

# Object.Finalize Method

Namespace: [System](#)

Assembly: System.Runtime.dll

## In this article

[Definition](#)


[Examples](#)

[Remarks](#)

[Applies to](#)

[See also](#)

Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.

C#	 Copy
<pre>~Object ();</pre>	

## Examples

The following example verifies that the [Finalize](#) method is called when an object that overrides [Finalize](#) is destroyed. Note that, in a production application, the [Finalize](#) method would be overridden to release unmanaged resources held by the object. Also note that the C# example provides a destructor instead of overriding the [Finalize](#) method.

C#	 Copy
<pre>using System; using System.Diagnostics;  public class ExampleClass {     Stopwatch sw;      public ExampleClass()     {         sw = Stopwatch.StartNew();     } }</pre>	

```

        Console.WriteLine("Instantiated object");
    }

    public void ShowDuration()
    {
        Console.WriteLine("This instance of {0} has been in existence for
{1}",
                        this, sw.Elapsed);
    }

    ~ExampleClass()
    {
        Console.WriteLine("Finalizing object");
        sw.Stop();
        Console.WriteLine("This instance of {0} has been in existence for
{1}",
                        this, sw.Elapsed);
    }
}

public class Demo
{
    public static void Main()
    {
        ExampleClass ex = new ExampleClass();
        ex.ShowDuration();
    }
}

// The example displays output like the following:
//     Instantiated object
//     This instance of ExampleClass has been in existence for
00:00:00.0011060
//     Finalizing object
//     This instance of ExampleClass has been in existence for
00:00:00.0036294

```

For an additional example that overrides the [Finalize](#) method, see the [GC.SuppressFinalize](#) method.

## Remarks

The [Finalize](#) method is used to perform cleanup operations on unmanaged resources held by the current object before the object is destroyed. The method is protected and therefore is accessible only through this class or through a derived class.

In this section:

- [How finalization works](#)
- [Notes for implementers](#)
- [The SafeHandle alternative](#)

## How finalization works

The [Object](#) class provides no implementation for the [Finalize](#) method, and the garbage collector does not mark types derived from [Object](#) for finalization unless they override the [Finalize](#) method.

If a type does override the [Finalize](#) method, the garbage collector adds an entry for each instance of the type to an internal structure called the finalization queue. The finalization queue contains entries for all the objects in the managed heap whose finalization code must run before the garbage collector can reclaim their memory. The garbage collector then calls the [Finalize](#) method automatically under the following conditions:

- After the garbage collector has discovered that an object is inaccessible, unless the object has been exempted from finalization by a call to the [GC.SuppressFinalize](#) method.
- **On .NET Framework only**, during shutdown of an application domain, unless the object is exempt from finalization. During shutdown, even objects that are still accessible are finalized.

[Finalize](#) is automatically called only once on a given instance, unless the object is re-registered by using a mechanism such as [GC.ReRegisterForFinalize](#) and the [GC.SuppressFinalize](#) method has not been subsequently called.

[Finalize](#) operations have the following limitations:

- The exact time when the finalizer executes is undefined. To ensure deterministic release of resources for instances of your class, implement a `Close` method or provide a [IDisposable.Dispose](#) implementation.
- The finalizers of two objects are not guaranteed to run in any specific order, even if one object refers to the other. That is, if Object A has a reference to Object B and both have finalizers, Object B might have already been finalized when the finalizer of Object A starts.
- The thread on which the finalizer runs is unspecified.

The [Finalize](#) method might not run to completion or might not run at all under the following exceptional circumstances:

- If another finalizer blocks indefinitely (goes into an infinite loop, tries to obtain a lock it can never obtain, and so on). Because the runtime tries to run finalizers to completion, other finalizers might not be called if a finalizer blocks indefinitely.
- If the process terminates without giving the runtime a chance to clean up. In this case, the runtime's first notification of process termination is a `DLL_PROCESS_DETACH` notification.

The runtime continues to finalize objects during shutdown only while the number of finalizable objects continues to decrease.

If [Finalize](#) or an override of [Finalize](#) throws an exception, and the runtime is not hosted by an application that overrides the default policy, the runtime terminates the process and no active `try/finally` blocks or finalizers are executed. This behavior ensures process integrity if the finalizer cannot free or destroy resources.

## Overriding the Finalize method

You should override [Finalize](#) for a class that uses unmanaged resources, such as file handles or database connections that must be released when the managed object that uses them is discarded during garbage collection. You shouldn't implement a [Finalize](#) method for managed objects because the garbage collector releases managed resources automatically.

### Important

If a **SafeHandle** object is available that wraps your unmanaged resource, the recommended alternative is to implement the dispose pattern with a safe handle and not override [Finalize](#). For more information, see [The SafeHandle alternative](#) section.

The [Object.Finalize](#) method does nothing by default, but you should override [Finalize](#) only if necessary, and only to release unmanaged resources. Reclaiming memory tends to take much longer if a finalization operation runs, because it requires at least two garbage collections. In addition, you should override the [Finalize](#) method for reference types only. The common language runtime only finalizes reference types. It ignores finalizers on value types.

