# Programming with Application Domains and Assemblies

Article • 06/04/2024

> **Note**
>
> This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

Hosts such as ASP.NET and the Windows shell load the common language runtime into a process, create an application domain in that process, and then load and execute user code in that application domain when running a .NET Framework application. In most cases, you do not have to worry about creating application domains and loading assemblies into them because the runtime host performs those tasks.

However, if you're creating an application that will host the common language runtime, creating tools or code you want to unload programmatically, or creating pluggable components that can be unloaded and reloaded on the fly, you will be creating your own application domains. Even if you are not creating a runtime host, this section provides important information on how to work with application domains and assemblies loaded in these application domains.

## In This Section

Using Application Domains
Provides examples of creating, configuring, and using application domains.

Programming with Assemblies
Describes how to create, sign, and set attributes on assemblies.

## Related Sections

Emitting Dynamic Methods and Assemblies
Describes how to create dynamic assemblies.

Assemblies in .NET
Provides a conceptual overview of assemblies.

Application Domains

Provides a conceptual overview of application domains.

Reflection Overview

Describes how to use the **Reflection** class to obtain information about an assembly.

# Assembly Placement

Article • 06/04/2024

**Note**

This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

For most .NET Framework applications, you locate assemblies that make up an application in the application's directory, in a subdirectory of the application's directory, or in the global assembly cache (if the assembly is shared). You can override where the common language runtime looks for an assembly by using the <codeBase> Element in a configuration file. If the assembly does not have a strong name, the location specified using the <codeBase> Element is restricted to the application directory or a subdirectory. If the assembly has a strong name, the <codeBase> Element can specify any location on the computer or on a network.

Similar rules apply to locating assemblies when working with unmanaged code or COM interop applications: if the assembly will be shared by multiple applications, it should be installed into the global assembly cache. Assemblies used with unmanaged code must be exported as a type library and registered. Assemblies used by COM interop must be registered in the catalog, although in some cases this registration occurs automatically.

## See also

- How the Runtime Locates Assemblies
- Configuring Apps
- Interoperating with unmanaged code
- Assemblies in .NET

# Global Assembly Cache

Article • 09/15/2021

> ⓘ **Note**
>
> This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

Each computer where the Common Language Runtime is installed has a machine-wide code cache called the Global Assembly Cache. The Global Assembly Cache stores assemblies specifically designated to be shared by several applications on the computer.

You should share assemblies by installing them into the Global Assembly Cache only when you need to. As a general guideline, keep assembly dependencies private, and locate assemblies in the application directory unless sharing an assembly is explicitly required. In addition, it is not necessary to install assemblies into the Global Assembly Cache to make them accessible to COM interop or unmanaged code.

> ⓘ **Note**
>
> There are scenarios where you explicitly do not want to install an assembly into the Global Assembly Cache. If you place one of the assemblies that make up an application in the Global Assembly Cache, you can no longer replicate or install the application by using the **xcopy** command to copy the application directory. You must move the assembly in the Global Assembly Cache as well.

There are two ways to deploy an assembly into the Global Assembly Cache:

- Use an installer designed to work with the Global Assembly Cache. This is the preferred option for installing assemblies into the Global Assembly Cache.

- Use a developer tool called the Global Assembly Cache tool (Gacutil.exe), provided by the Windows SDK.

  > ⓘ **Note**
  >
  > In deployment scenarios, use Windows Installer to install assemblies into the Global Assembly Cache. Use the Global Assembly Cache tool only in

development scenarios, because it does not provide assembly reference counting and other features provided when using the Windows Installer.

Starting with the .NET Framework 4, the default location for the Global Assembly Cache is **%windir%\Microsoft.NET\assembly**. In earlier versions of the .NET Framework, the default location is **%windir%\assembly**.

Administrators often protect the systemroot directory using an access control list (ACL) to control write and execute access. Because the Global Assembly Cache is installed in a subdirectory of the systemroot directory, it inherits that directory's ACL. It is recommended that only users with Administrator privileges be allowed to delete files from the Global Assembly Cache.

Assemblies deployed in the Global Assembly Cache must have a strong name. When an assembly is added to the Global Assembly Cache, integrity checks are performed on all files that make up the assembly. The cache performs these integrity checks to ensure that an assembly has not been tampered with, for example, when a file has changed but the manifest does not reflect the change.

## See also

- Assemblies in .NET
- Working with Assemblies and the Global Assembly Cache
- Strong-Named Assemblies

# Working with Assemblies and the Global Assembly Cache

Article • 09/15/2021

> **Note**
>
> This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

If you intend to share an assembly among several applications, you can install it into the global assembly cache. Each computer where the common language runtime is installed has this machine-wide code cache. The global assembly cache stores assemblies specifically designated to be shared by several applications on the computer. An assembly must have a strong name to be installed in the global assembly cache.

> **Note**
>
> Assemblies placed in the global assembly cache must have the same assembly name and file name (not including the file name extension). For example, an assembly with the assembly name of myAssembly must have a file name of either myAssembly.exe or myAssembly.dll.

You should share assemblies by installing them into the global assembly cache only when necessary. As a general guideline, keep assembly dependencies private and locate assemblies in the application directory unless sharing an assembly is explicitly required. In addition, you do not have to install assemblies into the global assembly cache to make them accessible to COM interop or unmanaged code.

There are several reasons why you might want to install an assembly into the global assembly cache:

- Shared location.

  Assemblies that should be used by applications can be put in the global assembly cache. For example, if all applications should use an assembly located in the global assembly cache, a version policy statement can be added to the Machine.config file that redirects references to the assembly.

- File security.

Administrators often protect the systemroot directory using an Access Control List (ACL) to control write and execute access. Because the global assembly cache is installed in the systemroot directory, it inherits that directory's ACL. It is recommended that only users with Administrator privileges be allowed to delete files from the global assembly cache.

- Side-by-side versioning.

  Multiple copies of assemblies with the same name but different version information can be maintained in the global assembly cache.

- Additional search location.

  The common language runtime checks the global assembly cache for an assembly that matches the assembly request before probing or using the codebase information in a configuration file.

Note that there are scenarios where you explicitly do not want to install an assembly into the global assembly cache. If you place one of the assemblies that make up an application into the global assembly cache, you can no longer replicate or install the application by using XCOPY to copy the application directory. In this case, you must also move the assembly into the global assembly cache.

## In This Section

How to: Install an Assembly into the Global Assembly Cache Describes the ways to install an assembly into the global assembly cache.

How to: View the Contents of the Global Assembly Cache Explains how to use the Gacutil.exe (Global Assembly Cache Tool) to view the contents of the global assembly cache.

How to: Remove an Assembly from the Global Assembly Cache Explains how to use the Gacutil.exe (Global Assembly Cache Tool) to remove an assembly from the global assembly cache.

Using Serviced Components with the Global Assembly Cache Explains why serviced components (managed COM+ components) should be placed in the global assembly cache.

## Related Sections

Creating Assemblies Provides an overview of creating assemblies.

Global Assembly Cache Describes the global assembly cache.

How to: View Assembly Contents Explains how to use the Ildasm.exe (IL Disassembler) to view common intermediate language (CIL) information in an assembly.

How the Runtime Locates Assemblies Describes how the common language runtime locates and loads the assemblies that make up your application.

Programming with Assemblies Describes assemblies, the building blocks of managed applications.

# How to: View the contents of the global assembly cache

Article • 06/04/2024

> **Note**
>
> This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

Use the global assembly cache tool (gacutil.exe) to view the contents of the global assembly cache (GAC).

