CHAPTER 22

CLR Hosting and AppDomains

In this chapter:

In this chapter, I'll discuss two main topics that really show off the incredible value provided by the Microsoft .NET Framework: *hosting* and *AppDomains*. Hosting allows any application to use the features of the common language runtime (CLR). In particular, this allows existing applications to be at least partially written using managed code. Furthermore, hosting allows applications the ability to offer customization and extensibility via programming.

Allowing extensibility means that third-party code will be running inside your process. In Windows, loading a third party's DLLs into a process has been fraught with peril. The DLL could easily have code in it that could compromise the application's data structures and code. The DLL could also try to use the security context of the application to gain access to resources that it should not have access to. The CLR's AppDomain feature solves all of these problems. AppDomains allow third-party untrusted code to run in an existing process, and the CLR guarantees that the data structures, code, and security context will not be exploited or compromised.

Programmers typically use hosting and AppDomains along with assembly loading and reflection. Using these four technologies together makes the CLR an incredibly rich and powerful platform. In this chapter, I'll focus on hosting and AppDomains. In the next chapter, I'll focus on assembly loading and reflection. When you learn and understand all of these technologies, you'll see how your investment in the .NET Framework today will certainly pay off down the line.

CLR Hosting

The .NET Framework runs on top of Windows. This means that the .NET Framework must be built using technologies that Windows can interface with. For starters, all managed module and assembly files must use the Windows portable executable (PE) file format and be either a Windows executable (EXE) file or a DLL.

When developing the CLR, Microsoft implemented it as a COM server contained inside a DLL; that is, Microsoft defined a standard COM interface for the CLR and assigned GUIDs to this interface and the COM server. When you install the .NET Framework, the COM server representing the CLR is registered in the Windows registry just as any other COM server would. If you want more information about this topic, refer to the MetaHost.h C++ header file that ships with the .NET Framework SDK. This header file defines the GUIDs and the unmanaged `ICLRMetaHost` interface definition.

Any Windows application can host the CLR. However, you shouldn't create an instance of the CLR COM server by calling `CoCreateInstance`; instead, your unmanaged host should call the `CLRCreateInstance` function declared in MetaHost.h. The `CLRCreateInstance` function is implemented in the MSCorEE.dll file, which is usually found in the C:\Windows\System32 directory. This DLL is affectionately referred to as the *shim*, and its job is to determine which version of the CLR to create; the shim DLL doesn't contain the CLR COM server itself.

A single machine may have multiple versions of the CLR installed, but there will be only one version of the MSCorEE.dll file (the shim).[1] The version of MSCorEE.dll installed on the machine is the version that shipped with the latest version of the CLR installed on the machine. Therefore, this version of MSCorEE.dll knows how to find any previous versions of the CLR that may be installed.

The actual CLR code is contained in a file whose name has changed with different versions of the CLR. For versions 1.0, 1.1, and 2.0, the CLR code is in a file called MSCorWks.dll, and for version 4, the CLR code is in a file called Clr.dll. Because you can have multiple versions of the CLR installed on a single machine, these files are installed into different directories as follows.[2]

- Version 1.0 is in C:\Windows\Microsoft.NET\Framework\v1.0.3705
- Version 1.1 is in C:\Windows\Microsoft.NET\Framework\v1.0.4322
- Version 2.0 is in C:\Windows\Microsoft.NET\Framework\v2.0.50727
- Version 4 is in C:\Windows\Microsoft.NET\Framework\v4.0.21006

The `CLRCreateInstance` function can return an `ICLRMetaHost` interface. A host application can call this interface's `GetRuntime` function, specifying the version of the CLR that the host would like to create. The shim then loads the desired version of the CLR into the host's process.

[1] If you are using a 64-bit version of Windows, there are actually two versions of the MSCorEE.dll file installed. One version is the 32-bit x86 version, which will be located in the C:\Windows\SysWOW64 directory. The other version is the 64-bit x64 or IA64 version (depending on your computer's CPU architecture), which will be located in the C:\Windows\System32 directory.

[2] Note that versions 3.0 and 3.5 of the .NET Framework were shipped with version 2.0 of the CLR; I do not show the directories for .NET Framework versions 3.0 and 3.5 because the CLR DLL loads from the v2.0.50727 directory.

By default, when a managed executable starts, the shim examines the executable file and extracts the information indicating the version of the CLR that the application was built and tested with. However, an application can override this default behavior by placing `requiredRuntime` and `supportedRuntime` entries in its XML configuration file (described in Chapter 2, "Building, Packaging, Deploying, and Administering Applications and Types," and Chapter 3, "Shared Assemblies and Strongly Named Assemblies").

The `GetRuntime` function returns a pointer to the unmanaged `ICLRRuntimeInfo` interface from which the `ICLRRuntimeHost` interface is obtained via the `GetInterface` method. The hosting application can call methods defined by this interface to:

- Set Host managers. Tell the CLR that the host wants to be involved in making decisions related to memory allocations, thread scheduling/synchronization, assembly loading, and more. The host can also state that it wants notifications of garbage collection starts and stops and when certain operations time out.
- Get CLR managers. Tell the CLR to prevent the use of some classes/members. In addition, the host can tell which code can and can't be debugged and which methods in the host should be called when a special event—such as an AppDomain unload, CLR stop, or stack overflow exception—occurs.
- Initialize and start the CLR.
- Load an assembly and execute code in it.
- Stop the CLR, thus preventing any more managed code from running in the Windows process.

There are many reasons why hosting the CLR is useful. Hosting allows any application to offer CLR features and a programmability story and to be at least partially written in managed code. Any application that hosts the runtime offers many benefits to developers who are trying to extend the application. Here are some of the benefits:

- Programming can be done in any programming language.
- Code is just-in-time (JIT)–compiled for speed (versus being interpreted).
- Code uses garbage collection to avoid memory leaks and corruption.
- Code runs in a secure sandbox.
- The host doesn't need to worry about providing a rich development environment. The host makes use of existing technologies: languages, compilers, editors, debuggers, profilers, and more.

If you are interested in using the CLR for hosting scenarios, I highly recommend that you get Steven Pratschner's excellent book, *Customizing the Microsoft .NET Framework Common Language Runtime* (Microsoft Press 2005), even though it focuses on pre-4 versions of the CLR.

Note Of course, a Windows process does not need to load the CLR at all. It needs to be loaded only if you want to execute managed code in a process. Prior to .NET Framework 4, the CLR allowed only one instance of itself to reside within a Windows process. That is, a process could contain no CLR, v1.0 of the CLR, v1.1 of the CLR, or v2.0 of the CLR. Allowing only one CLR version per process is a huge limitation. For example, Microsoft Outlook couldn't load two add-ins that were built and tested against different versions of the .NET Framework.

However, with .NET Framework 4, Microsoft now supports the ability to load v2.0 and v4.0 in a single Windows process, allowing components written for .NET Framework versions 2.0 and 4 to run side by side without experiencing any compatibility problems. This is a fantastic new feature, because it allows .NET Framework components to be used reliably in more scenarios than ever before. You can use the ClrVer.exe tool to see which CLR version(s) are loaded into any given process.

After a CLR is loaded into a Windows process, it can never be unloaded; calling the `AddRef` and `Release` methods on the `ICLRRuntimeHost` interface has no effect. The only way for the CLR to be unloaded from a process is for the process to terminate, causing Windows to clean up all resources used by the process.

AppDomains

When the CLR COM server initializes, it creates an *AppDomain*. An AppDomain is a logical container for a set of assemblies. The first AppDomain created when the CLR is initialized is called the *default AppDomain*; this AppDomain is destroyed only when the Windows process terminates.

In addition to the default AppDomain, a host using either unmanaged COM interface methods or managed type methods can instruct the CLR to create additional AppDomains. The whole purpose of an AppDomain is to provide isolation. Here are the specific features offered by an AppDomain:

- **Objects created by code in one AppDomain cannot be accessed directly by code in another AppDomain** When code in an AppDomain creates an object, that object is "owned" by that AppDomain. In other words, the object is not allowed to live beyond the lifetime of the AppDomain whose code constructed it. Code in other AppDomains can access another AppDomain's object only by using marshal-by-reference or marshal-by-value semantics. This enforces a clean separation and boundary because code in one AppDomain can't have a direct reference to an object created by code in a different AppDomain. This isolation allows AppDomains to be easily unloaded from a process without affecting code running in other AppDomains.

