Use .NET Deterministic Finalization

Implement methods on your objects that release expensive resources without needing to wait for garbage collection.

by Juval Löwy

Technology Toolbox

- ✓ VB.NET
- ✓ C#
- ☐ SQL Server 2000
- ☐ ASP.NET
- ☐ XML
- ☐ VB6

Go Online!

Use these Locator+ codes at **www.visualstudiomagazine.com** to go directly to these related resources.

Download
VS0304DT Download the code for this article, which includes a C# version and a VB.NET version of Listing 1.

Discuss
VS0304DT_D Discuss this article in the C# forum.

Read More
VS0304DT_T Read this article online. It includes Listing 1.

VS0212FB_T "Master Memory Management" by Francesco Balena

VS0208MS_T "Take Charge of Garbage Collection" by Mauro Sant'Anna

VS0110CE_T C# Explorer, "Manage C# Objects," by Bill Wagner

Most coding defects in traditional component development that aren't business-logic–specific are due to memory-management and object-lifecycle issues—memory leaks, cyclic reference counts, failing to call Delete or Release(), accessing objects that are deallocated already, and so on. Writing impeccable code isn't impossible, but it takes years of experience, iron discipline, a mature development process, management's commitment to quality, and strict coding and development standards (such as code review and quality assurance). Most software projects today lack almost all these ingredients.

One of the .NET platform's goals is to simplify component development to cope with this reality and increase the resulting code's quality. .NET takes almost all the burden of managing memory allocation for objects and memory deallocation off your shoulders. Unlike COM, .NET doesn't use reference counting of objects. It has a sophisticated garbage-collection mechanism that detects when clients no longer use an object, then destroys the object. However, .NET's simplification of the object lifecycle can limit system scalability and throughput. I'll show several ways you can prevent your code from incurring this cost.

.NET keeps track of accessible paths to the object in your code. An object is *reachable* as long as at least one client has a reference to the object. .NET keeps reachable objects alive. It considers unreachable objects to be garbage, so destroying them does no harm. The garbage collector traverses the heap of existing objects periodically; it uses an intricate pointing schema to detect unreachable objects, then releases the memory .NET allocates to them (see Figure 1). As a result, clients don't need to delete objects explicitly or increment or decrement a reference count on the objects they create.

The object should implement a method called Finalize() if it has specific cleanup to do. However, you don't provide a Finalize() method in C#—you provide a destructor instead. The compiler converts the destructor definition to a Finalize() method, and also calls your base class's Finalize() method. For example, suppose you have this class definition:

```
public class MyClass
{
   public MyClass(){}
   ~MyClass(){}
}
```

The compiler actually generates this code:

```
public class MyClass
{
   public MyClass(){}
   protected virtual void Finalize()
   {
      try
      {
         //Your destructor code goes here
      }
      finally
      {
         base.Finalize(); //everybody has
                          //one, from Object
      }
   }
}
```

The garbage collector knows the object has a Finalize() method by reading the object's metadata, and it calls Finalize() just before it destroys the object.

If the object holds onto expensive resources such as files, graphics device interface (GDI) objects, communication ports, or database connections, it releases these resources only when the garbage collector calls Finalize(). This happens at an undetermined point in the future—usually when the process hosting your component is out of memory (this is called *nondeterministic finalization.*) In theory, the expensive resources the object holds might never be released, which would hamper system scalability and throughput severely. A few solutions to the problems nondeterministic finalization causes are available. They're called *deterministic finalization*, because they take place at a known, determined time.