## View the assemblies in the GAC

To view a list of the assemblies in the global assembly cache, open Visual Studio Developer Command Prompt or Visual Studio Developer PowerShell, and then enter the following command:

```shell
gacutil -l
```

-or-

```shell
gacutil /l
```

> **Note**
>
> In earlier versions of .NET Framework, the **Shfusion.dll** Windows shell extension enabled you to view the global assembly cache in File Explorer. Beginning with .NET Framework 4, Shfusion.dll is obsolete.

## See also

- Working with Assemblies and the Global Assembly Cache

- Gacutil.exe (Global Assembly Cache Tool)

# How to: Install an assembly into the global assembly cache

Article • 06/04/2024

> Note
>
> This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

The global assembly cache (GAC) stores assemblies that several applications share. Install an assembly into the global assembly cache with one of the following components:

- Windows Installer
- Global Assembly Cache tool

> Important
>
> You can install only strong-named assemblies into the global assembly cache. For information about how to create a strong-named assembly, see **How to: Sign an assembly with a strong name**.

## Windows Installer

Windows Installer, the Windows installation engine, is the recommended way to add assemblies to the global assembly cache. Windows Installer provides reference counting of assemblies in the global assembly cache and other benefits. To create an installer package for Windows Installer, use the WiX toolset extension for Visual Studio 2017.

## Global Assembly Cache tool

You can use the .NET Global Assembly Cache utility (gacutil.exe) to add assemblies to the global assembly cache and to view the contents of the global assembly cache.

> Note

*Gacutil.exe* is for development purposes only. Don't use it to install production assemblies into the global assembly cache.

The syntax for using *gacutil.exe* to install an assembly in the GAC is as follows:

Windows Command Prompt

```
gacutil -i <assembly name>
```

In this command, *<assembly name>* is the name of the assembly to install in the global assembly cache.

If *gacutil.exe* isn't in your system path, use Visual Studio Developer Command Prompt or Visual Studio Developer PowerShell.

The following example installs an assembly with the file name *hello.dll* into the global assembly cache.

Windows Command Prompt

```
gacutil -i hello.dll
```

**Note**

In earlier versions of .NET Framework, the *Shfusion.dll* Windows shell extension let you install assemblies by dragging them to File Explorer. Beginning with .NET Framework 4, *Shfusion.dll* is obsolete.

## See also

- Work with assemblies and the global assembly cache
- How to: Remove an assembly from the global assembly cache
- Gacutil.exe (Global Assembly Cache tool)
- How to: Sign an assembly with a strong name

# How to: Remove an Assembly from the Global Assembly Cache

Article • 06/04/2024

ⓘ **Note**

This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

There are two ways to remove an assembly from the global assembly cache (GAC):

- By using the Global Assembly Cache tool (Gacutil.exe). You can use this option to uninstall assemblies that you've placed in the GAC during development and testing.
- By using Windows Installer. You should use this option to uninstall assemblies when testing installation packages and for production systems.

## Removing an assembly with Gacutil.exe

At the command prompt, type the following command:

**gacutil –u** *<assembly name>*

In this command, *assembly name* is the name of the assembly to remove from the global assembly cache.

⚠ **Warning**

You should not use Gacutil.exe to remove assemblies on production systems because of the possibility that the assembly may still be required by some application. Instead, you should use the Windows Installer, which maintains a reference count for each assembly it installs in the GAC.

The following example removes an assembly named `hello.dll` from the global assembly cache:

Console

```
gacutil -u hello
```

## Removing an assembly with Windows Installer

From the **Programs and Features** app in **Control Panel**, select the app that you want to uninstall. If the installation package placed assemblies in the GAC, Windows Installer will remove them if they are not used by another application.

> **Note**
>
> Windows Installer maintains a reference count for assemblies installed in the GAC. An assembly is removed from the GAC only when its reference count reaches zero, which indicates that it is not used by any application installed by a Windows Installer package.

## See also

- Working with Assemblies and the Global Assembly Cache
- How to: Install an Assembly into the Global Assembly Cache
- Gacutil.exe (Global Assembly Cache Tool)

# Using Serviced Components with the Global Assembly Cache

Article • 06/04/2024

ⓘ Note

This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

Serviced components (managed code COM+ components) should be put in the Global Assembly Cache. In some scenarios, the Common Language Runtime and COM+ Services can handle serviced components that are not in the Global Assembly Cache; in other scenarios, they cannot. The following scenarios illustrate this:

- For serviced components in a COM+ Server application, the assembly containing the components must be in the Global Assembly Cache, because Dllhost.exe does not run in the same directory as the one that contains the serviced components.

- For serviced components in a COM+ Library application, the runtime and COM+ Services can resolve the reference to the assembly containing the components by searching in the current directory. In this case, the assembly does not have to be in the global assembly cache.

- For serviced components in an ASP.NET application, the situation is different. If you place the assembly containing the serviced components in the bin directory of the application base and use on-demand registration, the assembly will be shadow-copied into the download cache because ASP.NET leverages the shadow capabilities of the runtime.

## See also

- Working with Assemblies and the Global Assembly Cache
- Gacutil.exe (Global Assembly Cache Tool)

# How to: Build a .NET Framework single-file assembly

Article • 06/04/2024

> Note
>
> This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

A single-file assembly, which is the simplest type of assembly, contains type information and implementation, as well as the assembly manifest. You can use command-line compilers or Visual Studio to create a single-file assembly that targets the .NET Framework. By default, the compiler creates an assembly file with an *.exe* extension.

> Note
>
> Visual Studio for C# and Visual Basic can be used only to create single-file assemblies. If you want to create multifile assemblies, you must use command-line compilers or Visual C++.

The following procedures show how to create single-file assemblies using command-line compilers.

## Create an assembly with an .exe extension

At the command prompt, type the following command:

*<compiler command> <module name>*

In this command, *compiler command* is the compiler command for the language used in your code module, and *module name* is the name of the code module to compile into the assembly.

The following example creates an assembly named *myCode.exe* from a code module called `myCode`.

C#

```
csc myCode.cs
```

## Create an assembly with an .exe extension and specify the output file name

At the command prompt, type the following command:

*<compiler command>* **/out:***<file name> <module name>*

In this command, *compiler command* is the compiler command for the language used in your code module, *file name* is the output file name, and *module name* is the name of the code module to compile into the assembly.

The following example creates an assembly named *myAssembly.exe* from a code module called `myCode`.

```C#
csc -out:myAssembly.exe myCode.cs
```

## Create library assemblies

A library assembly is similar to a class library. It contains types that will be referenced by other assemblies, but it has no entry point to begin execution.

To create a library assembly, at the command prompt, type the following command:

*<compiler command>* **-t:library** *<module name>*

In this command, *compiler command* is the compiler command for the language used in your code module, and *module name* is the name of the code module to compile into the assembly. You can also use other compiler options, such as the **-out:** option.

The following example creates a library assembly named *myCodeAssembly.dll* from a code module called `myCode`.

```C#
csc -out:myCodeLibrary.dll -t:library myCode.cs
```

## See also

- Create assemblies
- Multifile assemblies

- How to: Build a multifile assembly
- Program with assemblies

# How to: Share an assembly with other applications

Article • 06/04/2024

ⓘ **Note**

This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

Assemblies can be private or shared: by default, most simple programs consist of a private assembly because they are not intended to be used by other applications.

In order to share an assembly with other applications, it must be placed in the global assembly cache (GAC).

To share an assembly:

1. Create your assembly. For more information, see Create assemblies.

2. Assign a strong name to your assembly. For more information, see How to: Sign an assembly with a strong name.

3. Assign version information to your assembly. For more information, see Assembly versioning.

4. Add your assembly to the global assembly cache. For more information, see How to: Install an assembly into the global assembly cache.

5. Access the types contained in the assembly from other applications. For more information, see How to: Reference a strong-named assembly.