The scope of the [Object.Finalize](#) method is `protected`. You should maintain this limited scope when you override the method in your class. By keeping a [Finalize](#) method protected, you prevent users of your application from calling an object's [Finalize](#) method directly.

Every implementation of [Finalize](#) in a derived type must call its base type's implementation of [Finalize](#). This is the only case in which application code is allowed to call [Finalize](#). An object's [Finalize](#) method shouldn't call a method on any objects other than that of its base class. This is because the other objects being called could be collected at the same time as the calling object, such as in the case of a common language runtime shutdown.

#### ⓘ Note

The C# compiler does not allow you to override the [Finalize](#) method. Instead, you provide a finalizer by implementing a [destructor](#) for your class. A C# destructor automatically calls the destructor of its base class.

Visual C++ also provides its own syntax for implementing the [Finalize](#) method. For more information, see the "Destructors and finalizers" section of [How to: Define and Consume Classes and Structs \(C++/CLI\)](#).

Because garbage collection is non-deterministic, you do not know precisely when the garbage collector performs finalization. To release resources immediately, you can also choose to implement the [dispose pattern](#) and the [IDisposable](#) interface. The [IDisposable.Dispose](#) implementation can be called by consumers of your class to free unmanaged resources, and you can use the [Finalize](#) method to free unmanaged resources in the event that the [Dispose](#) method is not called.

[Finalize](#) can take almost any action, including resurrecting an object (that is, making the object accessible again) after it has been cleaned up during garbage collection. However, the object can only be resurrected once; [Finalize](#) cannot be called on resurrected objects during garbage collection.

## The SafeHandle alternative

Creating reliable finalizers is often difficult, because you cannot make assumptions about the state of your application, and because unhandled system exceptions such as [OutOfMemoryException](#) and [StackOverflowException](#) terminate the finalizer. Instead of implementing a finalizer for your class to release unmanaged resources, you can use an

object that is derived from the [System.Runtime.InteropServices.SafeHandle](#) class to wrap your unmanaged resources, and then implement the dispose pattern without a finalizer. The .NET Framework provides the following classes in the [Microsoft.Win32](#) namespace that are derived from [System.Runtime.InteropServices.SafeHandle](#):

- [SafeFileHandle](#) is a wrapper class for a file handle.
- [SafeMemoryMappedFileHandle](#) is a wrapper class for memory-mapped file handles.
- [SafeMemoryMappedViewHandle](#) is a wrapper class for a pointer to a block of unmanaged memory.
- [SafeNCryptKeyHandle](#), [SafeNCryptProviderHandle](#), and [SafeNCryptSecretHandle](#) are wrapper classes for cryptographic handles.
- [SafePipeHandle](#) is a wrapper class for pipe handles.
- [SafeRegistryHandle](#) is a wrapper class for a handle to a registry key.
- [SafeWaitHandle](#) is a wrapper class for a wait handle.

The following example uses the [dispose pattern](#) with safe handles instead of overriding the [Finalize](#) method. It defines a `FileAssociation` class that wraps registry information about the application that handles files with a particular file extension. The two registry handles returned as `out` parameters by Windows [RegOpenKeyEx](#) function calls are passed to the [SafeRegistryHandle](#) constructor. The type's protected `Dispose` method then calls the `SafeRegistryHandle.Dispose` method to free these two handles.

C#

 Copy

```
using Microsoft.Win32.SafeHandles;
using System;
using System.ComponentModel;
using System.IO;
using System.Runtime.InteropServices;

public class FileAssociationInfo : IDisposable
{
    // Private variables.
    private String ext;
    private String openCmd;
    private String args;
    private SafeRegistryHandle hExtHandle, hAppIdHandle;

    // Windows API calls.
    [DllImport("advapi32.dll", CharSet= CharSet.Auto,
```

```

SetLastError=true)]
    private static extern int RegOpenKeyEx(IntPtr hKey,
        String lpSubKey, int ulOptions, int samDesired,
        out IntPtr phkResult);
    [DllImport("advapi32.dll", CharSet= CharSet.Unicode, EntryPoint =
"RegQueryValueExW",
        SetLastError=true)]
    private static extern int RegQueryValueEx(IntPtr hKey,
        string lpValueName, int lpReserved, out uint lpType,
        string lpData, ref uint lpcbData);
    [DllImport("advapi32.dll", SetLastError = true)]
    private static extern int RegSetValueEx(IntPtr hKey,
[MarshalAs(UnmanagedType.LPStr)] string lpValueName,
        int Reserved, uint dwType,
[MarshalAs(UnmanagedType.LPStr)] string lpData,
        int cpData);
    [DllImport("advapi32.dll", SetLastError=true)]
    private static extern int RegCloseKey(UIntPtr hKey);

    // Windows API constants.
    private const int HKEY_CLASSES_ROOT = unchecked((int) 0x80000000);
    private const int ERROR_SUCCESS = 0;

    private const int KEY_QUERY_VALUE = 1;
    private const int KEY_SET_VALUE = 0x2;

    private const uint REG_SZ = 1;

    private const int MAX_PATH = 260;

    public FileAssociationInfo(String fileExtension)
    {
        int retVal = 0;
        uint lpType = 0;

        if (!fileExtension.StartsWith("."))
            fileExtension = "." + fileExtension;
        ext = fileExtension;

        IntPtr hExtension = IntPtr.Zero;
        // Get the file extension value.
        retVal = RegOpenKeyEx(new IntPtr(HKEY_CLASSES_ROOT),
fileExtension, 0, KEY_QUERY_VALUE, out hExtension);
        if (retVal != ERROR_SUCCESS)
            throw new Win32Exception(retVal);
        // Instantiate the first SafeRegistryHandle.
        hExtHandle = new SafeRegistryHandle(hExtension, true);

        string appId = new string(' ', MAX_PATH);
        uint appIdLength = (uint) appId.Length;
        retVal = RegQueryValueEx(hExtHandle.DangerousGetHandle(),