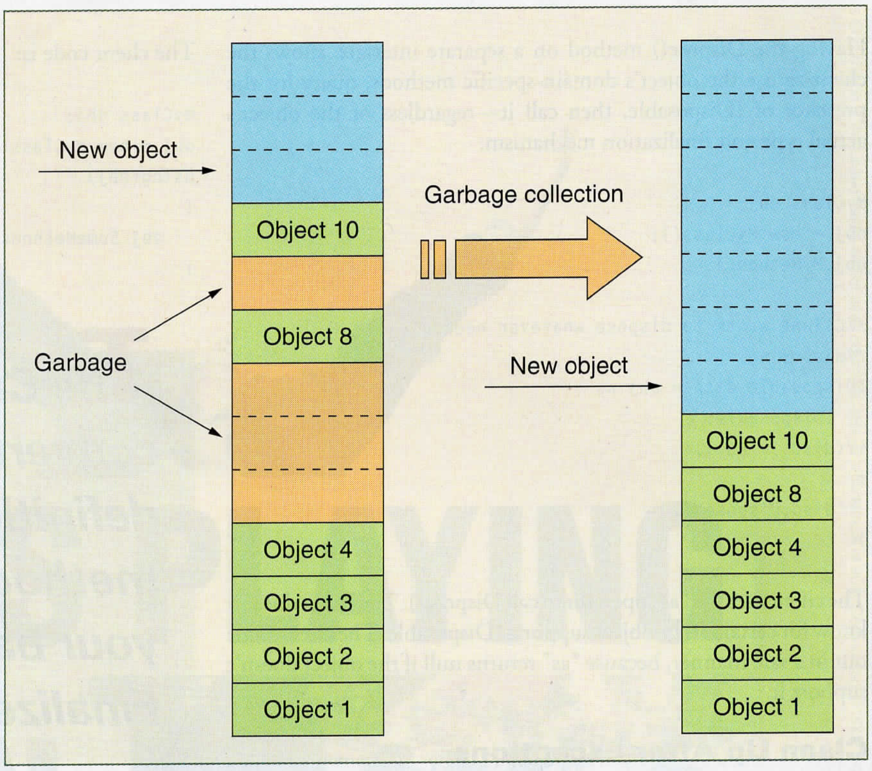

Figure 1 .NET Garbage Collection at Work. The garbage collector identifies objects on the managed heap that their clients no longer use. The garbage collector moves usable objects down and overrides the garbage, thereby reclaiming the memory they used.

Open and Close—or Dispose

The first solution is to implement methods on your object that allow the client to order explicit cleanup of expensive resources the object holds. Use this pattern when you can reallocate the resources the object holds onto. The object should expose methods such as Open() and Close(). An object that encapsulates a file is a good example. The client calls Close() on the object, allowing the object to release the file. If the client wants to access the file again, it calls Open(), without re-creating the object. Many classes in the .NET Framework use this pattern—files, streams (disk I/O, memory, network), database-access types, and communication ports.

The main drawback to Close() is that it makes sharing the object between clients much more complex than COM's reference counting. The clients must coordinate which one of them is responsible for calling Close() and when it should be called—that is, when it's safe to call Close() without affecting other clients that might still want to use the object. As a result, the clients are coupled to each other.

Another problem is that you must decide where to implement Open and Close—on every interface the object supports, on a dedicated interface, or on the class directly as public methods. Whichever choice you make inevitably couples clients to your specific object-finalization mechanism. If the mechanism changes, then the change triggers a cascade of changes on all the clients.

A more common situation is one in which disposing of the resources the object holds amounts to destroying the object. In this case, the object should implement a method called Dispose():

```
void Dispose(){...}
```

The object disposes of all its expensive resources when a client calls Dispose(), and the client shouldn't try to access the object again. In essence, you put the same cleanup code in Dispose() that you'd put in the destructor, except you don't wait until garbage collection for the cleanup. If the object's base class has a Dispose() method, then the object should also call its base class's implementation of Dispose() to dispose of resources the base class holds.

Using Dispose() has similar problems to using Close(). Sharing the object between clients couples the clients to each other and to the object-finalization mechanism. And again, it's not clear where you should implement Dispose().

A better design approach to implementing Dispose() is to have it on a separate interface altogether. This special interface (defined in the System namespace) is called IDisposable:

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

In the object's implementation of Dispose(), the object disposes of all the expensive resources it holds:

```
public class MyClass : IDisposable
{
   public void MyMethod()
   {...}
   public void Dispose()
   {
      //do object cleanup, call
      //base.Dispose() if it has one
   }
   //More methods and resources
}
```

Having the Dispose() method on a separate interface allows the client to use the object's domain-specific methods, query for the presence of IDisposable, then call it—regardless of the object's actual type and finalization mechanism:

```
MyClass obj;
obj = new MyClass();
obj.MyMethod();

//Client wants to dispose whatever needs
//disposing:
IDisposable disp = obj as
   IDisposable;
if(disp != null)
{
   disp.Dispose();
}
```

The client uses the "as" operator to call Dispose(). The client doesn't know for certain if the object supports IDisposable. The client finds out in a safe manner, because "as" returns null if the object doesn't support it.

Clean Up After Exceptions

Whether you use IDisposable or only Dispose() as a method, you should scope disposing of the resources in try/finally blocks, because you should still call Dispose() if you get an exception:

```
MyClass obj;
obj = new MyClass();
try
{
   obj.SomeMethod();
}
finally
{
      IDisposable disp;
         disp = obj as IDisposable;
      if(disp!= null)
      {
         disp.Dispose();
      }
}
```

The problem with the preceding programming model is that the code gets messy if multiple objects are involved, because every one of them can throw an exception, and you should still clean up after using them. To automate calling Dispose() with proper error handling, C# supports the using statement to generate a try/catch block automatically. Suppose you have this class definition:

```
public class MyClass: IDisposable
{
   public void SomeMethod(){...}
   public void Dispose(){...}
   /* Expensive resources here */
}
```

The client code is:

```
MyClass obj;
obj = new MyClass();
using(obj)
{
   obj.SomeMethod();
}
```

The C# compiler converts the destructor definition to a Finalize() method, and also calls your base class's Finalize() method.