## See also

- C# programming guide
- Programming concepts (Visual Basic)
- Program with assemblies

# Multifile assemblies

Article • 06/04/2024

> **Note**
>
> This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

You can create multifile assemblies that target the .NET Framework using command-line compilers or Visual Studio with Visual C++. One file in the assembly must contain the assembly manifest. An assembly that starts an application must also contain an entry point, such as a `Main` or `WinMain` method.

For example, suppose you have an application that contains two code modules, *Client.cs* and *Stringer.cs*. *Stringer.cs* creates the `myStringer` namespace that is referenced by the code in *Client.cs*. *Client.cs* contains the `Main` method, which is the application's entry point. In this example, you compile the two code modules, and then create a third file that contains the assembly manifest, which launches the application. The assembly manifest references both the *Client* and *Stringer* modules.

> **Note**
>
> Multifile assemblies can have only one entry point, even if the assembly has multiple code modules.

There are several reasons you might want to create a multifile assembly:

- To combine modules written in different languages. This is the most common reason for creating a multifile assembly.
- To optimize downloading an application by putting seldom-used types in a module that's downloaded only when needed.
- To combine code modules written by several developers. Although each developer can compile each code module into an assembly, this can force some types to be exposed publicly that are not exposed if all modules are put into a multifile assembly.

Once you create the assembly, you can sign the file that contains the assembly manifest, and hence the assembly, or you can give the file and the assembly a strong name and put it in the global assembly cache.

## See also

- How to: Build a multifile assembly
- Program with assemblies

# How to: Build a multifile assembly

Article • 09/15/2021

> **Note**
>
> This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

This article explains how to create a multifile assembly and provides code that illustrates each step in the procedure.

> **Note**
>
> The Visual Studio IDE for C# and Visual Basic can only be used to create single-file assemblies. If you want to create multifile assemblies, you must use the command-line compilers or Visual Studio with Visual C++. Multifile assemblies are supported by .NET Framework only.

## Create a multifile assembly

1. Compile all files that contain namespaces referenced by other modules in the assembly into code modules. The default extension for code modules is *.netmodule*.

   For example, let's say the `Stringer` file has a namespace called `myStringer`, which includes a class called `Stringer`. The `Stringer` class contains a method called `StringerMethod` that writes a single line to the console.

   C#

   ```csharp
   // Assembly building example in the .NET Framework.
   using System;

   namespace myStringer
   {
       public class Stringer
       {
           public void StringerMethod()
           {
               System.Console.WriteLine("This is a line from
   StringerMethod.");
           }
   ```

```
    }
}
```

2. Use the following command to compile this code:

   C#

   ```
   csc /t:module Stringer.cs
   ```

   Specifying the *module* parameter with the **/t:** compiler option indicates that the file should be compiled as a module rather than as an assembly. The compiler produces a module called *Stringer.netmodule*, which can be added to an assembly.

3. Compile all other modules, using the necessary compiler options to indicate the other modules that are referenced in the code. This step uses the **/addmodule** compiler option.

   In the following example, a code module called *Client* has an entry point `Main` method that references a method in the *Stringer.netmodule* module created in step 1.

   C#

   ```
   using System;
   using myStringer;

   class MainClientApp
   {
       // Static method Main is the entry point method.
       public static void Main()
       {
           Stringer myStringInstance = new Stringer();
           Console.WriteLine("Client code executes");
           myStringInstance.StringerMethod();
       }
   }
   ```

4. Use the following command to compile this code:

   C#

   ```
   csc /addmodule:Stringer.netmodule /t:module Client.cs
   ```

   Specify the **/t:module** option because this module will be added to an assembly in a future step. Specify the **/addmodule** option because the code in *Client* references a namespace created by the code in *Stringer.netmodule*. The compiler produces a module called *Client.netmodule* that contains a reference to another module, *Stringer.netmodule*.

> ⓘ **Note**
>
> The C# and Visual Basic compilers support directly creating multifile assemblies using the following two different syntaxes.
>
> Two compilations create a two-file assembly:
>
> C#
>
> ```
> csc /t:module Stringer.cs
> csc Client.cs /addmodule:Stringer.netmodule
> ```
>
> One compilation creates a two-file assembly:
>
> C#
>
> ```
> csc /out:Client.exe Client.cs /out:Stringer.netmodule Stringer.cs
> ```

5. Use the Assembly Linker (Al.exe) to create the output file that contains the assembly manifest. This file contains reference information for all modules or resources that are part of the assembly.

   At the command prompt, type the following command:

   **al** <*module name*> <*module name*> … **/main:**<*method name*> **/out:**<*file name*> **/target:**<*assembly file type*>

   In this command, the *module name* arguments specify the name of each module to include in the assembly. The **/main:** option specifies the method name that is the assembly's entry point. The **/out:** option specifies the name of the output file, which contains assembly metadata. The **/target:** option specifies that the assembly is a console application executable (*.exe*) file, a Windows executable (*.win*) file, or a library (*.lib*) file.

   In the following example, *Al.exe* creates an assembly that is a console application executable called *myAssembly.exe*. The application consists of two modules called *Client.netmodule* and *Stringer.netmodule*, and the executable file called *myAssembly.exe*, which contains only assembly metadata. The entry point of the assembly is the `Main` method in the class `MainClientApp`, which is located in *Client.dll*.

Windows Command Prompt

```
al Client.netmodule Stringer.netmodule /main:MainClientApp.Main
/out:myAssembly.exe /target:exe
```

You can use IL Disassembler (Ildasm.exe) to examine the contents of an assembly, or determine whether a file is an assembly or a module.

## See also

- Create assemblies
- How to: View assembly contents
- How the runtime locates assemblies
- Multifile assemblies

# Application domains

Article • 09/15/2021

> **Note**
>
> This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

Operating systems and runtime environments typically provide some form of isolation between applications. For example, Windows uses processes to isolate applications. This isolation is necessary to ensure that code running in one application cannot adversely affect other, unrelated applications.

Application domains provide an isolation boundary for security, reliability, and versioning, and for unloading assemblies. Application domains are typically created by runtime hosts, which are responsible for bootstrapping the common language runtime before an application is run.

## The benefits of isolating applications

Historically, process boundaries have been used to isolate applications running on the same computer. Each application is loaded into a separate process, which isolates the application from other applications running on the same computer.

The applications are isolated because memory addresses are process-relative; a memory pointer passed from one process to another cannot be used in any meaningful way in the target process. In addition, you cannot make direct calls between two processes. Instead, you must use proxies, which provide a level of indirection.

Managed code must be passed through a verification process before it can be run (unless the administrator has granted permission to skip the verification). The verification process determines whether the code can attempt to access invalid memory addresses or perform some other action that could cause the process in which it is running to fail to operate properly. Code that passes the verification test is said to be type-safe. The ability to verify code as type-safe enables the common language runtime to provide as great a level of isolation as the process boundary, at a much lower performance cost.

Application domains provide a more secure and versatile unit of processing that the common language runtime can use to provide isolation between applications. You can run several application domains in a single process with the same level of isolation that would exist in separate processes, but without incurring the additional overhead of making cross-process calls or switching between processes. The ability to run multiple applications within a single process dramatically increases server scalability.

Isolating applications is also important for application security. For example, you can run controls from several Web applications in a single browser process in such a way that the controls cannot access each other's data and resources.

The isolation provided by application domains has the following benefits:

- Faults in one application cannot affect other applications. Because type-safe code cannot cause memory faults, using application domains ensures that code running in one domain cannot affect other applications in the process.

- Individual applications can be stopped without stopping the entire process. Using application domains enables you to unload the code running in a single application.