- **AppDomains can be unloaded** The CLR doesn't support the ability to unload a single assembly from an AppDomain. However, you can tell the CLR to unload an AppDomain, which will cause all of the assemblies currently contained in it to be unloaded as well.
- **AppDomains can be individually secured** When created, an AppDomain can have a permission set applied to it that determines the maximum rights granted to assemblies running in the AppDomain. This allows a host to load some code and be ensured that the code cannot corrupt or read important data structures used by the host itself.
- **AppDomains can be individually configured** When created, an AppDomain can have a bunch of configuration settings associated with it. These settings mostly affect how the CLR loads assemblies into the AppDomain. There are configuration settings related to search paths, version binding redirects, shadow copying, and loader optimizations.

Important A great feature of Windows is that it runs each application in its own process address space. This ensures that code in one application cannot access code or data in use by another application. Process isolation prevents security holes, data corruption, and other unpredictable behaviors from occurring, making Windows and the applications running on it robust. Unfortunately, creating processes in Windows is very expensive. The Win32 `CreateProcess` function is very slow, and Windows requires a lot of memory to virtualize a process's address space.

However, if an application consists entirely of managed code that is verifiably safe and doesn't call out into unmanaged code, there are no problems related to running multiple managed applications in a single Windows process. And AppDomains provide the isolation required to secure, configure, and terminate each of these applications.

Figure 22-1 shows a single Windows process that has one CLR COM server running in it. This CLR is currently managing two AppDomains (although there is no hard-coded limit to the number of AppDomains that could be running in a single Windows process). Each AppDomain has its own loader heap, each of which maintains a record of which types have been accessed because the AppDomain was created. These type objects were discussed in Chapter 4, "Type Fundamentals;" each type object in the loader heap has a method table, and each entry in the method table points to JIT-compiled native code if the method has been executed at least once.

In addition, each AppDomain has some assemblies loaded into it. AppDomain #1 (the default AppDomain) has three assemblies: MyApp.exe, TypeLib.dll, and System.dll. AppDomain #2 has two assemblies loaded into it: Wintellect.dll and System.dll.

You'll notice that the System.dll assembly has been loaded into both AppDomains. If both AppDomains are using a single type from System.dll, both AppDomains will have a type object for the same type allocated in each loader heap; the memory for the type object is not shared by all of the AppDomains. Furthermore, as code in an AppDomain calls methods defined by a type, the method's Intermediate Language (IL) code is JIT-compiled, and the resulting native code is associated with each AppDomain; the code for the method is not shared by all AppDomains that call it.

Not sharing the memory for the type objects or native code is wasteful. However, the whole purpose of AppDomains is to provide isolation; the CLR needs to be able to unload an AppDomain and free up all of its resources without adversely affecting any other AppDomain. Replicating the CLR data structures ensures that this is possible. It also ensures that a type used by multiple AppDomains has a set of static fields for each AppDomain.

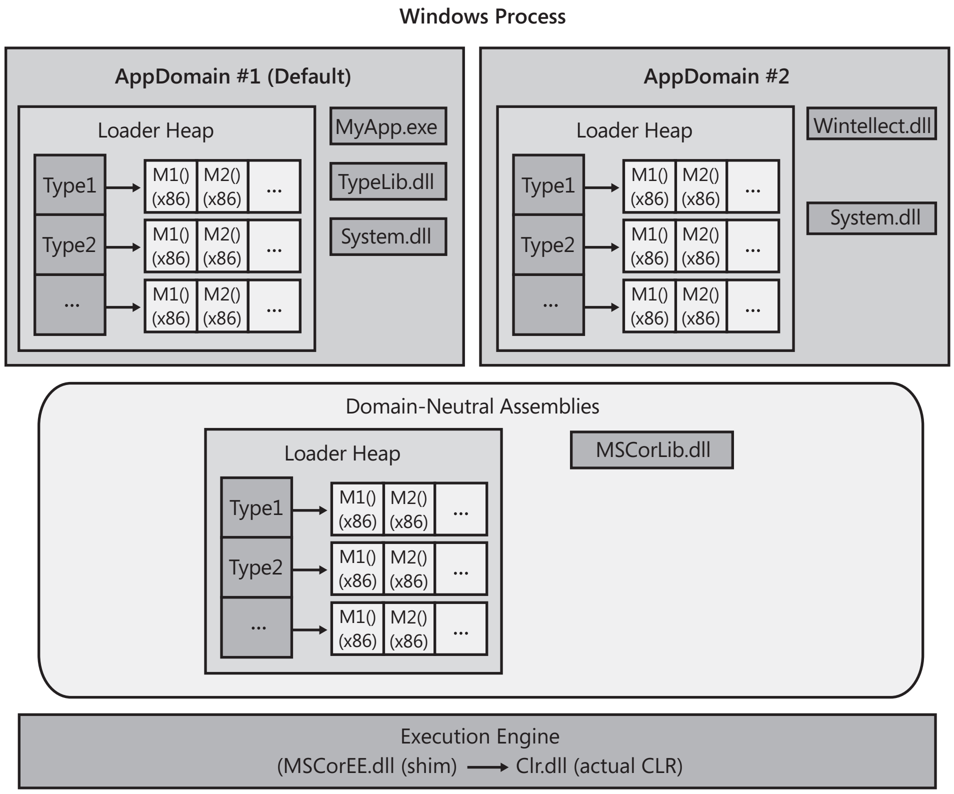


FIGURE 22-1 A single Windows process hosting the CLR and two AppDomains.

Some assemblies are expected to be used by several AppDomains. MSCorLib.dll is the best example. This assembly contains `System.Object`, `System.Int32`, and all of the other types that are so integral to the .NET Framework. This assembly is automatically loaded when the CLR initializes, and all AppDomains share the types in this assembly. To reduce resource usage, MSCorLib.dll is loaded in an AppDomain-neutral fashion; that is, the CLR maintains a special loader heap for assemblies that are loaded in a domain-neutral fashion. All type objects in this loader heap and all native code for methods of these types are shared by all AppDomains in the process. Unfortunately, the benefit gained by sharing these resources does come with a price: assemblies that are loaded domain-neutral can never be unloaded. The only way to reclaim the resources used by them is to terminate the Windows process to cause Windows to reclaim the resources.

