4 *The CLR (Common Language Runtime)*

The CLR (*Common Language Runtime*) is the central element to the .NET platform architecture. The CLR is a software layer which manages the execution of .NET application code. The word 'manage' covers a wide range of operations needed for the execution of managed applications. Here are a few:

- Hosting of multiple applications in a single *Windows* process;
- Compilation of IL code into machine language code;
- Exception management;
- Destruction of unused objects;
- Loading of applications and assemblies;
- Resolving of types.

The CLR is conceptually close to what is commonly known as the *virtual machine* in Java.

Application Domains (AppDomain)

Introduction

An application domain (which we will commonly refer to as an *AppDomain*) can be seen as a *light-weight process*. A *Windows* process can contain many AppDomains. The notion of AppDomain is used so that a physical server can host several applications. For example, ASP.NET takes advantage of AppDomains to host, within a single process, a multitude of web applications. In fact, the *Microsoft* stress tests create up to 1000 simple web applications within a single process. The performance gain from the use of AppDomains is two fold:

- The creation of an AppDomain requires less system resources than the creation of a *Windows* process.
- AppDomains hosted in the same *Windows* process share the same resources such as the CLR, the base .NET types, the addressing space and threads.

When an assembly is executed, the CLR automatically creates a *default AppDomain* for the execution. Each AppDomain has a name and the default AppDomain has the name of the executed assembly (including the `.exe` extension).

If a single assembly is loaded within multiple AppDomains, there are two possible behaviors:

- Either the CLR will load the assembly multiple times, once for each AppDomain in the process.
- Either the CLR will load the assembly a single time outside of all the AppDomains in the process. The assembly can then be used by all the AppDomains within the process. Such an assembly is said to be *domain neutral*.

We'll see a little later in this chapter that the choice of this behavior can be configured. The default behavior being to load the assembly multiple times as needed.

Threads and AppDomains

Do not confuse the notions of 'threads' and 'application domains'.

There is no notion of ownership between the threads of a process and the AppDomains within this process. Let us remind you that this is not the case for processes since the threads belong to a single process and that each process can have one or multiple threads. In reality, a thread is not confined to a single AppDomain and that at any given time, multiple threads can execute within the context of the same AppDomain.

Let there be two AppDomains `DA` and `DB` hosted within the same process. Suppose that a method from object `A`, for which the assembly is contained in `DA`, calls a method of object `B`, where the assembly resides in `DB`. In this case, the same thread will execute both the caller and the called methods. This thread will essentially cross the boundary between the `DA` and `DB` AppDomains.

In other words, the concepts of threads and AppDomains are orthogonal.

Unloading an AppDomain

Once an assembly is loaded inside an AppDomain, you cannot unload it from this AppDomain. However, you can unload an AppDomain as a whole. This operation has some big consequences since the threads which are currently executing within the AppDomain must be aborted by the CLR and problems can occur when some of these are executing non-managed code. All managed objects within the application domain must also be garbage collected.

We recommend against having an architecture which depends on the frequent load/unload of AppDomains. We'll see that type of architecture is sometimes a necessary evil when implementing servers which require a high availability rate (99,999% of the time such as *SQL Server 2005*).

AppDomains and isolation

Isolation between AppDomains comes from the following characteristics:

- An AppDomain can be unloaded independently from other AppDomains.
- An AppDomain does not have access to assemblies and objects from other AppDomains.
- An AppDomain can have its own exception management strategy as long as it does not let an exception exit from its bounds meaning that problems from within an AppDomain will not affect the other domains within the same process.
- Each AppDomain can define its own security strategy for code access to an assembly.
- Each AppDomain can define its own rules in regard to how the CLR will locate the assemblies before loading them.

The System.AppDomain class

An instance of the `System.AppDomain` class is a reference to an AppDomain within the current process. The static property `CurrentDomain{get;}` of this class allows you to get a reference to the current AppDomain. The following example illustrates the use of this class to enumerate the assemblies loaded in the current AppDomain:

Example 4-1

```
using System;
using System.Reflection; // For the Assembly class.
class Program {
   static void Main() {
      AppDomain curAppDomain = AppDomain.CurrentDomain;
      foreach ( Assembly assembly in curAppDomain.GetAssemblies() )
         Console.WriteLine( assembly.FullName );
   }
}
```

Hosting several applications in a single process

The `AppDomain` class contains the `CreateDomain()` static method that allows the creation of a new AppDomain within the current process. This method offers multiple overloaded variations. To use this method, you must specify the following:

- (mandatory) A name for your AppDomain;
- (optional) The security rules for the CAS on this new AppDomain (with an object of type `System.Security.Policy.Evidence`);
- (optional) Information that allows the location mechanism of the CLR for this AppDomain (with an object of type `System.AppDomainSetup`).

The two important properties of a `System.AppDomainSetup` are:

- `ApplicationBase`: This property defines the base directory for the AppDomain. This directory is used by the assembly locating mechanism of the CLR when loading assemblies into this AppDomain.
- `ConfigurationFile`: This property references a configuration file for the AppDomain. This file is a XML file containing information about versioning and locating rules to be used in the AppDomain.

Now that you know how to create an application domain, we can present to you how you can load and execute an assembly within this domain by calling the `System.AppDomain.ExecuteAssembly()` method. The assembly must be an executable and the flow of execution will start at its entry point. Note that the thread calling `ExecuteAssembly()` will be executing the loaded assembly. This illustrates how a thread can cross the boundaries between AppDomains.