  > **Note**
  >
  > You cannot unload individual assemblies or types. Only a complete domain can be unloaded.

- Code running in one application cannot directly access code or resources from another application. The common language runtime enforces this isolation by preventing direct calls between objects in different application domains. Objects that pass between domains are either copied or accessed by proxy. If the object is copied, the call to the object is local. That is, both the caller and the object being referenced are in the same application domain. If the object is accessed through a proxy, the call to the object is remote. In this case, the caller and the object being referenced are in different application domains. Cross-domain calls use the same remote call infrastructure as calls between two processes or between two machines. As such, the metadata for the object being referenced must be available to both application domains to allow the method call to be JIT-compiled properly. If the calling domain does not have access to the metadata for the object being called, the compilation might fail with an exception of type FileNotFoundException. For more information, see Remote Objects. The mechanism for determining how objects can be accessed across domains is determined by the object. For more information, see System.MarshalByRefObject.

- The behavior of code is scoped by the application in which it runs. In other words, the application domain provides configuration settings such as application version policies, the location of any remote assemblies it accesses, and information about where to locate assemblies that are loaded into the domain.

- Permissions granted to code can be controlled by the application domain in which the code is running.

## Application domains and assemblies

This section describes the relationship between application domains and assemblies. You must load an assembly into an application domain before you can execute the code it contains. Running a typical application causes several assemblies to be loaded into an application domain.

The way an assembly is loaded determines whether its just-in-time (JIT) compiled code can be shared by multiple application domains in the process, and whether the assembly can be unloaded from the process.

- If an assembly is loaded domain-neutral, all application domains that share the same security grant set can share the same JIT-compiled code, which reduces the memory required by the application. However, the assembly can never be unloaded from the process.

- If an assembly is not loaded domain-neutral, it must be JIT-compiled in every application domain in which it is loaded. However, the assembly can be unloaded from the process by unloading all the application domains in which it is loaded.

The runtime host determines whether to load assemblies as domain-neutral when it loads the runtime into a process. For managed applications, apply the LoaderOptimizationAttribute attribute to the entry-point method for the process, and specify a value from the associated LoaderOptimization enumeration. For unmanaged applications that host the common language runtime, specify the appropriate flag when you call the CorBindToRuntimeEx Function method.

There are three options for loading domain-neutral assemblies:

- LoaderOptimization.SingleDomain loads no assemblies as domain-neutral, except Mscorlib, which is always loaded domain-neutral. This setting is called single domain because it is commonly used when the host is running only a single application in the process.

- LoaderOptimization.MultiDomain loads all assemblies as domain-neutral. Use this setting when there are multiple application domains in the process, all of which run the same code.

- LoaderOptimization.MultiDomainHost loads strong-named assemblies as domain-neutral, if they and all their dependencies have been installed in the global assembly cache. Other assemblies are loaded and JIT-compiled separately for each application domain in which they are loaded, and thus can be unloaded from the process. Use this setting when running more than one application in the same process, or if you have a mixture of assemblies that are shared by many application domains and assemblies that need to be unloaded from the process.

JIT-compiled code cannot be shared for assemblies loaded into the load-from context, using the LoadFrom method of the Assembly class, or loaded from images using overloads of the Load method that specify byte arrays.

Assemblies that have been compiled to native code by using the Ngen.exe (Native Image Generator) can be shared between application domains, if they are loaded domain-neutral the first time they are loaded into a process.

JIT-compiled code for the assembly that contains the application entry point is shared only if all its dependencies can be shared.

A domain-neutral assembly can be JIT-compiled more than once. For example, when the security grant sets of two application domains are different, they cannot share the same JIT-compiled code. However, each copy of the JIT-compiled assembly can be shared with other application domains that have the same grant set.

When you decide whether to load assemblies as domain-neutral, you must make a tradeoff between reducing memory use and other performance factors.

- Access to static data and methods is slower for domain-neutral assemblies because of the need to isolate assemblies. Each application domain that accesses the assembly must have a separate copy of the static data, to prevent references to objects in static fields from crossing domain boundaries. As a result, the runtime contains additional logic to direct a caller to the appropriate copy of the static data or method. This extra logic slows down the call.

- All the dependencies of an assembly must be located and loaded when the assembly is loaded domain-neutral, because a dependency that cannot be loaded domain-neutral prevents the assembly from being loaded domain-neutral.

## Application domains and threads

An application domain forms an isolation boundary for security, versioning, reliability, and unloading of managed code. A thread is the operating system construct used by the common language runtime to execute code. At run time, all managed code is loaded into an application domain and is run by one or more managed threads.

There is not a one-to-one correlation between application domains and threads. Several threads can execute in a single application domain at any given time, and a particular thread is not confined to a single application domain. That is, threads are free to cross application domain boundaries; a new thread is not created for each application domain.

At any given time, every thread executes in an application domain. Zero, one, or multiple threads might be executing in any given application domain. The runtime keeps track of which threads are running in which application domains. You can locate the domain in which a thread is executing at any time by calling the Thread.GetDomain method.

### Application domains and cultures

Culture, which is represented by a CultureInfo object, is associated with threads. You can get the culture that is associated with the currently executing thread by using the CultureInfo.CurrentCulture property, and you can get or set the culture that is associated with the currently executing thread by using the Thread.CurrentCulture property. If the culture that is associated with a thread has been explicitly set by using the Thread.CurrentCulture property, it continues to be associated with that thread when the thread crosses application domain boundaries. Otherwise, the culture that is associated with the thread at any given time is determined by the value of the CultureInfo.DefaultThreadCurrentCulture property in the application domain in which the thread is executing:

- If the value of the property is not `null`, the culture that is returned by the property is associated with the thread (and therefore returned by the Thread.CurrentCulture and CultureInfo.CurrentCulture properties).

- If the value of the property is `null`, the current system culture is associated with the thread.

## Programming with application domains

Application domains are usually created and manipulated programmatically by runtime hosts. However, sometimes an application program might also want to work with application domains. For example, an application program could load an application component into a domain to be able to unload the domain (and the component) without having to stop the entire application.

The AppDomain is the programmatic interface to application domains. This class includes methods to create and unload domains, to create instances of types in domains, and to register for various notifications such as application domain unloading. The following table lists commonly used AppDomain methods.

Expand table

| AppDomain Method | Description |
|---|---|
| CreateDomain | Creates a new application domain. It is recommended that you use an overload of this method that specifies an AppDomainSetup object. This is the preferred way to set the properties of a new domain, such as the application base, or root directory for the application; the location of the configuration file for the domain; and the search path that the common language runtime is to use to load assemblies into the domain. |
| ExecuteAssembly and ExecuteAssemblyByName | Executes an assembly in the application domain. This is an instance method, so it can be used to execute code in another application domain to which you have a reference. |
| CreateInstanceAndUnwrap | Creates an instance of a specified type in the application domain, and returns a proxy. Use this method to avoid loading the assembly containing the created type into the calling assembly. |
| Unload | Performs a graceful shutdown of the domain. The application domain is not unloaded until all threads running in the domain have either stopped or are no longer in the domain. |

**Note**

The common language runtime does not support serialization of global methods, so delegates cannot be used to execute global methods in other application domains.

The unmanaged interfaces described in the common language runtime Hosting Interfaces Specification also provide access to application domains. Runtime hosts can use interfaces from unmanaged code to create and gain access to the application domains within a process.

# The COMPLUS_LoaderOptimization environment variable

An environment variable that sets the default loader optimization policy of an executable application.

## Syntax

```env
COMPLUS_LoaderOptimization = 1
```

## Remarks

A typical application loads several assemblies into an application domain before the code they contain can be executed.

The way the assembly is loaded determines whether its just-in-time (JIT) compiled code can be shared by multiple application domains in the process.