Accessing Objects Across AppDomain Boundaries

Code in one AppDomain can communicate with types and objects contained in another AppDomain. However, access to these types and objects is allowed only through well-defined mechanisms. The following Ch22-1-AppDomains sample application demonstrates how to create a new AppDomain, load an assembly into it, and construct an instance of a type defined in that assembly. The code shows the different behaviors when constructing a type that is marshaled by reference, a type that is marshaled by value, and a type that can't be marshaled at all. The code also shows how these differently marshaled objects behave when the AppDomain that created them is unloaded. The Ch22-1-AppDomains sample application has very little code in it, but I have added a lot of comments. After the code listing, I'll walk through the code, explaining what the CLR is doing.

```
private static void Marshalling() {
   // Get a reference to the AppDomain that the calling thread is executing in
   AppDomain adCallingThreadDomain = Thread.GetDomain();

   // Every AppDomain is assigned a friendly string name (helpful for debugging)
   // Get this AppDomain's friendly string name and display it
   String callingDomainName = adCallingThreadDomain.FriendlyName;
   Console.WriteLine("Default AppDomain's friendly name={0}", callingDomainName);

   // Get and display the assembly in our AppDomain that contains the 'Main' method
   String exeAssembly = Assembly.GetEntryAssembly().FullName;
   Console.WriteLine("Main assembly={0}", exeAssembly);

   // Define a local variable that can refer to an AppDomain
   AppDomain ad2 = null;

   // *** DEMO 1: Cross-AppDomain Communication Using Marshal-by-Reference ***
   Console.WriteLine("{0}Demo #1", Environment.NewLine);

   // Create new AppDomain (security and configuration match current AppDomain)
   ad2 = AppDomain.CreateDomain("AD #2", null, null);
   MarshalByRefType mbrt = null;

   // Load our assembly into the new AppDomain, construct an object, marshal
   // it back to our AD (we really get a reference to a proxy)
   mbrt = (MarshalByRefType)
      ad2.CreateInstanceAndUnwrap(exeAssembly, "MarshalByRefType");

   Console.WriteLine("Type={0}", mbrt.GetType());  // The CLR lies about the type

   // Prove that we got a reference to a proxy object
   Console.WriteLine("Is proxy={0}", RemotingServices.IsTransparentProxy(mbrt));

   // This looks like we're calling a method on MarshalByRefType but we're not.
   // We're calling a method on the proxy type. The proxy transitions the thread
   // to the AppDomain owning the object and calls this method on the real object.
   mbrt.SomeMethod();
```

```
// Unload the new AppDomain
AppDomain.Unload(ad2);
// mbrt refers to a valid proxy object; the proxy object refers to an invalid AppDomain

try {
   // We're calling a method on the proxy type. The AD is invalid, exception is thrown
   mbrt.SomeMethod();
   Console.WriteLine("Successful call.");
}
catch (AppDomainUnloadedException) {
   Console.WriteLine("Failed call.");
}


// *** DEMO 2: Cross-AppDomain Communication Using Marshal-by-Value ***
Console.WriteLine("{0}Demo #2", Environment.NewLine);

// Create new AppDomain (security and configuration match current AppDomain)
ad2 = AppDomain.CreateDomain("AD #2", null, null);

// Load our assembly into the new AppDomain, construct an object, marshal
// it back to our AD (we really get a reference to a proxy)
mbrt = (MarshalByRefType)
   ad2.CreateInstanceAndUnwrap(exeAssembly, "MarshalByRefType");

// The object's method returns a COPY of the returned object;
// the object is marshaled by value (not by reference).
MarshalByValType mbvt = mbrt.MethodWithReturn();

// Prove that we did NOT get a reference to a proxy object
Console.WriteLine("Is proxy={0}", RemotingServices.IsTransparentProxy(mbvt));

// This looks like we're calling a method on MarshalByValType and we are.
Console.WriteLine("Returned object created " + mbvt.ToString());

// Unload the new AppDomain
AppDomain.Unload(ad2);
// mbvt refers to valid object; unloading the AppDomain has no impact.

try {
   // We're calling a method on an object; no exception is thrown
   Console.WriteLine("Returned object created " + mbvt.ToString());
   Console.WriteLine("Successful call.");
}
catch (AppDomainUnloadedException) {
   Console.WriteLine("Failed call.");
}


// DEMO 3: Cross-AppDomain Communication Using non-marshalable type ***
Console.WriteLine("{0}Demo #3", Environment.NewLine);

// Create new AppDomain (security and configuration match current AppDomain)
ad2 = AppDomain.CreateDomain("AD #2", null, null);
```

```
      // Load our assembly into the new AppDomain, construct an object, marshal
      // it back to our AD (we really get a reference to a proxy)
      mbrt = (MarshalByRefType)
         ad2.CreateInstanceAndUnwrap(exeAssembly, "MarshalByRefType");

      // The object's method returns a non-marshalable object; exception
      NonMarshalableType nmt = mbrt.MethodArgAndReturn(callingDomainName);
      // We won't get here...
   }


   // Instances can be marshaled-by-reference across AppDomain boundaries
   public sealed class MarshalByRefType : MarshalByRefObject {
      public MarshalByRefType() {
         Console.WriteLine("{0} ctor running in {1}",
            this.GetType().ToString(), Thread.GetDomain().FriendlyName);
      }

      public void SomeMethod() {
         Console.WriteLine("Executing in " + Thread.GetDomain().FriendlyName);
      }

      public MarshalByValType MethodWithReturn() {
         Console.WriteLine("Executing in " + Thread.GetDomain().FriendlyName);
         MarshalByValType t = new MarshalByValType();
         return t;
      }

      public NonMarshalableType MethodArgAndReturn(String callingDomainName) {
         // NOTE: callingDomainName is [Serializable]
         Console.WriteLine("Calling from '{0}' to '{1}'.",
            callingDomainName, Thread.GetDomain().FriendlyName);
         NonMarshalableType t = new NonMarshalableType();
         return t;
      }
   }


   // Instances can be marshaled-by-value across AppDomain boundaries
   [Serializable]
   public sealed class MarshalByValType : Object {
      private DateTime m_creationTime = DateTime.Now; // NOTE: DateTime is [Serializable]

      public MarshalByValType() {
         Console.WriteLine("{0} ctor running in {1}, Created on {2:D}",
            this.GetType().ToString(),
            Thread.GetDomain().FriendlyName,
            m_creationTime);
      }

      public override String ToString() {
         return m_creationTime.ToLongDateString();
      }
   }
```

```
// Instances cannot be marshaled across AppDomain boundaries
// [Serializable]
public sealed class NonMarshalableType : Object {
   public NonMarshalableType() {
      Console.WriteLine("Executing in " + Thread.GetDomain().FriendlyName);
   }
}
```

If you build and run the Ch22-1-AppDomains application, you get the following output.

```
Default AppDomain's friendly name= Ch22-1-AppDomains.exe
Main assembly=Ch22-1-AppDomains, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null

Demo #1
MarshalByRefType ctor running in AD #2
Type=MarshalByRefType
Is proxy=True
Executing in AD #2
Failed call.

Demo #2
MarshalByRefType ctor running in AD #2
Executing in AD #2
MarshalByValType ctor running in AD #2, Created on Friday, August 07, 2009
Is proxy=False
Returned object created Saturday, June 23, 2012
Returned object created Saturday, June 23, 2012
Successful call.

Demo #3
MarshalByRefType ctor running in AD #2
Calling from 'Ch22-1-AppDomains.exe' to 'AD #2'.
Executing in AD #2

Unhandled Exception: System.Runtime.Serialization.SerializationException:
Type 'NonMarshalableType' in assembly 'Ch22-1-AppDomains, Version=0.0.0.0,
Culture=neutral, PublicKeyToken=null' is not marked as serializable.
at MarshalByRefType.MethodArgAndReturn(String callingDomainName)
at Program.Marshalling()
at Program.Main()
```

Now, I will discuss what this code and the CLR are doing.

Inside the `Marshalling` method, I first get a reference to an `AppDomain` object that identifies the AppDomain the calling thread is currently executing in. In Windows, a thread is always created in the context of one process, and the thread lives its entire lifetime in that process. However, a one-to-one correspondence doesn't exist between threads and AppDomains. AppDomains are a CLR feature; Windows knows nothing about AppDomains. Because multiple AppDomains can be in a single Windows process, a thread can execute code in one AppDomain and then execute code in another AppDomain. From the CLR's perspective, a thread is executing code in one AppDomain at a time. A thread can ask the CLR what AppDomain it is currently executing in by calling `System.Threading.Thread`'s static `GetDomain` method. The thread could also query `System.AppDomain`'s static, read-only `CurrentDomain` property to get the same information.

When an AppDomain is created, it can be assigned a *friendly name*. A friendly name is just a `String` that you can use to identify an AppDomain. This is typically useful in debugging scenarios. Because the CLR creates the default AppDomain before any of our code can run, the CLR uses the executable file's file name as the default AppDomain's friendly name. My `Marshalling` method queries the default AppDomain's friendly name by using `System.AppDomain`'s read-only `FriendlyName` property.

Next, my `Marshalling` method queries the strong-name identity of the assembly (loaded into the default AppDomain) that defines the entry point method `Main` that calls `Marshalling`. This assembly defines several types: `Program`, `MarshalByRefType`, `MarshalBy ValType`, and `NonMarshalable-Type`. At this point, we're ready to look at the three demos that are all pretty similar to each other.

Demo #1: Cross-AppDomain Communication Using Marshal-by-Reference

In Demo #1, `System.AppDomain`'s static `CreateDomain` method is called, instructing the CLR to create a new AppDomain in the same Windows process. The `AppDomain` type actually offers several overloads of the `CreateDomain` method; I encourage you to study them and select the version that is most appropriate when you are writing code to create a new AppDomain. The version of `Create-Domain` that I call accepts three arguments:

- **A `String` identifying the friendly name I want assigned to the new AppDomain** I'm passing in "AD #2" here.
- **A `System.Security.Policy.Evidence` identifying the evidence that the CLR should use to calculate the AppDomain's permission set** I'm passing `null` here so that the new AppDomain will inherit the same permission set as the AppDomain creating it. Usually, if you want to create a security boundary around code in an AppDomain, you'd construct a `System.Security.PermissionSet` object, add the desired permission objects to it (instances of types that implement the `IPermission` interface), and then pass the resulting `Permission-Set` object reference to the overloaded version of the `CreateDomain` method that accepts a `PermissionSet`.
- **A `System.AppDomainSetup` identifying the configuration settings the CLR should use for the new AppDomain** Again, I'm passing `null` here so that the new AppDomain will inherit the same configuration settings as the AppDomain creating it. If you want the App-Domain to have a special configuration, construct an `AppDomainSetup` object, set its various properties to whatever you want, such as the name of the configuration file, and then pass the resulting `AppDomainSetup` object reference to the `CreateDomain` method.

Internally, the `CreateDomain` method creates a new AppDomain in the process. This AppDomain will be assigned the specified friendly name, security, and configuration settings. The new AppDomain will have its very own loader heap, which will be empty because there are currently no assemblies loading into the new AppDomain. When you create an AppDomain, the CLR does not create any threads in this AppDomain; no code runs in the AppDomain unless you explicitly have a thread call code in the AppDomain.

