

Object Resurrection

In Chapter 5, I briefly described the technique known as object resurrection, through which an object being finalized can store a reference to itself in a variable defined outside the class so that this new reference keeps the object alive. A minor problem with this technique is that by default, the garbage collector calls the `Finalize` method exactly once. The object being resurrected will eventually be set to `Nothing` or go out of scope again, but its `Finalize` method isn't going to be invoked again unless you call the `GC.ReRegisterForFinalize` method to ask the garbage collector to do so.

Object resurrection is an advanced technique that's likely to be useful only in unusual scenarios, such as when you're implementing a pool of objects whose creation and destruction is time-consuming. The companion source code for this book includes an application named `ObjectPool`, an example that demonstrates how resurrection can be used to implement an object pool (see Figure 1). In these pages, I'll merely sketch how the application works.

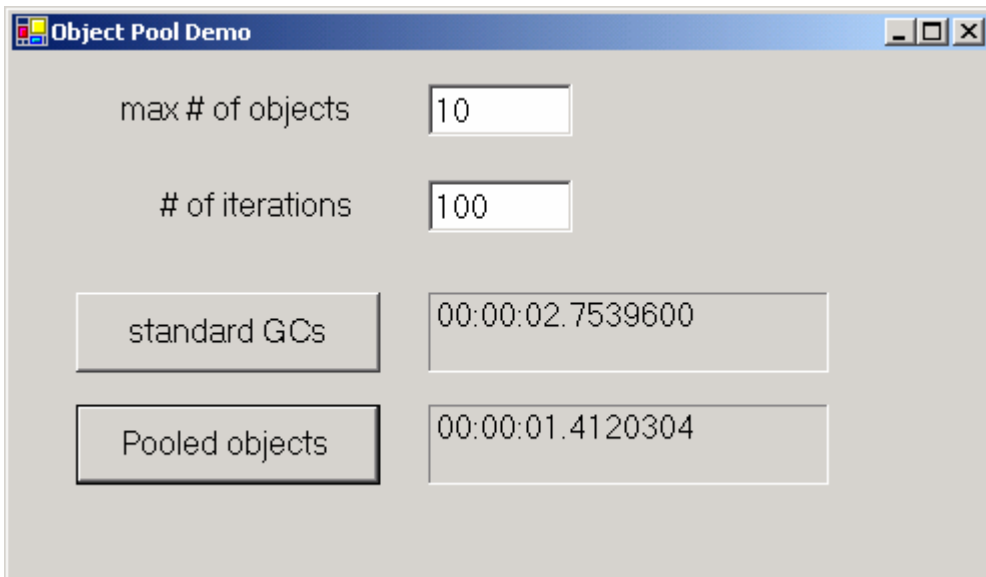


Figure 1

The ObjectPool demo application.

The `ObjectPool` demo application shows that an object pool manager can improve performance when many objects are frequently created and destroyed. Assume that you have a `RandomArray` class, which encapsulates an array of random numbers. The main program creates and destroys thousands of `RandomArray` objects, even though only a few objects are alive at a given moment. Because the class creates the random array in its constructor method (a time-consuming operation), this situation is ideal for a pooling technique:

```
Class RandomArray
    Public ReadOnly ArrRand(100000) As Double

    ' The constructor creates the random array.
    Sub New()
        Dim rand As New Random()

        ' A time-consuming operation
        For i As Integer = 0 To UBound(ArrRand)
            ArrRand(i) = rand.NextDouble
        Next
    End Sub
```

```
' The PoolManager that owns this instance
Public OwnerPoolManager As PoolManager
```

```
End Class
```

The program also contains a class named `PoolManager`, whose purpose is to provide the main program with a function that returns an initialized `RandomArray` object. This method takes a `RandomArray` object from an internal stack if possible; otherwise, it creates it using a `New` operator in the usual way.

```
Class PoolManager
    Implements IDisposable

    ' This stack contains all the objects in the pool.
    Public PooledObjects As New System.Collections.Stack()

    ' Return a new instance of the RandomArray class.
    Function NewRandomArray() As RandomArray
        If PooledObjects.Count > 1 Then
            ' If there is an object in the pool, use it.
            Return PooledObjects.Pop()
        Else
            ' Otherwise, create a new object.
            NewRandomArray = New RandomArray()
            ' Let it know that it's owned by this PoolManager object.
            NewRandomArray.OwnerPoolManager = Me
        End If
    End Function
End Class
```

```
End Class
```

I describe the `System.Collections.Stack` object in the section “The Stack Class” in Chapter 8. The main program creates a new instance of the `RandomArray` class using this syntax:

```
Dim PoolMan As New PoolManager
Dim ra As RandomArray = PoolMan.NewRandomArray()
```

The crucial point in the pooling technique is that the `PoolManager` class contains a reference to unused objects in the pool (in the `PooledObjects` Stack object), but not to objects being used by the main program. In fact, the latter objects are kept alive only by references in the main program. When the main program sets a `RandomArray` object to `Nothing` (or lets it go out of scope) and a garbage collection occurs, the garbage collector invokes the object’s `Finalize` method. The code inside the `RandomArray`’s `Finalize` method has therefore an occasion to resurrect itself by storing a reference to itself in the `PoolManager`’s `PooledObjects` structure. So when the `NewRandomArray` function is called again, the `PoolManager` object can return a pooled object to the client without going through the time-consuming process of creating a new one:

```
' ... (Inside the RandomArray class) ...

Protected Overrides Sub Finalize()
    If OwnerPoolManager Is Nothing Then
        ' This instance isn't owned by a pool manager.
    ElseIf OwnerPoolManager.PooledObjects Is Nothing Then
        ' This instance had an owner that was already finalized.
    Else
        ' Put this object back in the PooledObjects structure.
        OwnerPoolManager.PooledObjects.Push(Me)
        ' Reregister current object for the Finalize method.
        GC.ReRegisterForFinalize(Me)
    End If
End Sub
```

```
End If
End Sub
```

The most important statement in the preceding Finalize method is the GC.ReRegisterForFinalize call. Remember that by default, the garbage collector doesn't call the Finalize method again when an object resurrects itself; therefore, the object will be pooled only once. The call to ReRegisterForFinalize ensures that RandomArray objects go in and out of the pool as many times as necessary. (Just ensure that you don't call ReRegisterForFinalize more than once because that would cause the Finalize method to be called multiple times.)

The only missing part in the pooling schema is the Dispose method in the PoolManager class. The code in this method frees all the objects in the pool and sets the PooledObjects stack structure to Nothing so that the objects currently referenced by the main application can determine that they shouldn't resurrect themselves:

```
' ...(Inside the PoolManager class)...

Sub Dispose() Implements IDisposable.Dispose
' Suppress Finalize for all objects still in the pool.
For Each ra As RandomArray In PooledObjects
    ra.OwnerPoolManager = Nothing
    GC.SuppressFinalize(ra)
Next
' Let unpooled objects (that is, objects owned by the main program)
' know that they shouldn't put themselves back in the pool.
PooledObjects = Nothing
End Sub
```

The ObjectPool demo application lets you test and benchmark two similar pieces of code that create thousands of RandomArray objects, with or without the support of the PoolManager class. Depending on how many objects you create and on the speed of your CPU, the pooled version often runs from 1.5 to 2 times faster than the nonpooled version. Of course, this ratio heavily depends on the nature of the application that uses the PoolManager, the number of objects in use, and the number of times they are created and destroyed, so I suggest that you perform similar benchmarks before applying this technique in a real-world Visual Basic .NET program.