrides Sub Finalize()
```

```
    'Do not change this code.  Put cleanup
    'code in Dispose(ByVal disposing As Boolean) above.
    '    Dispose(False)
    '    MyBase.Finalize()
    'End Sub

    ' This code added by Visual Basic to correctly implement the disposable pattern.
    Public Sub Dispose() Implements IDisposable.Dispose
        ' Do not change this code.  Put cleanup
        'code in Dispose(ByVal disposing As Boolean) above.
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub
#End Region
End Class
```

Notice how `Dispose` is declared as `Overridable` so that you can provide different implementations in derived classes. Visual Studio is polite enough to provide comments showing you the right places for writing code that release managed or unmanaged resources. Also notice how there is an implementation of `Finalize` that is enclosed in comments and therefore is inactive. Such a destructor should be provided only if you have to release unmanaged resources. You invoke the `Dispose` method before setting your object reference to nothing, as demonstrated here:

```
Dim dp As New DoSomething
'Do your work here...

dp.Dispose()
dp = Nothing
```

As an alternative you can take advantage of the `Using..End Using` statement covered in next subsection. When implementing the `Dispose` pattern, in custom classes you need to remember to invoke the `Dispose` method of objects they use within their body so that they can correctly free up resources. Another important thing to take care of is checking if an object has already been disposed when `Dispose` is invoked but the auto-generated code for the `Dispose` method already keeps track of this for you.

DISPOSE AND INHERITANCE

When you define a class deriving from another class that implements `IDisposable`, you do not need to override `Dispose` unless you need to release additional resources in the derived class.

Using..End Using Statement

As an alternative to directly invoking Dispose, you can take advantage of the Using..End Using statement. This code block automatically releases and removes from memory the object that it points to, invoking Dispose behind the scenes for you. The following code example shows how you can open a stream for writing a file ensuring that the stream will be released even if you do not explicitly close it:

```
Using dp As New IO.StreamWriter("C:\TestFile.txt", False)
    dp.WriteLine("This is a demo text")
End Using
```

Notice how you simply create an instance of the object via the Using keyword. The End Using statement causes Dispose to be invoked on the previously mentioned instance. The advantage of Using..End Using is also that the resource is automatically released in cases of unhandled exceptions, and this can be useful.

Putting Dispose and Finalize Together

Implementing Dispose and Finalize cannot necessarily be required. It depends only on what kind of work your objects perform. Table 8.1 summarizes what and when you should implement.

TABLE 8.1 Implementing Destructors

What	When
No destructor	Objects that just work in memory and that reference other .NET in memory objects.
Finalize	Executing some code before the object gets finalized. The limitation is that you cannot predict when the GC comes in.
Dispose	Your objects access external resources that you need to free up as soon as possible when destroying the object.
Finalize and Dispose	Your objects access unmanaged resources that you need to free up as soon as possible when destroying the object.

You already have examples about Finalize and Dispose, so here you get an example of their combination. Before you see the code, you have to know that you will see invocations to Win32 unmanaged APIs that you do not need in real applications but these kinds of functions are useful to understand to know how to release unmanaged resources. Now take a look at Listing 8.2.

LISTING 8.2 Implementing Dispose and Finalize

```
Imports System.Runtime.InteropServices


Public Class ProperCleanup
    Implements IDisposable

    Private disposedValue As Boolean ' To detect redundant calls

    'A managed resource
    Private managedStream As IO.MemoryStream

    'Unmanaged resources
    <DllImport("winspool.drv")>
    Shared Function OpenPrinter(ByVal deviceName As String,
                                ByVal deviceHandle As Integer,
                                ByVal printerDefault As Object) _
                                As Integer
    End Function
    <DllImport("winspool.drv")>
    Shared Function _
    ClosePrinter(ByVal deviceHandle As Integer) _
                 As Integer
    End Function

    Private printerHandle As Integer

    'Initializes managed and unmanaged resources
    Public Sub New()
        managedStream = New IO.MemoryStream
        OpenPrinter("MyDevice", printerHandle, &H0)
    End Sub

    'Just a sample method that does nothing
    'particular except for checking if the object
    'has been already disposed
    Public Function FormatString(ByVal myString As String) As String
        If disposedValue = True Then
            Throw New ObjectDisposedException("ProperCleanup")
        Else
            Return "You entered: " & myString
        End If
    End Function
```

