**Dispose Pattern**

**.NET Framework 4.5**

[Other Versions](javascript:;)

Description: http://i.msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png

* [.NET Framework 4](http://msdn.microsoft.com/en-us/library/b1yfkh5e(d=printer,v=vs.100).aspx)
* [.NET Framework 3.5](http://msdn.microsoft.com/en-us/library/b1yfkh5e(d=printer,v=vs.90).aspx)
* [.NET Framework 3.0](http://msdn.microsoft.com/en-us/library/b1yfkh5e(d=printer,v=vs.85).aspx)
* [.NET Framework 2.0](http://msdn.microsoft.com/en-us/library/b1yfkh5e(d=printer,v=vs.80).aspx)
* [.NET Framework 1.1](http://msdn.microsoft.com/en-us/library/b1yfkh5e(d=printer,v=vs.71).aspx)

All programs acquire one or more system resources, such as memory, system handles, or database connections, during the course of their execution. Developers have to be careful when using such system resources, because they must be released after they have been acquired and used.

The CLR provides support for automatic memory management. Managed memory (memory allocated using the C# operator new) does not need to be explicitly released. It is released automatically by the garbage collector (GC). This frees developers from the tedious and difficult task of releasing memory and has been one of the main reasons for the unprecedented productivity afforded by the .NET Framework.

Unfortunately, managed memory is just one of many types of system resources. Resources other than managed memory still need to be released explicitly and are referred to as unmanaged resources. The GC was specifically not designed to manage such unmanaged resources, which means that the responsibility for managing unmanaged resources lies in the hands of the developers.

The CLR provides some help in releasing unmanaged resources. [System.Object](http://msdn.microsoft.com/en-us/library/system.object(v=vs.110).aspx) declares a virtual method [Finalize](http://msdn.microsoft.com/en-us/library/system.object.finalize(v=vs.110).aspx) (also called the finalizer) that is called by the GC before the object’s memory is reclaimed by the GC and can be overridden to release unmanaged resources. Types that override the finalizer are referred to as finalizable types.

Although finalizers are effective in some cleanup scenarios, they have two significant drawbacks:

* The finalizer is called when the GC detects that an object is eligible for collection. This happens at some undetermined period of time after the resource is not needed anymore. The delay between when the developer could or would like to release the resource and the time when the resource is actually released by the finalizer might be unacceptable in programs that acquire many scarce resources (resources that can be easily exhausted) or in cases in which resources are costly to keep in use (e.g., large unmanaged memory buffers).
* When the CLR needs to call a finalizer, it must postpone collection of the object’s memory until the next round of garbage collection (the finalizers run between collections). This means that the object’s memory (and all objects it refers to) will not be released for a longer period of time.

Therefore, relying exclusively on finalizers might not be appropriate in many scenarios when it is important to reclaim unmanaged resources as quickly as possible, when dealing with scarce resources, or in highly performant scenarios in which the added GC overhead of finalization is unacceptable.

The Framework provides the [System.IDisposable](http://msdn.microsoft.com/en-us/library/system.idisposable(v=vs.110).aspx) interface that should be implemented to provide the developer a manual way to release unmanaged resources as soon as they are not needed. It also provides the [GC.SuppressFinalize](http://msdn.microsoft.com/en-us/library/system.gc.suppressfinalize(v=vs.110).aspx) method that can tell the GC that an object was manually disposed of and does not need to be finalized anymore, in which case the object’s memory can be reclaimed earlier. Types that implement the IDisposable interface are referred to as disposable types.

The Dispose Pattern is intended to standardize the usage and implementation of finalizers and the IDisposable interface.

The main motivation for the pattern is to reduce the complexity of the implementation of the [Finalize](http://msdn.microsoft.com/en-us/library/system.object.finalize(v=vs.110).aspx) and the [Dispose](http://msdn.microsoft.com/en-us/library/system.idisposable.dispose(v=vs.110).aspx) methods. The complexity stems from the fact that the methods share some but not all code paths (the differences are described later in the chapter). In addition, there are historical reasons for some elements of the pattern related to the evolution of language support for deterministic resource management.

√ DO implement the Basic Dispose Pattern on types containing instances of disposable types. See the [Basic Dispose Pattern](http://msdn.microsoft.com/en-us/library/b1yfkh5e(d=printer,v=vs.110).aspx#basic_pattern) section for details on the basic pattern.

If a type is responsible for the lifetime of other disposable objects, developers need a way to dispose of them, too. Using the container’s Dispose method is a convenient way to make this possible.

√ DO implement the Basic Dispose Pattern and provide a finalizer on types holding resources that need to be freed explicitly and that do not have finalizers.