Now to create an instance of an object in the new AppDomain, we must first load an assembly into the new AppDomain and then construct an instance of a type defined in this assembly. This is precisely what the call to AppDomain's public, instance CreateInstanceAndUnwrap method does. When calling CreateInstanceAndUnwrap, I pass two arguments: a String identifying the assembly I want loaded into the new AppDomain (referenced by the ad2 variable) and another String identifying the name of the type that I want to construct an instance of. Internally, CreateInstanceAndUnwrap causes the calling thread to transition from the current AppDomain into the new AppDomain. Now, the thread (which is inside the call to CreateInstanceAndUnwrap) loads the specified assembly into the new AppDomain and then scans the assembly's type definition metadata table, looking for the specified type ("MarshalByRefType"). After the type is found, the thread calls the MarshalByRefType's parameterless constructor. Now the thread transitions back to the default AppDomain so that CreateInstanceAndUnwrap can return a reference to the new MarshalByRefType object.

Note There are overloaded versions of CreateInstanceAndUnwrap that allow you to call a type's constructor passing in arguments.

Although this sounds all fine and good, there is a problem: the CLR cannot allow a variable (root) living in one AppDomain to reference an object created in another AppDomain. If CreateInstanceAndUnwrap simply returned the reference to the object, isolation would be broken, and isolation is the whole purpose of AppDomains! So, just before CreateInstanceAndUnwrap returns the object reference, it performs some additional logic.

You'll notice that the MarshalByRefType type is derived from a very special base class: System.MarshalByRefObject. When CreateInstanceAndUnwrap sees that it is marshalling an object whose type is derived from MarshalByRefObject, the CLR will marshal the object by reference across the AppDomain boundaries. Here is what it means to marshal an object by reference from one AppDomain (the source AppDomain where the object is really created) to another AppDomain (the destination AppDomain from where CreateInstanceAndUnwrap is called).

When a source AppDomain wants to send or return the reference of an object to a destination AppDomain, the CLR defines a proxy type in the destination AppDomain's loader heap. This proxy type is defined using the original type's metadata, and therefore, it looks exactly like the original type; it has all of the same instance members (properties, events, and methods). The instance fields are not part of the type, but I'll talk more about this in a moment. This new type does have some instance fields defined inside of it, but these fields are not identical to that of the original data type. Instead, these fields indicate which AppDomain "owns" the real object and how to find the real object in the owning AppDomain. (Internally, the proxy object uses a GCHandle instance that refers to the real object. The GCHandle type is discussed in Chapter 21, "The Managed Heap and Garbage Collection.")

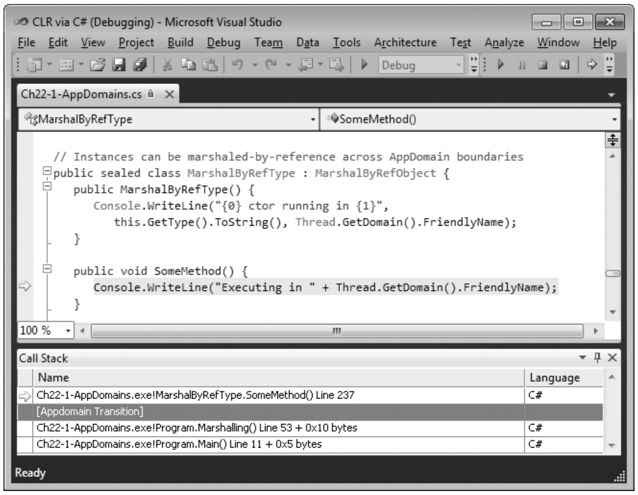
After this type is defined in the destination AppDomain, CreateInstanceAndUnwrap creates an instance of this proxy type, initializes its fields to identify the source AppDomain and the real object, and returns a reference to this proxy object to the destination AppDomain. In my Ch22-1-AppDomains application, the mbrt variable will be set to refer to this proxy. Notice that the object returned from

CreateInstanceAndUnwrap is actually not an instance of the MarshalByRefType type. The CLR will usually not allow you to cast an object of one type to an incompatible type. However, in this situation, the CLR does allow the cast, because this new type has the same instance members as defined on the original type. In fact, if you use the proxy object to call GetType, it actually lies to you and says that it is a MarshalByRefType object.

However, it is possible to prove that the object returned from CreateInstanceAndUnwrap is actually a reference to a proxy object. To do this, my Ch22-1-AppDomains application calls System.Runtime.Remoting.RemotingService's public, static IsTransparentProxy method passing in the reference returned from CreateInstanceAndUnwrap. As you can see from the output, IsTransparentProxy returns true, indicating that the object is a proxy.

Now, my Ch22-1-AppDomains application uses the proxy to call the SomeMethod method. Because the mbrt variable refers to a proxy object, the proxy's implementation of this method is called. The proxy's implementation uses the information fields inside the proxy object to transition the calling thread from the default AppDomain to the new AppDomain. Any actions now performed by this thread run under the new AppDomain's security and configuration settings. Next, the thread uses the proxy object's GCHandle field to find the real object in the new AppDomain, and then it uses the real object to call the real SomeMethod method.

There are two ways to prove that the calling thread has transitioned from the default AppDomain to the new AppDomain. First, inside the SomeMethod method, I call Thread.GetDomain().FriendlyName. This will return "AD #2" (as evidenced by the output) because the thread is now running in the new AppDomain created by using AppDomain.CreateDomain with "AD #2" as the friendly name parameter. Second, if you step through the code in a debugger and display the Call Stack window, the [AppDomain Transition] line marks where a thread has transitioned across an AppDomain boundary. See the Call Stack window near the bottom of Figure 22-2.


CLR via C# (Debugging) - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Architecture Test Analyze Window Help
Debug
Ch22-1-AppDomains.cs
MarshalByRefType
SomeMethod()
100 %
Ready

```
// Instances can be marshaled-by-reference across AppDomain boundaries
public sealed class MarshalByRefType : MarshalByRefObject {
   public MarshalByRefType() {
      Console.WriteLine("{0} ctor running in {1}",
         this.GetType().ToString(), Thread.GetDomain().FriendlyName);
   }

   public void SomeMethod() {
      Console.WriteLine("Executing in " + Thread.GetDomain().FriendlyName);
   }
```

Call Stack

Name	Language
Ch22-1-AppDomains.exe!MarshalByRefType.SomeMethod() Line 237	C#
[Appdomain Transition]	
Ch22-1-AppDomains.exe!Program.Marshalling() Line 53 + 0x10 bytes	C#
Ch22-1-AppDomains.exe!Program.Main() Line 11 + 0x5 bytes	C#

FIGURE 22-2 The Debugger's Call Stack window showing an AppDomain transition.

When the real `SomeMethod` method returns, it returns to the proxy's `SomeMethod` method, which transitions the thread back to the default AppDomain, and then the thread continues executing code in the default AppDomain.

Note When a thread in one AppDomain calls a method in another AppDomain, the thread transitions between the two AppDomains. This means that method calls across AppDomain boundaries are executed synchronously. However, at any given time, a thread is considered to be in just one AppDomain, and it executes code using that AppDomain's security and configuration settings. If you want to execute code in multiple AppDomains concurrently, you should create additional threads and have them execute whatever code you want in whatever AppDomains you want.

The next thing that my Ch22-1-AppDomains application does is call `AppDomain`'s public, static `Unload` method to force the CLR to unload the specified AppDomain, including all of the assemblies loaded into it. A garbage collection is forced to free up any objects that were created by code in the unloading AppDomain. At this point, the default AppDomain's `mbrt` variable still refers to a valid proxy object; however, the proxy object no longer refers to a valid AppDomain (because it has been unloaded).

When the default AppDomain attempts to use the proxy object to call the `SomeMethod` method, the proxy's implementation of this method is called. The proxy's implementation determines that the AppDomain that contained the real object has been unloaded, and the proxy's `SomeMethod` method throws an `AppDomainUnloadedException` to let the caller know that the operation cannot complete.

Wow! The CLR team at Microsoft had to do a lot of work to ensure AppDomain isolation, but it is important work because these features are used heavily and are being used more and more by developers every day. Obviously, accessing objects across AppDomain boundaries by using marshal-by-reference semantics has some performance costs associated with it, so you typically want to keep the use of this feature to a minimum.