```

```

String.Empty, 0, out lpType, appId, ref appIdLength);
    if (retVal != ERROR_SUCCESS)
        throw new Win32Exception(retVal);
    // We no longer need the hExtension handle.
    hExtHandle.Dispose();

    // Determine the number of characters without the terminating
null.
    appId = appId.Substring(0, (int) appIdLength / 2 - 1) +
@"\shell\open\Command";

    // Open the application identifier key.
    string exeName = new string(' ', MAX_PATH);
    uint exeNameLength = (uint) exeName.Length;
    IntPtr hAppId;
    retVal = RegOpenKeyEx(new IntPtr(HKEY_CLASSES_ROOT), appId, 0,
KEY_QUERY_VALUE | KEY_SET_VALUE,
                        out hAppId);
    if (retVal != ERROR_SUCCESS)
        throw new Win32Exception(retVal);

    // Instantiate the second SafeRegistryHandle.
    hAppIdHandle = new SafeRegistryHandle(hAppId, true);

    // Get the executable name for this file type.
    string exePath = new string(' ', MAX_PATH);
    uint exePathLength = (uint) exePath.Length;
    retVal = RegQueryValueEx(hAppIdHandle.DangerousGetHandle(),
String.Empty, 0, out lpType, exePath, ref exePathLength);
    if (retVal != ERROR_SUCCESS)
        throw new Win32Exception(retVal);

    // Determine the number of characters without the terminating
null.
    exePath = exePath.Substring(0, (int) exePathLength / 2 - 1);
    // Remove any environment strings.
    exePath = Environment.ExpandEnvironmentVariables(exePath);

    int position = exePath.IndexOf('%');
    if (position >= 0) {
        args = exePath.Substring(position);
        // Remove command line parameters ('%0', etc.).
        exePath = exePath.Substring(0, position).Trim();
    }
    openCmd = exePath;
}

public String Extension
{ get { return ext; } }

public String Open

```



```

    { get { return openCmd; }
      set {
          if (hAppIdHandle.IsInvalid | hAppIdHandle.IsClosed)
              throw new InvalidOperationException("Cannot write to registry key.");
          if (! File.Exists(value)) {
              string message = String.Format("'{0}' does not exist",
value);
              throw new FileNotFoundException(message);
          }
          string cmd = value + " %1";
          int retVal = RegSetValueEx(hAppIdHandle.DangerousGetHandle(),
String.Empty, 0,
                                     REG_SZ, value, value.Length + 1);
          if (retVal != ERROR_SUCCESS)
              throw new Win32Exception(retVal);
      } }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected void Dispose(bool disposing)
    {
        // Ordinarily, we release unmanaged resources here;
        // but all are wrapped by safe handles.

        // Release disposable objects.
        if (disposing) {
            if (hExtHandle != null) hExtHandle.Dispose();
            if (hAppIdHandle != null) hAppIdHandle.Dispose();
        }
    }
}

```

## Applies to

### .NET

5.0 RC1

### .NET Core

3.1, 3.0, 2.2, 2.1, 2.0, 1.1, 1.0

## .NET Framework

4.8, 4.7.2, 4.7.1, 4.7, 4.6.2, 4.6.1, 4.6, 4.5.2, 4.5.1, 4.5, 4.0, 3.5, 3.0, 2.0, 1.1

## .NET Standard

2.1, 2.0, 1.6, 1.5, 1.4, 1.3, 1.2, 1.1, 1.0

## UWP

10.0

## Xamarin.Android

7.1

## Xamarin.iOS

10.8

## Xamarin.Mac

3.0

## See also

- [SuppressFinalize\(Object\)](#)
- [ReRegisterForFinalize\(Object\)](#)
- [WaitForPendingFinalizers\(\)](#)
- [WeakReference](#)

---

Is this page helpful?

 Yes  No

---