For example, the pattern should be implemented on types storing unmanaged memory buffers. The [Finalizable Types](http://msdn.microsoft.com/en-us/library/b1yfkh5e(d=printer,v=vs.110).aspx#finalizable_types) section discusses guidelines related to implementing finalizers.

√ CONSIDER implementing the Basic Dispose Pattern on classes that themselves don’t hold unmanaged resources or disposable objects but are likely to have subtypes that do.

A great example of this is the [System.IO.Stream](http://msdn.microsoft.com/en-us/library/system.io.stream(v=vs.110).aspx) class. Although it is an abstract base class that doesn’t hold any resources, most of its subclasses do and because of this, it implements this pattern.

[Basic Dispose Pattern](javascript:void(0))

The basic implementation of the pattern involves implementing the System.IDisposable interface and declaring the Dispose(bool) method that implements all resource cleanup logic to be shared between the Dispose method and the optional finalizer.

The following example shows a simple implementation of the basic pattern:

public class DisposableResourceHolder : IDisposable {

private SafeHandle resource; // handle to a resource

public DisposableResourceHolder(){

this.resource = ... // allocates the resource

}

public void Dispose(){

Dispose(true);

GC.SuppressFinalize(this);

}

protected virtual void Dispose(bool disposing){

if (disposing){

if (resource!= null) resource.Dispose();

}

}

}

The Boolean parameter disposing indicates whether the method was invoked from the IDisposable.Dispose implementation or from the finalizer. The Dispose(bool) implementation should check the parameter before accessing other reference objects (e.g., the resource field in the preceding sample). Such objects should only be accessed when the method is called from the IDisposable.Dispose implementation (when the disposing parameter is equal to true). If the method is invoked from the finalizer (disposing is false), other objects should not be accessed. The reason is that objects are finalized in an unpredictable order and so they, or any of their dependencies, might already have been finalized.

Also, this section applies to classes with a base that does not already implement the Dispose Pattern. If you are inheriting from a class that already implements the pattern, simply override the Dispose(bool) method to provide additional resource cleanup logic.

√ DO declare a protected virtual void Dispose(bool disposing) method to centralize all logic related to releasing unmanaged resources.

All resource cleanup should occur in this method. The method is called from both the finalizer and the IDisposable.Dispose method. The parameter will be false if being invoked from inside a finalizer. It should be used to ensure any code running during finalization is not accessing other finalizable objects. Details of implementing finalizers are described in the next section.

protected virtual void Dispose(bool disposing){

if (disposing){

if (resource!= null) resource.Dispose();

}

}

√ DO implement the IDisposable interface by simply calling Dispose(true) followed by GC.SuppressFinalize(this).

The call to SuppressFinalize should only occur if Dispose(true) executes successfully.

public void Dispose(){

Dispose(true);

GC.SuppressFinalize(this);

}

X DO NOT make the parameterless Dispose method virtual.

The Dispose(bool) method is the one that should be overridden by subclasses.

// bad design

public class DisposableResourceHolder : IDisposable {

public virtual void Dispose(){ ... }

protected virtual void Dispose(bool disposing){ ... }

}

// good design

public class DisposableResourceHolder : IDisposable {

public void Dispose(){ ... }

protected virtual void Dispose(bool disposing){ ... }

}

X DO NOT declare any overloads of the Dispose method other than Dispose() and Dispose(bool).

Dispose should be considered a reserved word to help codify this pattern and prevent confusion among implementers, users, and compilers. Some languages might choose to automatically implement this pattern on certain types.

√ DO allow the Dispose(bool) method to be called more than once. The method might choose to do nothing after the first call.

public class DisposableResourceHolder : IDisposable {

bool disposed = false;

protected virtual void Dispose(bool disposing){

if(disposed) return;

// cleanup

...

disposed = true;

}

}

X AVOID throwing an exception from within Dispose(bool) except under critical situations where the containing process has been corrupted (leaks, inconsistent shared state, etc.).

Users expect that a call to Dispose will not raise an exception.

If Dispose could raise an exception, further finally-block cleanup logic will not execute. To work around this, the user would need to wrap every call to Dispose (within the finally block!) in a try block, which leads to very complex cleanup handlers. If executing a Dispose(bool disposing) method, never throw an exception if disposing is false. Doing so will terminate the process if executing inside a finalizer context.

√ DO throw an [ObjectDisposedException](http://msdn.microsoft.com/en-us/library/system.objectdisposedexception(v=vs.110).aspx) from any member that cannot be used after the object has been disposed of.

public class DisposableResourceHolder : IDisposable {

bool disposed = false;

SafeHandle resource; // handle to a resource

public void DoSomething(){

if(disposed) throw new ObjectDisposedException(...);

// now call some native methods using the resource

...