```
    ' IDisposable
    Protected Overridable Sub Dispose(ByVal disposing As Boolean)
        If Not Me.disposedValue Then
            If disposing Then
                ' TODO: dispose managed state (managed objects).
                managedStream.Dispose()
            End If

            ' TODO: free unmanaged resources (unmanaged objects)
            ' and override Finalize() below.
            ' TODO: set large fields to null.
            ClosePrinter(printerHandle)
        End If
        Me.disposedValue = True
    End Sub

    ' TODO: override Finalize() only if Dispose(ByVal disposing As Boolean)
    ' above has code to free unmanaged resources.
    Protected Overrides Sub Finalize()
        ' Do not change this code.  Put cleanup code in
        ' Dispose(ByVal disposing As Boolean) above.
        Dispose(False)
        MyBase.Finalize()
    End Sub

    ' This code added by Visual Basic to correctly implement the disposable pattern.
    Public Sub Dispose() Implements IDisposable.Dispose
        ' Do not change this code.  Put cleanup code
        ' in Dispose(ByVal disposing As Boolean) above.
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub
End Class
```

The code in Listing 8.2 has some interesting points. First, notice how both managed resources (a `System.IO.MemoryStream`) and unmanaged resources (`OpenPrinter` and `ClosePrinter` API functions) are declared. Second, notice how the constructor creates instances of the above resources. Because there are unmanaged resources, it is necessary to override the `Finalize` method. Visual Basic is polite enough to show you comments describing this necessity, so you simply uncomment the `Finalize` block definition. Such method invoke the `Dispose` one passing `False` as an argument; this can ensure that `Dispose` will clean up unmanaged resources as you can understand examining the conditional code block within the method overload that accepts a Boolean argument. Finally, notice how the other `Dispose` overload, the one accepting no arguments, invokes the

other overload passing `True` (therefore requiring managed resources to be released), and then it invokes the `GC.SuppressFinalize` method to ensure that `Finalize` is not invoked. There are no unmanaged resources to release at this point because `Finalize` was previously invoked to clean unmanaged resources.

Object Resurrection

With the *object resurrection* phrase, we describe the scenario in which an object is restored after its reference was removed, although the object was not removed yet from memory. This is an advanced technique, but it is not very useful and Microsoft strongly discourages you from using it in your applications. It is helpful to understand something more about objects' lifetime. Basically an object being finalized can store a self-reference to a global variable, and this can keep the object alive. In simpler words, a reference to a "died" object is restored when within the `Finalize` method the current object is assigned (using the `Me` keyword) to a class level or module level variable. Here's a small code example for demonstrating object resurrection; keep in mind that the code is simple to focus on the concept more than on the code difficulty, but you can use this technique with more and more complex objects. You add this code to a module:

```
Public resurrected As ResurrectionDemo


Sub TestResurrection()
    Dim r As New ResurrectionDemo
    'This will invoke Finalize
    r = Nothing

End Sub
```

The resurrected variable is of type `ResurrectionDemo`, a class that will be implemented next. This variable holds the actual reference to the finalizing object so that it can keep it alive. The `TestResurrection` method creates an instance of the class and sets it to `Nothing` causing the CLR to invoke `Finalize`. Now notice the implementation of the `ResurrectionDemo` class and specifically the `Finalize` implementation:

```
Class ResurrectionDemo


    Protected Overrides Sub Finalize()
        'The object is resurrected here
        resurrected = Me
        GC.ReRegisterForFinalize(Me)
    End Sub
End Class
```

Notice how `Finalize`'s body assigns the current object to the `resurrected` variable which holds the reference. When an object is resurrected, `Finalize` cannot be invoked a second

time because the garbage collector removed the object from the finalization queue. This is the reason why the `GC.ReRegisterForFinalize` method is invoked. As a consequence, multiple garbage collections are required for a resurrected object to be cleaned up. At this point, just think of how many system resources this might require. Moreover, when an object is resurrected, previously referenced objects are also resurrected. This can result in application faults because you cannot know if objects' finalization already occurred. As mentioned at the beginning of this section, the object resurrection technique rarely takes place in real-life application because of its implications and generally can be successfully used only in scenarios in which you need to create pools of objects whose frequent creation and destruction could be time-consuming.

Advanced Garbage Collection

The garbage collection is a complex mechanism, and in most cases you do not need to interact with the garbage collector because, in such cases, you must be extremely sure that what you are doing is correct. The .NET Framework automatically takes care of what the CLR needs. Understanding advanced features of the garbage collector can provide a better view of objects' lifetime. The goal of this section is therefore to show such advanced features.

Interacting with the Garbage Collector

The `System.GC` class provides several methods for manually interacting with the garbage collector. In this chapter you already learned some of them. Remember `Collect` that enables forcing a garbage collection; `WaitForPendingFinalizers` enabling to wait for all `Finalize` methods to be completed before cleaning up resources; `ReRegisterForFinalize` that puts an object back to the finalization queue in object resurrection; and `SuppressFinalize` that is used in the `Dispose` pattern for avoiding unnecessary finalizations. There are other interesting members; for example, you can get an approximate amount of allocated memory as follows:

```
Dim bytes As Long = System.GC.GetTotalMemory(False)
```

You pass `False` if you do not want a garbage collection to be completed before returning the result. Another method is `KeepAlive` that adds a reference to the specified object preventing the garbage collector from destroying it:

```
GC.KeepAlive(anObject)
```

You can then tell the garbage collector that a huge amount of unmanaged memory should be considered within a garbage collection process; you accomplish this invoking the `AddMemoryPressure` method that requires the amount of memory as an argument. Next you can tell the garbage collection that an amount of unmanaged memory has been released invoking the `RemoveMemoryPressure` method. There are other interesting members enabling garbage collector interaction, covered in the next subsection for their relationship with the specific topic.

Understanding Generations and Operation Modes

The garbage collector is based on *generations* that are basically a counter representing how many times an object survived to the garbage collection. The .NET Framework supports three generations. The first one is named gen0 and is when the object is at its pure state. The second generation is named gen1 and is when the object survived to one garbage collection, whereas the last generation is named gen2 and is when the object survived to more than two garbage collections. This is a good mechanism for the garbage collector's performances because it first goes to remove objects at gen2 instead of searching for all live references. The garbage collection process is available in two modes: server and workstation. It is important to know this because in server mode the garbage collection runs on a single thread and therefore needs to block other threads while executing. In a workstation context, the garbage collection can be executed on multiple threads. This is known as concurrent GC. Until .NET Framework 3.5 SP 1, concurrent GC could perform garbage collections on both gen0 and gen1 concurrently or most of a gen2 without pausing managed code but never gen2 concurrently with the other ones. In .NET Framework 4.0 there is a new feature named *Background GC* that enables collecting all generations together, still limited to workstation mode, and that also enable allocating memory while collecting. The good news is that you can now take advantage of a feature introduced in .NET Framework 3.5, which enables registering from garbage collection events, for getting noticed about gen2 completion. The code in Listing 8.3 demonstrates this (read comments for explanations).

LISTING 8.3 Registering for garbage collection events

```
Sub Main()
    Try
        'Registers for notification about gen2 (1st arg) and
        'large objects on the heap (2nd arg)
        GC.RegisterForFullGCNotification(10, 10)

        'Notifications are handled via a separate thread
        Dim thWaitForFullGC As New Thread(New  _
                                  ThreadStart(AddressOf WaitForFullGCProc))
        thWaitForFullGC.Start()

    Catch ex As InvalidOperationException

        'Probably concurrent GC is enabled
        Console.WriteLine(ex.Message)
    End Try
End Sub

Public Shared Sub WaitForFullGCProc()
    While True
        'Notification status
```

```
        Dim s As GCNotificationStatus

        'Register for an event advising
        'that a GC is imminent
        s = GC.WaitForFullGCApproach()

        If s = GCNotificationStatus.Succeeded Then
            'A garbage collection is imminent

        End If

        'Register for an event advising
        'that a GC was completed
        s = GC.WaitForFullGCComplete()
        If s = GCNotificationStatus.Succeeded Then

            'A garbage collection is completed
        End If
    End While
End Sub
```

You can see how you can easily subscribe for garbage collection events invoking `WaitForFullGCComplete` and `WaitForFullApproach`.

Summary

Understanding how memory and resources are released after objects usage is fundamental in every development environment. In .NET this is accomplished by the garbage collector, a complex mechanism that comes after you set an object reference to `Nothing` or when you attempt to create new instances of objects but no more memory is available. You can also implement explicit destructors, such as `Finalize` or `Dispose`, according to specific scenarios in which you do need to release external or unmanaged resources before destroying an object. The garbage collection process has been improved in.NET Framework 4.0 so that it can support a new operation mode, known as Background GC that enables executing the process across multiple threads in every generation step.