The C# compiler converts the preceding code to code semantically equivalent to this:

```
MyClass obj;
obj = new MyClass();
try
{
   obj.SomeMethod();
}
finally
{
   if(obj != null)
   {
      IDisposable disp;
      disp = obj;
      disp.Dispose();
   }
}
```

You can even stack multiple using statements on top of each other to handle using multiple objects:

```
MyClass obj1;
MyClass obj2;
MyClass obj3;

obj1 = new MyClass();
obj2 = new MyClass();
obj3 = new MyClass();
using(obj1)
using(obj2)
using(obj3)
```

```
{
   obj1.SomeMethod();
   obj2.SomeMethod();
   obj3.SomeMethod();
}
```

The using statement has one liability: The compiler-generated code uses a type-safe, implicit cast to IDisposable from the object. As a result, the type in the statement must derive from IDisposable, and the C# compiler enforces this. This precludes using the using statement with interfaces in the general case, even if the implementing type supports IDisposable.

However, two workarounds allow you to combine interfaces with the using statement. The first is to derive the interfaces from IDisposable. This approach is analogous to having every COM interface derive from IUnknown so that the interface has the reference-counting methods. The second workaround is to coerce the type you use to IDisposable by using an explicit cast to fool the compiler:

```
public interface IMyInterface
{
   void SomeMethod();
}
public class MyClass: IMyInterface,IDisposable
{
   public void SomeMethod(){}
   public void Dispose(){}
}
IMyInterface obj;
obj = new MyClass();
using((IDisposable)obj)
{
   obj.SomeMethod();
}
```

Dispose() and the C# destructor aren't mutually exclusive, and you should provide both. The reason is simple: When you have expensive resources to dispose of, there's no guarantee the client will actually call Dispose() even if you provide it—for example, if unhandled exceptions occur on the client side.

Combine Finalize() With Dispose

If the client doesn't call Dispose(), your fallback is to use the Finalize() method (the destructor in C#) and do the resources cleanup there. If Dispose() is called, there's no point in trying to dispose of the resources again in Finalize(). If you call Dispose(), the object should suppress finalization by calling the GC class's static SuppressFinalize() method, passing itself as a parameter:

```
public static void SuppressFinalize(object obj);
```

This prevents the object from being added to the finalization queue, as though the object definition doesn't contain a Finalize() method.

You must pay attention to some more details. The object should channel the Dispose() and Finalize() implementation to the same method to enforce the fact that it's doing exactly the same thing in either case. The object should handle multiple Dispose() calls. It should also detect in every method whether a client has called Dispose() already, refuse to execute the method if it has, then throw an exception instead.

The object should also handle class hierarchies properly and call its base class's Dispose() or Finalize() method. Clearly, implementing a bullet-proof Dispose() and Finalize() involves many details when inheritance is involved. The good news is that you can use a generic template, available in both C# and VB.NET code, for such cases (download Listing 1 from the *VSM* Web site; see the Go Online box for details). Only the top-most base class in the template implements IDisposable. Subclasses can't override this class because it doesn't provide a virtual Dispose() method. The top-most base class's Dispose() method calls a virtual protected Cleanup() method. This Cleanup() method serves both IDisposable.Dispose() and Finalize() (the C# destructor) as the single channeled implementation of the cleanup code.

The Dispose() method uses the lock statement to synchronize access to the method, because multiple clients on multiple threads might try to dispose of the object at the same time. The top-most base class maintains a Boolean flag called m_Disposed, signaling whether Dispose() was called already. The first time a client calls Dispose(), Dispose() sets m_Disposed to true, thereby preventing itself from calling Cleanup() again. As a result, calling Dispose() multiple times is benign. The top-most base class provides a thread-safe, read-only property called Disposed, which every method in the base class or the subclasses should check before executing method bodies—and throw ObjectDisposedException if Dispose() was called.

Every class in the hierarchy should implement its version of Cleanup() if it has cleanup to do. Note also that only the top-most base class should have a destructor. All the destructor is doing is delegating to the overridden protected Cleanup(). Of course, .NET never calls the destructor if you call Dispose() first, because Dispose() suppresses finalization.

Like most other aspects of .NET, object finalization involves a spectrum of options that cater to the widest range of developer preferences. At one end of the spectrum, you can let .NET manage the object's lifecycle and focus only on the business logic. At the other end, you can use the deterministic finalization template to ensure application scalability and throughput. You're on your way to applying .NET successfully—regardless of where you put yourself on that spectrum—as long as you understand your programming model's tradeoffs and implications. VSM

Juval Löwy is a software architect and the principal of IDesign, a consulting and training company focused on .NET design and migration. Juval is Microsoft's regional director for the Silicon Valley, working with Microsoft on helping the industry adopt .NET. This article derives from his latest book, *Programming .NET Components* (O'Reilly & Associates). Contact him at www.idesign.net.

Additional Resources

"Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework" by Jeffrey Richter: www.msdn.microsoft.com/msdnmag/issues/1100/gci/default.aspx