I promised you that I'd talk a little more about instance fields. A type derived from `MarshalByRefObject` can define instance fields. However, these instance fields are not defined as being part of the proxy type and are not contained inside a proxy object. When you write code that reads from or writes to an instance field of a type derived from `MarshalByRefObject`, the JIT compiler emits code that uses the proxy object (to find the real AppDomain/object) by calling `System.Object`'s `FieldGetter` or `FieldSetter` methods, respectively. These methods are private and undocumented; they are basically methods that use reflection to get and set the value in a field. So although you can access fields of a type derived from `MarshalByRefObject`, the performance is particularly bad because the CLR really ends up calling methods to perform the field access. In fact, the performance is bad even if the object that you are accessing is in your own AppDomain.[3]

[3] If the CLR required that all fields be private (which is recommended for good data encapsulation), then the `FieldGetter` and `FieldSetter` methods wouldn't have to exist and accessing fields from methods could always have been direct, avoiding any performance penalty.

The Performance of Accessing Instance Fields

To demonstrate the significance of this performance hit, I wrote the following code.

```
private sealed class NonMBRO : Object             { public Int32 x; }
private sealed class MBRO    : MarshalByRefObject { public Int32 x; }

private static void FieldAccessTiming(){
   const Int32 count = 100000000;
   NonMBRO nonMbro = new NonMBRO();
   MBRO mbro = new MBRO();

   Stopwatch sw = Stopwatch.StartNew();
   for (Int32 c = 0; c < count; c++) nonMbro.x++;
   Console.WriteLine("{0}", sw.Elapsed);   // 00:00:00.4073560

   sw = Stopwatch.StartNew();
   for (Int32 c = 0; c < count; c++) mbro.x++;
   Console.WriteLine("{0}", sw.Elapsed);   // 00:00:02.5388665
}
```

When I ran this code, it took ~.4 seconds to access the instance field of a `NonMBRO` class that is derived from `Object`, and it took 2.54 seconds to access the instance field of an `MBRO` class that is derived from `MarshalByRefObject`. So, accessing an instance field of a class derived from `MarshalByRefObject` takes more than ~6 times longer!

From a usability standpoint, a type derived from `MarshalByRefObject` should really avoid defining any static members. The reason is that static members are always accessed in the context of the calling AppDomain. No AppDomain transition can occur because a proxy object contains the information identifying which AppDomain to transition to, but there is no proxy object when calling a static member. Having a type's static members execute in one AppDomain while instance members execute in another AppDomain would make a very awkward programming model.

Because there are no roots in the second AppDomain, the original object referred to by the proxy could be garbage collected. Of course, this is not ideal. On the other hand, if the original object is held in memory indefinitely, then the proxy could go away and the original object would still live; this is also not ideal. The CLR solves this problem by using a *lease manager*. When a proxy for an object is created, the CLR keeps the object alive for five minutes. If no calls have been made through the proxy after five minutes, then the object is *deactivated* and will have its memory freed at the next garbage collection. After each call into the object, the lease manager renews the object's lease so that it is guaranteed to remain in memory for another two minutes before being deactivated. If an application attempts to call into an object through a proxy after the object's lease has expired, the CLR throws a `System.Runtime.Remoting.RemotingException`.

It is possible to override the default lease times of five minutes and two minutes by overriding `MarshalByRefObject`'s virtual `InitializeLifetimeServices` method. For more information, see the section titled "Lifetime Leases" in the .NET Framework SDK documentation.

Demo #2: Cross-AppDomain Communication Using Marshal-by-Value

Demo #2 is very similar to Demo #1. Again, another AppDomain is created exactly as Demo #1 did it. Then, `CreateInstanceAndUnwrap` is called to load the same assembly into the new AppDomain and create an instance of a `MarshalByRefType` object in this new AppDomain. Next, the CLR creates a proxy to the object and the `mbrt` variable (in the default AppDomain) is initialized referring to the proxy. Now, using the proxy, I call `MethodWithReturn`. This method, which takes no arguments, will execute in the new AppDomain to create an instance of the `MarshalByValType` type before returning a reference to the object to the default AppDomain.

`MarshalByValType` is not derived from `System.MarshalByRefObject`, and therefore, the CLR cannot define a proxy type to create an instance from; the object can't be marshaled by reference across the AppDomain boundary.

However, because `MarshalByValType` is marked with the `[Serializable]` custom attribute, `MethodWithReturn` is allowed to marshal the object by value. The next paragraph describes what it means to marshal an object by value from one AppDomain (the source AppDomain) to another AppDomain (the destination AppDomain). For more information about the CLR's serialization and deserialization mechanisms, see Chapter 24, "Runtime Serialization."

When a source AppDomain wants to send or return a reference to an object to a destination AppDomain, the CLR serializes the object's instance fields into a byte array. This byte array is copied from the source AppDomain to the destination AppDomain. Then, the CLR deserializes the byte array in the destination AppDomain. This forces the CLR to load the assembly that defines the type being deserialized into the destination AppDomain if it is not already loaded. Then, the CLR creates an instance of the type and uses the values in the byte array to initialize the object's fields so that they have values identical to those they had in the original object. In other words, the CLR makes an exact duplicate of the source object in the destination's AppDomain `MethodWithReturn`, and then returns a reference to this copy; the object has been marshaled by value across the AppDomain's boundary.

Important When loading the assembly, the CLR uses the destination AppDomain's policies and configuration settings (for example, the AppDomain can have a different AppBase directory or different version binding redirections). These policy differences might prevent the CLR from locating the assembly. If the assembly cannot be loaded, an exception will be thrown, and the destination will not receive a reference to the object.

At this point, the object in the source AppDomain and the object in the destination AppDomain live separate lifetimes, and their states can change independently of each other. If there are no roots in the source AppDomain keeping the original object alive (as in my Ch22-1-AppDomains application), its memory will be reclaimed at the next garbage collection.

To prove that the object returned from MethodWithReturn is not a reference to a proxy object, my Ch22-1-AppDomains application calls System.Runtime.Remoting.RemotingService's public, static IsTransparentProxy method passing in the reference returned from MethodWithReturn. As you can see from the output, IsTransparentProxy returns false, indicating that the object is a real object, not a proxy.

Now, my program uses the real object to call the ToString method. Because the mbvt variable refers to a real object, the real implementation of this method is called, and no AppDomain transition occurs. This can be evidenced by examining the debugger's Call Stack window, which will not show an [Appdomain Transition] line.

To further prove that no proxy is involved, my Ch22-1-AppDomains application unloads the new AppDomain and then attempts to call the ToString method again. Unlike in Demo #1, the call succeeds this time because unloading the new AppDomain had no impact on objects "owned" by the default AppDomain, and this includes the object that was marshaled by value.

Demo #3: Cross-AppDomain Communication Using Non-Marshalable Types

Demo #3 starts out very similar to Demos #1 and #2. Just as in Demos #1 and #2, an AppDomain is created. Then, CreateInstanceAndUnwrap is called to load the same assembly into the new AppDomain, create a MarshalByRefType object in this new AppDomain, and have the mbrt variable refer to a proxy to this object.

Then, using this proxy, I call MethodArgAndReturn, which accepts an argument. Again, the CLR must maintain AppDomain isolation, so it cannot simply pass the reference to the argument into the new AppDomain. If the type of the object is derived from MarshalByRefObject, the CLR will make a proxy for it and marshal it by reference. If the object's type is marked as [Serializable], the CLR will serialize the object (and its children) to a byte array, marshal the byte array into the new AppDomain, and then deserialize the byte array into an object graph, passing the root of the object graph into the MethodArgAndReturn method.

In this particular demo, I am passing a System.String object across AppDomain boundaries. The System.String type is not derived from MarshalByRefObject, so the CLR cannot create a proxy. Fortunately, System.String is marked as [Serializable], and therefore the CLR can marshal it by value, which allows the code to work. Note that for String objects, the CLR performs a special optimization. When marshaling a String object across an AppDomain boundary, the CLR just passes the reference to the String object across the boundary; it does not make a copy of the String object. The CLR can offer this optimization because String objects are immutable; therefore, it is impossible for code in one AppDomain to corrupt a String object's characters.[4] For more about String immutability, see Chapter 14, "Chars, Strings, and Working with Text."

[4] By the way, this is why the System.String class is sealed. If the class were not sealed, then you could define your own class derived from String and add your own fields. If you did this, there is no way that the CLR could ensure that your "string" class was immutable.

Inside `MethodArgAndReturn`, I display the string passed into it to show that the string came across the AppDomain boundary, and then I create an instance of the `NonMarshalableType` type and return a reference to this object to the default AppDomain. Because `NonMarshalableType` is not derived from `System.MarshalByRefObject` and is also not marked with the `[Serializable]` custom attribute, `MethodArgAndReturn` is not allowed to marshal the object by reference or by value—the object cannot be marshaled across an AppDomain boundary at all! To report this, `MethodArgAndReturn` throws a `SerializationException` in the default AppDomain. Because my program doesn't catch this exception, the program just dies.