- If an assembly is loaded domain-neutral, all application domains that share the same security grant set can share the same JIT-compiled code. This reduces the memory required by the application.
- If an assembly is not loaded domain-neutral, it must be JIT-compiled in every application domain in which it is loaded and the loader must not share internal resources across application domains.

When set to 1, the COMPLUS_LoaderOptimization environment flag forces the runtime host to load all assemblies in non-domain-neutral way known as SingleDomain. SingleDomain loads no assemblies as domain-neutral, except Mscorlib, which is always loaded domain-neutral. This setting is called single domain because it is commonly used when the host is running only a single application in the process.

**Caution**

The COMPLUS_LoaderOptimization environment flag was designed to be used in diagnostic and test scenarios. Having the flag turned on can cause severe slow-down and increase in memory usage.

## Code example

To force all assemblies not to be loaded as domain-neutral for the IISADMIN service can be achieved by appending `COMPLUS_LoaderOptimization=1` to the Environment's Multi-String Value in the HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\services\IISADMIN key.

```env
Key = HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\services\IISADMIN
Name = Environment
Type = REG_MULTI_SZ
Value (to append) = COMPLUS_LoaderOptimization=1
```

## See also

- System.AppDomain
- System.MarshalByRefObject
- Programming with Application Domains and Assemblies
- Using Application Domains

# Using Application Domains

Article • 06/04/2024

**Note**

This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

Application domains provide a unit of isolation for the common language runtime. They are created and run inside a process. Application domains are usually created by a runtime host, which is an application responsible for loading the runtime into a process and executing user code within an application domain. The runtime host creates a process and a default application domain, and runs managed code inside it. Runtime hosts include ASP.NET and the Windows shell.

For most applications, you do not need to create your own application domain; the runtime host creates any necessary application domains for you. However, you can create and configure additional application domains if your application needs to isolate code or to use and unload DLLs.

## In This Section

How to: Create an Application Domain Describes how to programmatically create an application domain.

How to: Unload an Application Domain Describes how to programmatically unload an application domain.

How to: Configure an Application Domain Provides an introduction to configuring an application domain.

Retrieving Setup Information from an Application Domain Describes how to retrieve setup information from an application domain.

How to: Load Assemblies into an Application Domain Describes how to load an assembly into an application domain.

How to: Obtain Type and Member Information from an Assembly Describes how to retrieve information about an assembly.

Shadow Copying Assemblies Describes how shadow copying allows updates to assemblies while they are in use, and how to configure shadow copying.

How to: Receive First-Chance Exception Notifications Explains how you can receive a notification that an exception has been thrown, before the common language runtime has begun searching for exception handlers.

Resolving Assembly Loads Provides guidance on using the AppDomain.AssemblyResolve event to resolve assembly load failures.

## Reference

AppDomain Represents an application domain. Provides methods for creating and controlling application domains.

## Related Sections

Assemblies in .NET Provides an overview of the functions performed by assemblies.

Programming with Assemblies Describes how to create, sign, and set attributes on assemblies.

Emitting Dynamic Methods and Assemblies Describes how to create dynamic assemblies.

Application Domains Provides a conceptual overview of application domains.

Reflection Overview Describes how to use the **Reflection** class to obtain information about an assembly.

# How to: Create an Application Domain

Article • 06/04/2024

ⓘ **Note**

This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

A common language runtime host creates application domains automatically when they are needed. However, you can create your own application domains and load into them those assemblies that you want to manage personally. You can also create application domains from which you execute code.

You create a new application domain using one of the overloaded **CreateDomain** methods in the System.AppDomain class. You can give the application domain a name and reference it by that name.

The following example creates a new application domain, assigns it the name `MyDomain`, and then prints the name of the host domain and the newly created child application domain to the console.

## Example

C#

```csharp
using System;
using System.Reflection;

class AppDomain1
{
    public static void Main()
    {
        Console.WriteLine("Creating new AppDomain.");
        AppDomain domain = AppDomain.CreateDomain("MyDomain");

        Console.WriteLine("Host domain: " +
AppDomain.CurrentDomain.FriendlyName);
        Console.WriteLine("child domain: " + domain.FriendlyName);
    }
}
```

## See also

- Programming with Application Domains
- Using Application Domains

# How to: Unload an Application Domain

Article • 06/04/2024

> **Note**
>
> This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

When you have finished using an application domain, unload it using the AppDomain.Unload method. The **Unload** method gracefully shuts down the specified application domain. During the unloading process, no new threads can access the application domain, and all application domain–specific data structures are freed.

Assemblies loaded into the application domain are removed and are no longer available. If an assembly in the application domain is domain-neutral, data for the assembly remains in memory until the entire process is shut down. There is no mechanism to unload a domain-neutral assembly other than shutting down the entire process. There are situations where the request to unload an application domain does not work and results in a CannotUnloadAppDomainException.

The following example creates a new application domain called `MyDomain`, prints some information to the console, and then unloads the application domain. Note that the code then attempts to print the friendly name of the unloaded application domain to the console. This action generates an exception that is handled by the try/catch statements at the end of the program.

## Example

C#

```csharp
using System;
using System.Reflection;

class AppDomain2
{
    public static void Main()
    {
        Console.WriteLine("Creating new AppDomain.");
        AppDomain domain = AppDomain.CreateDomain("MyDomain", null);

        Console.WriteLine("Host domain: " +
AppDomain.CurrentDomain.FriendlyName);
        Console.WriteLine("child domain: " + domain.FriendlyName);
        try
        {
            AppDomain.Unload(domain);
            Console.WriteLine();
            Console.WriteLine("Host domain: " +
AppDomain.CurrentDomain.FriendlyName);
            // The following statement creates an exception because the
domain no longer exists.
            Console.WriteLine("child domain: " + domain.FriendlyName);
        }
        catch (AppDomainUnloadedException e)
        {
            Console.WriteLine(e.GetType().FullName);
            Console.WriteLine("The appdomain MyDomain does not exist.");
        }
    }
}
```

## See also

- Programming with Application Domains
- How to: Create an Application Domain
- Using Application Domains

# How to: Configure an Application Domain

Article • 06/04/2024

> **Note**
>
> This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

You can provide the common language runtime with configuration information for a new application domain using the AppDomainSetup class. When creating your own application domains, the most important property is ApplicationBase. The other **AppDomainSetup** properties are used mainly by runtime hosts to configure a particular application domain.

The **ApplicationBase** property defines the root directory of the application. When the runtime needs to satisfy a type request, it probes for the assembly containing the type in the directory specified by the **ApplicationBase** property.

> **Note**
>
> A new application domain inherits only the **ApplicationBase** property of the creator.

The following example creates an instance of the **AppDomainSetup** class, uses this class to create a new application domain, writes the information to console, and then unloads the application domain.

## Example

C#

```
using System;
using System.Reflection;

class AppDomain4
{
    public static void Main()
    {
        // Create application domain setup information.
        AppDomainSetup domaininfo = new AppDomainSetup();
        domaininfo.ApplicationBase = "f:\\work\\development\\latest";

        // Create the application domain.
        AppDomain domain = AppDomain.CreateDomain("MyDomain", null,
domaininfo);

        // Write application domain information to the console.
        Console.WriteLine("Host domain: " +
AppDomain.CurrentDomain.FriendlyName);
        Console.WriteLine("child domain: " + domain.FriendlyName);
        Console.WriteLine("Application base is: " +
domain.SetupInformation.ApplicationBase);

        // Unload the application domain.
        AppDomain.Unload(domain);
    }
}
```

## See also

- Programming with Application Domains
- Using Application Domains

# Retrieve setup information from an application domain

Article • 06/04/2024

> **Note**
>
> This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

Each instance of an application domain consists of both properties and AppDomainSetup information. You can retrieve setup information from an application domain using the System.AppDomain class. This class provides several members that retrieve configuration information about an application domain.