}

protected virtual void Dispose(bool disposing){

if(disposed) return;

// cleanup

...

disposed = true;

}

}

√ CONSIDER providing method Close(), in addition to the Dispose(), if close is standard terminology in the area.

When doing so, it is important that you make the Close implementation identical to Dispose and consider implementing the IDisposable.Dispose method explicitly.

public class Stream : IDisposable {

IDisposable.Dispose(){

Close();

}

public void Close(){

Dispose(true);

GC.SuppressFinalize(this);

}

}

[Finalizable Types](javascript:void(0))

Finalizable types are types that extend the Basic Dispose Pattern by overriding the finalizer and providing finalization code path in the Dispose(bool) method.

Finalizers are notoriously difficult to implement correctly, primarily because you cannot make certain (normally valid) assumptions about the state of the system during their execution. The following guidelines should be taken into careful consideration.

Note that some of the guidelines apply not just to the Finalize method, but to any code called from a finalizer. In the case of the Basic Dispose Pattern previously defined, this means logic that executes inside Dispose(bool disposing) when the disposing parameter is false.

If the base class already is finalizable and implements the Basic Dispose Pattern, you should not override Finalize again. You should instead just override the Dispose(bool) method to provide additional resource cleanup logic.

The following code shows an example of a finalizable type:

public class ComplexResourceHolder : IDisposable {

private IntPtr buffer; // unmanaged memory buffer

private SafeHandle resource; // disposable handle to a resource

public ComplexResourceHolder(){

this.buffer = ... // allocates memory

this.resource = ... // allocates the resource

}

protected virtual void Dispose(bool disposing){

ReleaseBuffer(buffer); // release unmanaged memory

if (disposing){ // release other disposable objects

if (resource!= null) resource.Dispose();

}

}

~ ComplexResourceHolder(){

Dispose(false);

}

public void Dispose(){

Dispose(true);

GC.SuppressFinalize(this);

}

}

X AVOID making types finalizable.

Carefully consider any case in which you think a finalizer is needed. There is a real cost associated with instances with finalizers, from both a performance and code complexity standpoint. Prefer using resource wrappers such as [SafeHandle](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.safehandle(v=vs.110).aspx) to encapsulate unmanaged resources where possible, in which case a finalizer becomes unnecessary because the wrapper is responsible for its own resource cleanup.

X DO NOT make value types finalizable.

Only reference types actually get finalized by the CLR, and thus any attempt to place a finalizer on a value type will be ignored. The C# and C++ compilers enforce this rule.

√ DO make a type finalizable if the type is responsible for releasing an unmanaged resource that does not have its own finalizer.

When implementing the finalizer, simply call Dispose(false) and place all resource cleanup logic inside the Dispose(bool disposing) method.

public class ComplexResourceHolder : IDisposable {

~ ComplexResourceHolder(){

Dispose(false);

}

protected virtual void Dispose(bool disposing){

...

}

}

√ DO implement the Basic Dispose Pattern on every finalizable type.

This gives users of the type a means to explicitly perform deterministic cleanup of those same resources for which the finalizer is responsible.

X DO NOT access any finalizable objects in the finalizer code path, because there is significant risk that they will have already been finalized.

For example, a finalizable object A that has a reference to another finalizable object B cannot reliably use B in A’s finalizer, or vice versa. Finalizers are called in a random order (short of a weak ordering guarantee for critical finalization).

Also, be aware that objects stored in static variables will get collected at certain points during an application domain unload or while exiting the process. Accessing a static variable that refers to a finalizable object (or calling a static method that might use values stored in static variables) might not be safe if [Environment.HasShutdownStarted](http://msdn.microsoft.com/en-us/library/system.environment.hasshutdownstarted(v=vs.110).aspx) returns true.

√ DO make your Finalize method protected.

C#, C++, and VB.NET developers do not need to worry about this, because the compilers help to enforce this guideline.

X DO NOT let exceptions escape from the finalizer logic, except for system-critical failures.

If an exception is thrown from a finalizer, the CLR will shut down the entire process (as of .NET Framework version 2.0), preventing other finalizers from executing and resources from being released in a controlled manner.

√ CONSIDER creating and using a critical finalizable object (a type with a type hierarchy that contains [CriticalFinalizerObject](http://msdn.microsoft.com/en-us/library/system.runtime.constrainedexecution.criticalfinalizerobject(v=vs.110).aspx)) for situations in which a finalizer absolutely must execute even in the face of forced application domain unloads and thread aborts.

*Portions © 2005, 2009 Microsoft Corporation. All rights reserved.*

*Reprinted by permission of Pearson Education, Inc. from* [*Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*](http://www.informit.com/store/framework-design-guidelines-conventions-idioms-and-9780321545619) *by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.*