AppDomain Unloading

One of the great features of AppDomains is that you can unload them. Unloading an AppDomain causes the CLR to unload all of the assemblies in the AppDomain, and the CLR frees the AppDomain's loader heap as well. To unload an AppDomain, you call `AppDomain`'s `Unload` static method (as the Ch22-1-AppDomains application does). This call causes the CLR to perform a lot of actions to gracefully unload the specified AppDomain:

1. The CLR suspends all threads in the process that have ever executed managed code.
2. The CLR examines all of the threads' stacks to see which threads are currently executing code in the AppDomain being unloaded, or which threads might return at some point to code in the AppDomain that is being unloaded. The CLR forces any threads that have the unloading AppDomain on their stack to throw a `ThreadAbortException` (resuming the thread's execution). This causes the threads to unwind, executing any `finally` blocks on their way out so that cleanup code executes. If no code catches the `ThreadAbortException`, it will eventually become an unhandled exception that the CLR swallows; the thread dies, but the process is allowed to continue running. This is unusual, because for all other unhandled exceptions, the CLR kills the process.

Important The CLR will not immediately abort a thread that is currently executing code in a `finally` block, `catch` block, a `class` constructor, a critical execution region, or in unmanaged code. If the CLR allowed this, cleanup code, error recovery code, type initialization code, critical code, or arbitrary code that the CLR knows nothing about would not complete, resulting in the application behaving unpredictably and with potential security holes. An aborting thread is allowed to finish executing these code blocks and then, at the end of the code block, the CLR forces the thread to throw a `ThreadAbortException`.

3. After all threads discovered in step 2 have left the AppDomain, the CLR then walks the heap and sets a flag in each proxy object that referred to an object created by the unloaded AppDomain. These proxy objects now know that the real object they referred to is gone. If any code now calls a method on an invalid proxy object, the method will throw an `AppDomainUnloadedException`.
4. The CLR forces a garbage collection to occur, reclaiming the memory used by any objects that were created by the now unloaded AppDomain. The `Finalize` methods for these objects are called, giving the objects a chance to clean themselves up properly.
5. The CLR resumes all of the remaining threads. The thread that called `AppDomain.Unload` will now continue running; calls to `AppDomain.Unload` occur synchronously.

My Ch22-1-AppDomains application uses just one thread to do all of the work. Whenever my code calls `AppDomain.Unload`, there are no threads in the unloading AppDomain, and therefore, the CLR doesn't have to throw any `ThreadAbortException` exceptions. I'll talk more about `ThreadAbortException` later in this chapter.

By the way, when a thread calls `AppDomain.Unload`, the CLR waits 10 seconds for the threads in the unloading AppDomain to leave it. If after 10 seconds, the thread that called `AppDomain.Unload` doesn't return, it will throw a `CannotUnloadAppDomainException`, and the AppDomain may or may not be unloaded in the future.

Note If a thread calling `AppDomain.Unload` is in the AppDomain being unloaded, the CLR creates another thread that attempts to unload the AppDomain. The first thread will forcibly throw the `ThreadAbortException` and unwind. The new thread will wait for the AppDomain to unload, and then the new thread terminates. If the AppDomain fails to unload, the new thread will process a `CannotUnloadAppDomainException`, but because you did not write the code that this new thread executes, you can't catch this exception.

AppDomain Monitoring

A host application can monitor the resources that an AppDomain consumes. Some hosts will use this information to decide when to forcibly unload an AppDomain should its memory or CPU consumption rise above what the host considers reasonable. Monitoring can also be used to compare the resource consumption of different algorithms to determine which uses fewer resources. Because AppDomain monitoring incurs additional overhead, hosts must explicitly turn the monitoring on by setting `AppDomain`'s static `MonitoringEnabled` property to `true`. This turns on monitoring for all AppDomains. After monitoring is turned on, it cannot be turned off; attempting to set the `MonitoringEnabled` property to `false` causes an `ArgumentException` to be thrown.

After monitoring is turned on, your code can query the following four read-only properties offered by the AppDomain class:

- **`MonitoringSurvivedProcessMemorySize`** This static `Int64` property returns the number of bytes that are currently in use by all AppDomains controlled by the current CLR instance. The number is accurate as of the last garbage collection.
- **`MonitoringTotalAllocatedMemorySize`** This instance `Int64` property returns the number of bytes that have been allocated by a specific AppDomain. The number is accurate as of the last garbage collection.
- **`MonitoringSurvivedMemorySize`** This instance `Int64` property returns the number of bytes that are currently in use by a specific AppDomain. The number is accurate as of the last garbage collection.
- **`MonitoringTotalProcessorTime`** This instance `TimeSpan` property returns the amount of CPU usage incurred by a specific AppDomain.

The following class shows how to use three of these properties to see what has changed within an AppDomain between two points in time.

```
private sealed class AppDomainMonitorDelta : IDisposable {
   private AppDomain m_appDomain;
   private TimeSpan m_thisADCpu;
   private Int64 m_thisADMemoryInUse;
   private Int64 m_thisADMemoryAllocated;

   static AppDomainMonitorDelta() {
      // Make sure that AppDomain monitoring is turned on
      AppDomain.MonitoringIsEnabled = true;
   }

   public AppDomainMonitorDelta(AppDomain ad) {
      m_appDomain = ad ?? AppDomain.CurrentDomain;
      m_thisADCpu = m_appDomain.MonitoringTotalProcessorTime;
      m_thisADMemoryInUse = m_appDomain.MonitoringSurvivedMemorySize;
      m_thisADMemoryAllocated = m_appDomain.MonitoringTotalAllocatedMemorySize;
   }

   public void Dispose() {
      GC.Collect();
      Console.WriteLine("FriendlyName={0}, CPU={1}ms", m_appDomain.FriendlyName,
         (m_appDomain.MonitoringTotalProcessorTime - m_thisADCpu).TotalMilliseconds);
      Console.WriteLine("   Allocated {0:N0} bytes of which {1:N0} survived GCs",
         m_appDomain.MonitoringTotalAllocatedMemorySize - m_thisADMemoryAllocated,
         m_appDomain.MonitoringSurvivedMemorySize - m_thisADMemoryInUse);
    }
}
```

The following code shows how to use the AppDomainMonitorDelta class.

```
private static void AppDomainResourceMonitoring() {
    using (new AppDomainMonitorDelta(null)) {
       // Allocate about 10 million bytes that will survive collections
       var list = new List<Object>();
       for (Int32 x = 0; x < 1000; x++) list.Add(new Byte[10000]);

       // Allocate about 20 million bytes that will NOT survive collections
       for (Int32 x = 0; x < 2000; x++) new Byte[10000].GetType();

       // Spin the CPU for about 5 seconds
       Int64 stop = Environment.TickCount + 5000;
       while (Environment.TickCount < stop) ;
   }
}
```

When I execute this code, I get the following output.

```
FriendlyName=03-Ch22-1-AppDomains.exe, CPU=5031.25ms
   Allocated 30,159,496 bytes of which 10,085,080 survived GCs
```

AppDomain First-Chance Exception Notifications

Each AppDomain can have associated with it a series of callback methods that get invoked when the CLR begins looking for catch blocks within an AppDomain. These methods can perform logging, or a host can use this mechanism to monitor exceptions being thrown within an AppDomain. The callbacks cannot handle the exception or swallow it in any way; they are just receiving a notification that the exception has occurred. To register a callback method, just add a delegate to AppDomain's instance FirstChanceException event.

Here is how the CLR processes an exception: when the exception is first thrown, the CLR invokes any FirstChanceException callback methods registered with the AppDomain that are throwing the exception. Then, the CLR looks for any catch blocks on the stack that are within the same AppDomain. If a catch block handles the exception, then processing of the exception is complete and execution continues as normal. If the AppDomain has no catch block to handle the exception, then the CLR walks up the stack to the calling AppDomain and throws the same exception object again (after serializing and deserializing it). At this point, it is as if a brand new exception is being thrown, and the CLR invokes any FirstChanceException callback methods registered with the now current AppDomain. This continues until the top of the thread's stack is reached. At that point, if the exception is not handled by any code, the CLR terminates the whole process.

How Hosts Use AppDomains

So far, I've talked about hosts and how they load the CLR. I've also talked about how the hosts tell the CLR to create and unload AppDomains. To make the discussion more concrete, I'll describe some common hosting and AppDomain scenarios. In particular, I'll explain how different application types host the CLR and how they manage AppDomains.