You can also query the **AppDomainSetup** object for the application domain to obtain setup information that was passed to the domain when it was created.

The following example creates a new application domain and then prints several member values to the console.

C#

```
using System;
using System.Reflection;

class AppDomain3
{
    public static void Main()
    {
        // Create the new application domain.
        AppDomain domain = AppDomain.CreateDomain("MyDomain", null);

        // Output to the console.
        Console.WriteLine("Host domain: " +
AppDomain.CurrentDomain.FriendlyName);
        Console.WriteLine("New domain: " + domain.FriendlyName);
        Console.WriteLine("Application base is: " + domain.BaseDirectory);
        Console.WriteLine("Relative search path is: " +
domain.RelativeSearchPath);
        Console.WriteLine("Shadow copy files is set to: " +
domain.ShadowCopyFiles);
        AppDomain.Unload(domain);
    }
}
```

The following example sets, and then retrieves, setup information for an application domain. `AppDomain.SetupInformation.ApplicationBase` gets the configuration information.

C#

```
using System;
using System.Reflection;

class AppDomain5
{
    public static void Main()
    {
        // Application domain setup information.
        AppDomainSetup domaininfo = new AppDomainSetup();
        domaininfo.ApplicationBase = "f:\\work\\development\\latest";
        domaininfo.ConfigurationFile =
"f:\\work\\development\\latest\\appdomain5.exe.config";

        // Creates the application domain.
        AppDomain domain = AppDomain.CreateDomain("MyDomain", null,
domaininfo);

        // Write the application domain information to the console.
        Console.WriteLine("Host domain: " +
AppDomain.CurrentDomain.FriendlyName);
        Console.WriteLine("Child domain: " + domain.FriendlyName);
        Console.WriteLine();
        Console.WriteLine("Application base is: " +
domain.SetupInformation.ApplicationBase);
        Console.WriteLine("Configuration file is: " +
domain.SetupInformation.ConfigurationFile);

        // Unloads the application domain.
        AppDomain.Unload(domain);
    }
}
```

## See also

- Programming with Application Domains
- Using Application Domains

# How to: Load Assemblies into an Application Domain

Article • 06/04/2024

> ⓘ **Note**
>
> This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

There are several ways to load an assembly into an application domain. The recommended way is to use the `static` (`Shared` in Visual Basic) Load method of the System.Reflection.Assembly class. Other ways assemblies can be loaded include:

- The LoadFrom method of the Assembly class loads an assembly given its file location. Loading assemblies with this method uses a different load context.

- The ReflectionOnlyLoad and ReflectionOnlyLoadFrom methods load an assembly into the reflection-only context. Assemblies loaded into this context can be examined but not executed, allowing the examination of assemblies that target other platforms. See How to: Load Assemblies into the Reflection-Only Context.

> ⓘ **Note**
>
> The reflection-only context is new in .NET Framework version 2.0.

- Methods such as CreateInstance and CreateInstanceAndUnwrap of the AppDomain class can load assemblies into an application domain.

- The GetType method of the Type class can load assemblies.

- The Load method of the System.AppDomain class can load assemblies, but is primarily used for COM interoperability. It should not be used to load assemblies into an application domain other than the application domain from which it is called.

> ⓘ **Note**
>
> Starting with .NET Framework version 2.0, the runtime will not load an assembly that was compiled with a version of the .NET Framework that has a higher version

number than the currently loaded runtime. This applies to the combination of the major and minor components of the version number.

You can specify the way the just-in-time (JIT) compiled code from loaded assemblies is shared between application domains. For more information, see Application domains and assemblies.

## Example

The following code loads an assembly named "example.exe" or "example.dll" into the current application domain, gets a type named `Example` from the assembly, gets a parameterless method named `MethodA` for that type, and executes the method. For a complete discussion on obtaining information from a loaded assembly, see Dynamically Loading and Using Types.

C#

```csharp
using System;
using System.Reflection;

public class Asmload0
{
    public static void Main()
    {
        // Use the file name to load the assembly into the current
        // application domain.
        Assembly a = Assembly.Load("example");
        // Get the type to use.
        Type myType = a.GetType("Example");
        // Get the method to call.
        MethodInfo myMethod = myType.GetMethod("MethodA");
        // Create an instance.
        object obj = Activator.CreateInstance(myType);
        // Execute the method.
        myMethod.Invoke(obj, null);
    }
}
```

## See also

- ReflectionOnlyLoad
- Programming with Application Domains
- Reflection
- Using Application Domains

- How to: Load Assemblies into the Reflection-Only Context
- Application domains and assemblies

# Shadow Copying Assemblies

Article • 07/23/2022

**Note**

This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

Shadow copying enables assemblies that are used in an application domain to be updated without unloading the application domain. This is particularly useful for applications that must be available continuously, such as ASP.NET sites.

**Important**

Shadow copying is not supported in Windows 8.x Store apps.

The common language runtime locks an assembly file when the assembly is loaded, so the file cannot be updated until the assembly is unloaded. The only way to unload an assembly from an application domain is by unloading the application domain, so under normal circumstances, an assembly cannot be updated on disk until all the application domains that are using it have been unloaded.

When an application domain is configured to shadow copy files, assemblies from the application path are copied to another location and loaded from that location. The copy is locked, but the original assembly file is unlocked and can be updated.

**Important**

The only assemblies that can be shadow copied are those stored in the application directory or its subdirectories, specified by the ApplicationBase and PrivateBinPath properties when the application domain is configured. Assemblies stored in the global assembly cache are not shadow copied.

This article contains the following sections:

- Enabling and Using Shadow Copying describes the basic use and the options that are available for shadow copying.
- Startup Performance describes the changes that are made to shadow copying in the .NET Framework 4 to improve startup performance, and how to revert to the

behavior of earlier versions.

- Obsolete Methods describes the changes that were made to the properties and methods that control shadow copying in the .NET Framework 2.0.

## Enabling and Using Shadow Copying

You can use the properties of the AppDomainSetup class as follows to configure an application domain for shadow copying:

- Enable shadow copying by setting the ShadowCopyFiles property to the string value `"true"`.

  By default, this setting causes all assemblies in the application path to be copied to a download cache before they are loaded. This is the same cache maintained by the common language runtime to store files downloaded from other computers, and the common language runtime automatically deletes the files when they are no longer needed.

- Optionally set a custom location for shadow copied files by using the CachePath property and the ApplicationName property.

  The base path for the location is formed by concatenating the ApplicationName property to the CachePath property as a subdirectory. Assemblies are shadow copied to subdirectories of this path, not to the base path itself.

  > **Note**
  >
  > If the ApplicationName property is not set, the CachePath property is ignored and the download cache is used. No exception is thrown.

  If you specify a custom location, you are responsible for cleaning up the directories and copied files when they are no longer needed. They are not deleted automatically.

  There are a few reasons why you might want to set a custom location for shadow copied files. You might want to set a custom location for shadow copied files if your application generates a large number of copies. The download cache is limited by size, not by lifetime, so it is possible that the common language runtime will attempt to delete a file that is still in use. Another reason to set a custom location is when users running your application do not have write access to the directory location the common language runtime uses for the download cache.

- Optionally limit the assemblies that are shadow copied by using the ShadowCopyDirectories property.

  When you enable shadow copying for an application domain, the default is to copy all assemblies in the application path — that is, in the directories specified by the ApplicationBase and PrivateBinPath properties. You can limit the copying to selected directories by creating a string that contains only those directories you want to shadow copy, and assigning the string to the ShadowCopyDirectories property. Separate the directories with semicolons. The only assemblies that are shadow copied are the ones in the selected directories.