Executable Applications

Console UI applications, NT Service applications, Windows Forms applications, and Windows Presentation Foundation (WPF) applications are all examples of self-hosted applications that have managed EXE files. When Windows initializes a process by using a managed EXE file, Windows loads the shim, and the shim examines the CLR header information contained in the application's assembly (the EXE file). The header information indicates the version of the CLR that was used to build and test the application. The shim uses this information to determine which version of the CLR to load into the process. After the CLR loads and initializes, it again examines the assembly's CLR header to determine which method is the application's entry point (`Main`). The CLR invokes this method, and the application is now up and running.

As the code runs, it accesses other types. When referencing a type contained in another assembly, the CLR locates the necessary assembly and loads it into the same AppDomain. Any additionally referenced assemblies also load into the same AppDomain. When the application's `Main` method returns, the Windows process terminates (destroying the default AppDomain and all other AppDomains).

Note By the way, you can call `System.Environment`'s static `Exit` method if you want to shut down the Windows process including all of its AppDomains. `Exit` is the most graceful way of terminating a process because it first calls the `Finalize` methods of all of the objects on the managed heap and then releases all of the unmanaged COM objects held by the CLR. Finally, `Exit` calls the Win32 `ExitProcess` function.

It's possible for the application to tell the CLR to create additional AppDomains in the process's address space. In fact, this is what my Ch22-1-AppDomains application did.

Microsoft Silverlight Rich Internet Applications

Microsoft's Silverlight runtime technology uses a special CLR that is different from the normal desktop version of the .NET Framework. After the Silverlight runtime is installed, navigating to a website that uses Silverlight causes the Silverlight CLR (CoreClr.dll) to load in your browser (which may or may not be Windows Internet Explorer—you may not even be using a Windows machine). Each Silverlight control on the page runs in its own AppDomain. When the user closes a tab or navigates to another website, any Silverlight controls no longer in use have their AppDomains unloaded. The Silverlight code running in the AppDomain runs in a limited-security sandbox so that it cannot harm the user or the machine in any way.

Microsoft ASP.NET and XML Web Services Applications

ASP.NET is implemented as an ISAPI DLL (implemented in ASPNet_ISAPI.dll). The first time a client requests a URL handled by the ASP.NET ISAPI DLL, ASP.NET loads the CLR. When a client makes a request of a web application, ASP.NET determines if this is the first time a request has been made. If it is, ASP.NET tells the CLR to create a new AppDomain for this web application; each web application is identified by its virtual root directory. ASP.NET then tells the CLR to load the assembly that contains the type exposed by the web application into this new AppDomain, creates an instance of this type, and starts calling methods in it to satisfy the client's web request. If the code references more types, the CLR will load the required assemblies into the web application's AppDomain.

When future clients make requests of an already running web application, ASP.NET doesn't create a new AppDomain; instead, it uses the existing AppDomain, creates a new instance of the web application's type, and starts calling methods. The methods will already be JIT-compiled into native code, so the performance of processing all subsequent client requests is excellent.

If a client makes a request of a different web application, ASP.NET tells the CLR to create a new AppDomain. This new AppDomain is typically created inside the same worker process as the other AppDomains. This means that many web applications run in a single Windows process, which improves the efficiency of the system overall. Again, the assemblies required by each web application are loaded into an AppDomain created for the sole purpose of isolating that web application's code and objects from other web applications.

A fantastic feature of ASP.NET is that the code for a website can be changed on the fly without shutting down the web server. When a website's file is changed on the hard disk, ASP.NET detects this, unloads the AppDomain that contains the old version of the files (when the last currently running request finishes), and then creates a new AppDomain, loading into it the new versions of the files. To make this happen, ASP.NET uses an AppDomain feature called *shadow copying*.

Microsoft SQL Server

Microsoft SQL Server is an unmanaged application because most of its code is still written in unmanaged C++. SQL Server allows developers to create stored procedures by using managed code. The first time a request comes in to the database to run a stored procedure written in managed code, SQL Server loads the CLR. Stored procedures run in their own secured AppDomain, prohibiting the stored procedures from adversely affecting the database server.

This functionality is absolutely incredible! It means that developers will be able to write stored procedures in the programming language of their choice. The stored procedure can use strongly typed data objects in its code. The code will also be JIT-compiled into native code when executed instead of being interpreted. And developers can take advantage of any types defined in the Framework Class Library (FCL) or in any other assembly. The result is that our job becomes much easier and our applications perform much better. What more could a developer ask for?!

Your Own Imagination

Productivity applications such as word processors and spreadsheets also allow users to write macros in any programming language they choose. These macros will have access to all of the assemblies and types that work with the CLR. They will be compiled, so they will execute fast, and, most important, these macros will run in a secure AppDomain so that users don't get hit with any unwanted surprises. Your own applications can use this ability, too, in any way you want.

Advanced Host Control

In this section, I'll mention some more advanced topics related to hosting the CLR. My intent is to give you a taste of what is possible, and this will help you to understand more of what the CLR is capable of. I encourage you to seek out other texts if you find this information particularly interesting.

Managing the CLR by Using Managed Code

The `System.AppDomainManager` class allows a host to override CLR default behavior by using managed code instead of using unmanaged code. Of course, using managed code makes implementing a host easier. All you need to do is define your class and derive it from the `System.AppDomainManager` class, overriding any virtual methods where you want to take over control. Your class should then be built into its very own assembly and installed into the global assembly cache (GAC) because the assembly needs to be granted full-trust, and all assemblies in the GAC are always granted full-trust.

Then, you need to tell the CLR to use your `AppDomainManager`-derived class. In code, the best way to do this is to create an `AppDomainSetup` object initializing its `AppDomainManagerAssembly` and `AppDomainManagerType` properties, both of which are of type `String`. Set the `AppDomainManagerAssembly` property to the string identifying the strong-name identity of the assembly that defines your `AppDomainManager`-derived class, and then set the `AppDomainManagerType` property to the full name of your `AppDomainManager`-derived class. Alternatively, `AppDomainManager` can be set in your application's XML configuration file by using the `appDomainManagerAssembly` and `appDomainManagerType` elements. In addition, a native host could query for the `ICLRControl` interface and call this interface's `SetAppDomainManagerType` function, passing in the identity of the GAC-installed assembly and the name of the `AppDomainManager`-derived class.[5]

Now, let's talk about what an `AppDomainManager`-derived class can do. The purpose of the `AppDomainManager`-derived class is to allow a host to maintain control even when an add-in tries to create AppDomains of its own. When code in the process tries to create a new AppDomain, the `AppDomainManager`-derived object in that AppDomain can modify security and configuration settings.

[5] It is also possible to configure an `AppDomainManager` by using environment variables and registry settings, but these mechanisms are more cumbersome than the methods mentioned in the text and should be avoided except for some testing scenarios.

It can also decide to fail an AppDomain creation, or it can decide to return a reference to an existing AppDomain instead. When a new AppDomain is created, the CLR creates a new `AppDomainManager`-derived object in the AppDomain. This object can also modify configuration settings, how execution context is flowed between threads, and permissions granted to an assembly.

Writing a Robust Host Application

A host can tell the CLR what actions to take when a failure occurs in managed code. Here are some examples (listed from least severe to most severe):

- The CLR can abort a thread if the thread is taking too long to execute and return a response. (I'll discuss this more in the next section.)
- The CLR can unload an AppDomain. This aborts all of the threads that are in the AppDomain and causes the problematic code to be unloaded.
- The CLR can be disabled. This stops any more managed code from executing in the process, but unmanaged code is still allowed to run.
- The CLR can exit the Windows process. This aborts all of the threads and unloads all of the AppDomains first so that cleanup operations occur, and then the process terminates.

The CLR can abort a thread or AppDomain gracefully or rudely. A graceful abort means that cleanup code executes. In other words, code in `finally` blocks runs, and objects have their `Finalize` methods executed. A rude abort means that cleanup code does not execute. In other words, code in `finally` blocks may not run, and objects may not have their `Finalize` methods executed. A graceful abort cannot abort a thread that is in a `catch` or `finally` block. However, a rude abort will abort a thread that is in a `catch` or `finally` block. Unfortunately, a thread that is in unmanaged code or in a constrained execution region (CER) cannot be aborted at all.

A host can set what is called an *escalation policy*, which tells the CLR how to deal with managed code failures. For example, SQL Server tells the CLR what to do should an unhandled exception be thrown while the CLR is executing managed code. When a thread experiences an unhandled exception, the CLR first attempts to upgrade the exception to a graceful thread abort. If the thread does not abort in a specified time period, the CLR attempts to upgrade the graceful thread abort to a rude thread abort.