  > **Note**
  >
  > If you don't assign a string to the ShadowCopyDirectories property, or if you set this property to `null`, all assemblies in the directories specified by the ApplicationBase and PrivateBinPath properties are shadow copied.

  > **Important**
  >
  > Directory paths must not contain semicolons, because the semicolon is the delimiter character. There is no escape character for semicolons.

## Startup Performance

When an application domain that uses shadow copying starts, there is a delay while assemblies in the application directory are copied to the shadow copy directory, or verified if they are already in that location. Before the .NET Framework 4, all assemblies were copied to a temporary directory. Each assembly was opened to verify the assembly name, and the strong name was validated. Each assembly was checked to see whether it had been updated more recently than the copy in the shadow copy directory. If so, it was copied to the shadow copy directory. Finally, the temporary copies were discarded.

Beginning with the .NET Framework 4, the default startup behavior is to directly compare the file date and time of each assembly in the application directory with the file date and time of the copy in the shadow copy directory. If the assembly has been updated, it is copied by using the same procedure as in earlier versions of the .NET Framework; otherwise, the copy in the shadow copy directory is loaded.

The resulting performance improvement is largest for applications in which assemblies do not change frequently and changes usually occur in a small subset of assemblies. If a majority of assemblies in an application change frequently, the new default behavior might cause a performance regression. You can restore the startup behavior of previous versions of the .NET Framework by adding the <shadowCopyVerifyByTimestamp> element to the configuration file, with `enabled="false"`.

## Obsolete Methods

The AppDomain class has several methods, such as SetShadowCopyFiles and ClearShadowCopyPath, that can be used to control shadow copying on an application domain, but these have been marked obsolete in .NET Framework version 2.0. The recommended way to configure an application domain for shadow copying is to use the properties of the AppDomainSetup class.

## See also

- AppDomainSetup.ShadowCopyFiles
- AppDomainSetup.CachePath
- AppDomainSetup.ApplicationName
- AppDomainSetup.ShadowCopyDirectories
- <shadowCopyVerifyByTimestamp> Element

# How to: Receive First-Chance Exception Notifications

Article • 09/15/2021

ⓘ Note

This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

The FirstChanceException event of the AppDomain class lets you receive a notification that an exception has been thrown, before the common language runtime has begun searching for exception handlers.

The event is raised at the application domain level. A thread of execution can pass through multiple application domains, so an exception that is unhandled in one application domain could be handled in another application domain. The notification occurs in each application domain that has added a handler for the event, until an application domain handles the exception.

The procedures and examples in this article show how to receive first-chance exception notifications in a simple program that has one application domain, and in an application domain that you create.

For a more complex example that spans several application domains, see the example for the FirstChanceException event.

## Receiving First-Chance Exception Notifications in the Default Application Domain

In the following procedure, the entry point for the application, the `Main()` method, runs in the default application domain.

### To demonstrate first-chance exception notifications in the default application domain

1. Define an event handler for the FirstChanceException event, using a lambda function, and attach it to the event. In this example, the event handler prints the

name of the application domain where the event was handled and the exception's Message property.

```C#
using System;
using System.Runtime.ExceptionServices;

class Example
{
    static void Main()
    {
        AppDomain.CurrentDomain.FirstChanceException +=
            (object source, FirstChanceExceptionEventArgs e) =>
            {
                Console.WriteLine("FirstChanceException event raised in
{0}: {1}",
                    AppDomain.CurrentDomain.FriendlyName,
e.Exception.Message);
            };
```

2. Throw an exception and catch it. Before the runtime locates the exception handler, the FirstChanceException event is raised and displays a message. This message is followed by the message that is displayed by the exception handler.

```C#
try
{
    throw new ArgumentException("Thrown in " +
AppDomain.CurrentDomain.FriendlyName);
}
catch (ArgumentException ex)
{
    Console.WriteLine("ArgumentException caught in {0}: {1}",
        AppDomain.CurrentDomain.FriendlyName, ex.Message);
}
```

3. Throw an exception, but do not catch it. Before the runtime looks for an exception handler, the FirstChanceException event is raised and displays a message. There is no exception handler, so the application terminates.

```C#
        throw new ArgumentException("Thrown in " +
AppDomain.CurrentDomain.FriendlyName);
    }
}
```

The code that is shown in the first three steps of this procedure forms a complete console application. The output from the application varies, depending on the name of the .exe file, because the name of the default application domain consists of the name and extension of the .exe file. See the following for sample output.

```C#
/* This example produces output similar to the following:

FirstChanceException event raised in Example.exe: Thrown in Example.exe
ArgumentException caught in Example.exe: Thrown in Example.exe
FirstChanceException event raised in Example.exe: Thrown in Example.exe

Unhandled Exception: System.ArgumentException: Thrown in Example.exe
   at Example.Main()
 */
```

## Receiving First-Chance Exception Notifications in Another Application Domain

If your program contains more than one application domain, you can choose which application domains receive notifications.

### To receive first-chance exception notifications in an application domain that you create

1. Define an event handler for the FirstChanceException event. This example uses a `static` method (`Shared` method in Visual Basic) that prints the name of the application domain where the event was handled and the exception's Message property.

   ```C#
   static void FirstChanceHandler(object source,
   FirstChanceExceptionEventArgs e)
   {
       Console.WriteLine("FirstChanceException event raised in {0}: {1}",
           AppDomain.CurrentDomain.FriendlyName, e.Exception.Message);
   }
   ```

2. Create an application domain and add the event handler to the FirstChanceException event for that application domain. In this example, the application domain is named `AD1`.

C#

```
AppDomain ad = AppDomain.CreateDomain("AD1");
ad.FirstChanceException += FirstChanceHandler;
```

You can handle this event in the default application domain in the same way. Use the `static` (`Shared` in Visual Basic) AppDomain.CurrentDomain property in `Main()` to get a reference to the default application domain.

## To demonstrate first-chance exception notifications in the application domain

1. Create a `Worker` object in the application domain that you created in the previous procedure. The `Worker` class must be public, and must derive from MarshalByRefObject, as shown in the complete example at the end of this article.

C#

```
Worker w = (Worker) ad.CreateInstanceAndUnwrap(
                        typeof(Worker).Assembly.FullName, "Worker");
```

2. Call a method of the `Worker` object that throws an exception. In this example, the `Thrower` method is called twice. The first time, the method argument is `true`, which causes the method to catch its own exception. The second time, the argument is `false`, and the `Main()` method catches the exception in the default application domain.

C#

```
// The worker throws an exception and catches it.
w.Thrower(true);

try
{
    // The worker throws an exception and doesn't catch it.
    w.Thrower(false);
}
catch (ArgumentException ex)
{
    Console.WriteLine("ArgumentException caught in {0}: {1}",
        AppDomain.CurrentDomain.FriendlyName, ex.Message);
}
```

3. Place code in the `Thrower` method to control whether the method handles its own exception.

   C#

   ```
   if (catchException)
   {
       try
       {
           throw new ArgumentException("Thrown in " +
   AppDomain.CurrentDomain.FriendlyName);
       }
       catch (ArgumentException ex)
       {
           Console.WriteLine("ArgumentException caught in {0}: {1}",
               AppDomain.CurrentDomain.FriendlyName, ex.Message);
       }
   }
   else
   {
       throw new ArgumentException("Thrown in " +
   AppDomain.CurrentDomain.FriendlyName);
   }
   ```

## Example

The following example creates an application domain named `AD1` and adds an event handler to the application domain's FirstChanceException event. The example creates an instance of the `Worker` class in the application domain, and calls a method named `Thrower` that throws an ArgumentException. Depending on the value of its argument, the method either catches the exception or fails to handle it.