What I just described is what usually happens. However, if the thread experiencing the unhandled exception is in a *critical region*, the policy is different. A thread that is in a critical region is a thread that has entered a thread synchronization lock that must be released by the same thread, for example, a thread that has called `Monitor.Enter`, `Mutex`'s `WaitOne`, or one of `ReaderWriterLock`'s `AcquireReaderLock` or `AcquireWriterLock` methods.[6] Successfully waiting for an `AutoResetEvent`, `ManualResetEvent`, or `Semaphore` doesn't cause the thread to be in a critical region because another

[6] All of these locks internally call `Thread`'s `BeginCriticalRegion` and `EndCriticalRegion` methods to indicate when they enter and leave critical regions. Your code can call these methods too if you need to. Normally, this would be necessary only if you are interoperating with unmanaged code.

thread can signal these synchronization objects. When a thread is in a critical region, the CLR believes that the thread is accessing data that is shared by multiple threads in the same AppDomain. After all, this is probably why the thread took the lock. If the thread is accessing shared data, just terminating the thread isn't good enough, because other threads may then try to access the shared data that is now corrupt, causing the AppDomain to run unpredictably or with possible security vulnerabilities.

So, when a thread in a critical region experiences an unhandled exception, the CLR first attempts to upgrade the exception to a graceful AppDomain unload in an effort to get rid of all of the threads and data objects that are currently in use. If the AppDomain doesn't unload in a specified amount of time, the CLR upgrades the graceful AppDomain unload to a rude AppDomain unload.

How a Host Gets Its Thread Back

Normally, a host application wants to stay in control of its threads. Let's take a database server as an example. When a request comes into the database server, a thread picks up the request and then dispatches the request to another thread that is to perform the actual work. This other thread may need to execute code that wasn't created and tested by the team that produced the database server. For example, imagine a request coming into the database server to execute a stored procedure written in managed code by the company running the server. It's great that the database server can run the stored procedure code in its own AppDomain, which is locked down with security. This prevents the stored procedure from accessing any objects outside of its own AppDomain, and it also prevents the code from accessing resources that it is not allowed to access, such as disk files or the clipboard.

But what if the code in the stored procedure enters an infinite loop? In this case, the database server has dispatched one of its threads into the stored procedure code, and this thread is never coming back. This puts the server in a precarious position; the future behavior of the server is unknown. For example, the performance might be terrible now because a thread is in an infinite loop. Should the server create more threads? Doing so uses more resources (such as stack space), and these threads could also enter an infinite loop themselves.

To solve these problems, the host can take advantage of thread aborting. Figure 22-3 shows the typical architecture of a host application trying to solve the runaway thread problem. Here's how it works (the numbers correspond to the circled numbers in the figure):

1. A client sends a request to the server.
2. A server thread picks up this request and dispatches it to a thread pool thread to perform the actual work.
3. A thread pool thread picks up the client request and executes trusted code written by the company that built and tested the host application.

4. This trusted code then enters a `try` block, and from within the `try` block, calls across an AppDomain boundary (via a type derived from `MarshalByRefObject`). This AppDomain contains the untrusted code (perhaps a stored procedure) that was not built and tested by the company that produced the host application. At this point, the server has given control of its thread to some untrusted code; the server is feeling nervous right now.

5. When the host originally received the client's request, it recorded the time. If the untrusted code doesn't respond to the client in some administrator-set amount of time, the host calls `Thread`'s `Abort` method, asking the CLR to stop the thread pool thread, forcing it to throw a `ThreadAbortException`.

6. At this point, the thread pool thread starts unwinding, calling `finally` blocks so that cleanup code executes. Eventually, the thread pool thread crosses back over the AppDomain boundary. Because the host's stub code called the untrusted code from inside a `try` block, the host's stub code has a `catch` block that catches the `ThreadAbortException`.

7. In response to catching the `ThreadAbortException`, the host calls `Thread`'s `ResetAbort` method. I'll explain the purpose of this call shortly.

8. Now that the host's code has caught the `ThreadAbortException`, the host can return some sort of failure back to the client and allow the thread pool thread to return to the pool so that it can be used for a future client request.

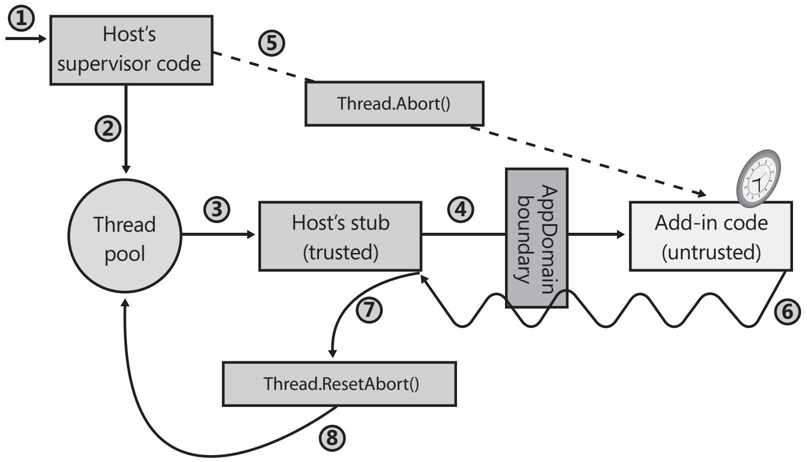


FIGURE 22-3 How a host application gets its thread back.

Let me now clear up a few loose ends about this architecture. First, Thread's Abort method is asynchronous. When Abort is called, it sets the target thread's AbortRequested flag and returns immediately. When the runtime detects that a thread is to be aborted, the runtime tries to get the thread to a *safe place*. A thread is in a safe place when the runtime feels that it can stop what the thread is doing without causing disastrous effects. A thread is in a safe place if it is performing a managed blocking operation such as sleeping or waiting. A thread can be corralled to a safe place by using hijacking (described in Chapter 21). A thread is not in a safe place if it is executing a type's class constructor, code in a catch or finally block, code in a CER, or unmanaged code.

After the thread reaches a safe place, the runtime will detect that the AbortRequested flag is set for the thread. This causes the thread to throw a ThreadAbortException. If this exception is not caught, the exception will be unhandled, all pending finally blocks will execute, and the thread will kill itself gracefully. Unlike all other exceptions, an unhandled ThreadAbortException does not cause the application to terminate. The runtime silently eats this exception and the thread dies, but the application and all of its remaining threads continue to run just fine.

In my example, the host catches the ThreadAbortException, allowing the host to regain control of the thread and return it to the pool. But there is a problem: What is to stop the untrusted code from catching the ThreadAbortException itself to keep control of the thread? The answer is that the CLR treats the ThreadAbortException in a very special manner. Even when code catches the ThreadAbortException, the CLR doesn't allow the exception to be swallowed. In other words, at the end of the catch block, the CLR automatically rethrows the ThreadAbortException exception.

This CLR feature raises another question: If the CLR rethrows the ThreadAbortException at the end of a catch block, how can the host catch it to regain control of the thread? Inside the host's catch block, there is a call to Thread's ResetAbort method. Calling this method tells the CLR to stop rethrowing the ThreadAbortException at the end of each catch block.

This raises yet another question: What's to stop the untrusted code from catching the Thread-AbortException and calling Thread's ResetAbort method itself to keep control of the thread? The answer is that Thread's ResetAbort method requires the caller to have the SecurityPermission with the ControlThread flag set to true. When the host creates the AppDomain for the untrusted code, the host will not grant this permission, and now, the untrusted code cannot keep control of the host's thread.

I should point out that there is still a potential hole in this story: while the thread is unwinding from its ThreadAbortException, the untrusted code can execute catch and finally blocks. Inside these blocks, the untrusted code could enter an infinite loop, preventing the host from regaining control of its thread. A host application fixes this problem by setting an escalation policy (discussed earlier). If an aborting thread doesn't finish in a reasonable amount of time, the CLR can upgrade the thread abort to a rude thread abort, a rude AppDomain unload, disabling of the CLR, or killing of the process. I should also note that the untrusted code could catch the ThreadAbortException and, inside the catch block, throw some other kind of exception. If this other exception is caught, at the end of the catch block, the CLR automatically rethrows the ThreadAbortException.

It should be noted, though, that most untrusted code is not actually intended to be malicious; it is just written in such a way so as to be taking too long by the host's standards. Usually, `catch` and `finally` blocks contain very little code, and this code usually executes quickly without any infinite loops or long-running tasks. And so it is very unlikely that the escalation policy will have to go into effect for the host to regain control of its thread.

By the way, the `Thread` class actually offers two `Abort` methods: one takes no parameters, and the other takes an `Object` parameter allowing you to pass anything. When code catches the `Thread-AbortException`, it can query its read-only `ExceptionState` property. This property returns the object that was passed to `Abort`. This allows the thread calling `Abort` to specify some additional information that can be examined by code catching the `ThreadAbortException`. The host can use this to let its own handling code know why it is aborting threads.