Each time the `Thrower` method throws an exception in `AD1`, the FirstChanceException event is raised in `AD1`, and the event handler displays a message. The runtime then looks for an exception handler. In the first case, the exception handler is found in `AD1`. In the second case, the exception is unhandled in `AD1`, and instead is caught in the default application domain.

**Note**

The name of the default application domain is the same as the name of the executable.

If you add a handler for the FirstChanceException event to the default application domain, the event is raised and handled before the default application domain handles the exception. To see this, add the C# code `AppDomain.CurrentDomain.FirstChanceException += FirstChanceException;` (in Visual Basic, `AddHandler AppDomain.CurrentDomain.FirstChanceException, FirstChanceException`) at the beginning of `Main()`.

C#

```csharp
using System;
using System.Reflection;
using System.Runtime.ExceptionServices;

class Example
{
    static void Main()
    {
        // To receive first chance notifications of exceptions in
        // an application domain, handle the FirstChanceException
        // event in that application domain.
        AppDomain ad = AppDomain.CreateDomain("AD1");
        ad.FirstChanceException += FirstChanceHandler;

        // Create a worker object in the application domain.
        Worker w = (Worker) ad.CreateInstanceAndUnwrap(
                                typeof(Worker).Assembly.FullName, "Worker");

        // The worker throws an exception and catches it.
        w.Thrower(true);

        try
        {
            // The worker throws an exception and doesn't catch it.
            w.Thrower(false);
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine("ArgumentException caught in {0}: {1}",
                AppDomain.CurrentDomain.FriendlyName, ex.Message);
        }
    }

    static void FirstChanceHandler(object source,
FirstChanceExceptionEventArgs e)
    {
        Console.WriteLine("FirstChanceException event raised in {0}: {1}",
            AppDomain.CurrentDomain.FriendlyName, e.Exception.Message);
    }
}

public class Worker : MarshalByRefObject
{
    public void Thrower(bool catchException)
    {
        if (catchException)
        {
            try
            {
                throw new ArgumentException("Thrown in " +
AppDomain.CurrentDomain.FriendlyName);
            }
            catch (ArgumentException ex)
            {
                Console.WriteLine("ArgumentException caught in {0}: {1}",
                    AppDomain.CurrentDomain.FriendlyName, ex.Message);
            }
        }
        else
        {
            throw new ArgumentException("Thrown in " +
AppDomain.CurrentDomain.FriendlyName);
        }
    }
}

/* This example produces output similar to the following:

FirstChanceException event raised in AD1: Thrown in AD1
ArgumentException caught in AD1: Thrown in AD1
FirstChanceException event raised in AD1: Thrown in AD1
ArgumentException caught in Example.exe: Thrown in AD1
 */
```

## See also

- FirstChanceException

# System.AppDomain.UnhandledException event

Article • 08/05/2024

This article provides supplementary remarks to the reference documentation for this API.

The UnhandledException event provides notification of uncaught exceptions. It allows the application to log information about the exception before the system default handler reports the exception to the user and terminates the application. If sufficient information about the state of the application is available, other actions may be undertaken - such as saving program data for later recovery. Caution is advised, because program data can become corrupted when exceptions are not handled. The handler will also be running while holding locks held when the exception was thrown, so care should be taken to avoid waiting on other resources which could introduce deadlocks.

This event can be handled in any application domain. However, the event is not necessarily raised in the application domain where the exception occurred. An exception is unhandled only if the entire stack for the thread has been unwound without finding an applicable exception handler, so the first place the event can be raised is in the application domain where the thread originated.

If the UnhandledException event is handled in the default application domain, it is raised there for any unhandled exception in any thread, no matter what application domain the thread started in. If the thread started in an application domain that has an event handler for UnhandledException, the event is raised in that application domain. If that application domain is not the default application domain, and there is also an event handler in the default application domain, the event is raised in both application domains.

For example, suppose a thread starts in application domain "AD1", calls a method in application domain "AD2", and from there calls a method in application domain "AD3", where it throws an exception. The first application domain in which the UnhandledException event can be raised is "AD1". If that application domain is not the default application domain, the event can also be raised in the default application domain.

**Note**

The common language runtime suspends thread aborts while event handlers for the **UnhandledException** event are executing.

If the event handler has a ReliabilityContractAttribute attribute with the appropriate flags, the event handler is treated as a constrained execution region.

Starting with .NET Framework 4, this event is not raised for exceptions that corrupt the state of the process, such as stack overflows or access violations, unless the event handler is security-critical and has the HandleProcessCorruptedStateExceptionsAttribute attribute.

To register an event handler for this event, you must have the required permissions, or a SecurityException is thrown.

For more information about handling events, see Handling and Raising Events.

## Other events for unhandled exceptions

For certain application models, the UnhandledException event can be preempted by other events if the unhandled exception occurs in the main application thread.

In applications that use Windows Forms, unhandled exceptions in the main application thread cause the Application.ThreadException event to be raised. If this event is handled, the default behavior is that the unhandled exception does not terminate the application, although the application is left in an unknown state. In that case, the UnhandledException event is not raised. This behavior can be changed by using the application configuration file, or by using the Application.SetUnhandledExceptionMode method to change the mode to UnhandledExceptionMode.ThrowException before the ThreadException event handler is hooked up. This applies only to the main application thread. The UnhandledException event is raised for unhandled exceptions thrown in other threads.

The Visual Basic application framework provides another event for unhandled exceptions in the main application thread—the WindowsFormsApplicationBase.UnhandledException event. This event has an event arguments object with the same name as the event arguments object used by AppDomain.UnhandledException, but with different properties. In particular, this event arguments object has an ExitApplication property that allows the application to continue running, ignoring the unhandled exception (and leaving the application in an unknown state). In that case, the AppDomain.UnhandledException event is not raised.

# Running Intranet Applications in Full Trust

Article • 06/04/2024

> **Note**
>
> This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

Starting with the .NET Framework version 3.5 Service Pack 1 (SP1), applications and their library assemblies can be run as full-trust assemblies from a network share. MyComputer zone evidence is automatically added to assemblies that are loaded from a share on the intranet. This evidence gives those assemblies the same grant set (which is typically full trust) as the assemblies that reside on the computer. This functionality does not apply to ClickOnce applications or to applications that are designed to run on a host.

## Rules for Library Assemblies

The following rules apply to assemblies that are loaded by an executable on a network share:

- Library assemblies must reside in the same folder as the executable assembly. Assemblies that reside in a subfolder or are referenced on a different path are not given the full-trust grant set.
- If the executable delay-loads an assembly, it must use the same path that was used to start the executable. For example, if the share \*network-computer*\*share* is mapped to a drive letter and the executable is run from that path, assemblies that are loaded by the executable by using the network path will not be granted full trust. To delay-load an assembly in the MyComputer zone, the executable must use the drive letter path.

## Restoring the Former Intranet Policy

In earlier versions of the .NET Framework, shared assemblies were granted Intranet zone evidence. You had to specify code access security policy to grant full trust to an assembly on a share.

This new behavior is the default for intranet assemblies. You can return to the earlier behavior of providing Intranet evidence by setting a registry key that applies to all applications on the computer. This process is different for 32-bit and 64-bit computers:

- On 32-bit computers, create a subkey under the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework key in the system registry. Use the key name LegacyMyComputerZone with a DWORD value of 1.
- On 64-bit computers, create a subkey under the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework key in the system registry. Use the key name LegacyMyComputerZone with a DWORD value of 1. Create the same subkey under the HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\.NETFramework key.

## See also

- Programming with Assemblies