Here is a C# example. The first piece of code is the assembly which will be loaded by the assembly defined in the second segment of code:

Example 4-2 *AssemblyToLoad.exe*

```
using System;
using System.Threading;
public class Program {
   public static void Main() {
      Console.WriteLine(
         “Thread:{0} Hi from the domain: {1}”,
         Thread.CurrentThread.Name,
         AppDomain.CurrentDomain.FriendlyName);
   }
}
```

Example 4-3 *AssemblyLoader.exe*

```
using System;
using System.Threading;
public class Program {
   public static void Main() {
      // Name the current thread.
      Thread.CurrentThread.Name = “MyThread”;
      // Create an AppDomainSetup instance.
      AppDomainSetup info = new AppDomainSetup();
      info.ApplicationBase = “file:///”+ Environment.CurrentDirectory;
      // Create a new appdomain without security parameters.
      AppDomain newDomain = AppDomain.CreateDomain(
               “NewDomain”, null, info);
      Console.WriteLine(
```

Example 4-3 *AssemblyLoader.exe*

```
            "Thread:{0} Calling  ExecuteAssembly() from appdomain {1}",
            Thread.CurrentThread.Name,
            AppDomain.CurrentDomain.FriendlyName );
         // Load the assembly 'AssemblyACharger.exe' inside
         // 'NewDomain' and then execute it.
         newDomain.ExecuteAssembly( "AssemblyToLoad.exe" );
         // Unload the new domain.
         AppDomain.Unload( newDomain );
      }
   }
```

This example will display the following:

```
Thread:MyThread Calling ExecuteAssembly() from appdomain AssemblyLoader.exe
Thread:MyThread Hi from the domain: NewDomain
```

Note the need to add "`file:///`" to specify that the assembly is located on a local storage device. If this was not the case, we could have used "`http:///`" to load the assembly from the web.

This example also illustrates that the default name for an AppDomain is the name of the main module from the launched assembly (in this case, `AssemblyLoader.exe`).

Running some code inside the context of another AppDomain

Thanks to the instance method `AppDomain.DoCallBack()` you have the possibility of executing code of the assembly within the current AppDomain in the context of another application domain. For this, the code must be contained within a method which will be referenced through a `System.CrossAppDomainDelegate` delegate. The process is illustrated with the following example:

Example 4-4

```
using System;
using System.Threading;
public class Program {
   public static void Main() {
      Thread.CurrentThread.Name = "MyThread";
      AppDomain newDomain = AppDomain.CreateDomain( "NewDomain" );
      CrossAppDomainDelegate deleg = new CrossAppDomainDelegate(Fct);
      newDomain.DoCallBack(deleg);
      AppDomain.Unload( newDomain );
   }
   public static void Fct() {
      Console.WriteLine(
         "Thread:{0} execute Fct() inside the appdomain {1}",
         Thread.CurrentThread.Name,
         AppDomain.CurrentDomain.FriendlyName);
   }
}
```

This example displays the following:

```
Thread:MyThread execute Fct() inside the appdomain  NewDomain
```

Be aware that the possibility of 'injecting' code within an AppDomain can cause a security exception to be raised if you do not have the required security privileges.

Events of the AppDomain class

The `AppDomain` class exposes the following events:

Event.	Description
`AssemblyLoad`	Triggered when an assembly has been loaded.
`AssemblyResolve`	Triggered when an assembly to be loaded cannot be found.
`DomainUnload`	Triggered when the AppDomain is about to be unloaded.
`ProcessExit`	Triggered when the process terminates (triggered before `DomainUnload`).
`ReflectionOnly-AssemblyResolve`	Triggered when the resolution of an assembly destined to be used by the reflection mechanism fails.
`ResourceResolve`	Triggered when a resource cannot be found.
`TypeResolve`	Triggered when a type cannot be found.
`UnhandledException`	Triggered when an exception is not handled by the code in the AppDomain.

Some of these events can be used to remedy to the source of the problem that has triggered the event. The following example illustrates how to use the `AssemblyResolve` event to allow an assembly to load from a location which was not initially specified to the CLR's location mechanism:

Example 4-5

```
using System;
using System.Reflection;  // For the Assembly class.
public class Program {
   public static void Main() {
      AppDomain.CurrentDomain.AssemblyResolve += AssemblyResolve;
      Assembly.Load("AssemblyToLoad.dll");
   }
   public static Assembly AssemblyResolve( object sender,
                                           ResolveEventArgs e ) {
      Console.WriteLine("Can't find assembly : {0}", e.Name);
      return Assembly.LoadFrom(@"C:\AppDir\ThisAssemblyToLoad.dll");
   }
}
```

If the second load attempt succeeds, no exception is thrown. The name of the assembly loaded on the second attempt isn't necessarily the same as the one we initially attempted to load.

At page 432, we'll show you a program taking advantage of the `UnhandledException` event.

At page 65, we explain that classes in the `System.Deployment` namespace can make use of the `AssemblyResolve` and `ResourceResolve` events to dynamically download a group of files when an application taking advantage of the *ClickOnce* technology is run for the first time.

Sharing information between the AppDomains of a same process

Taking advantage of the `SetData()` and `GetData()` methods of the `AppDomain` class, you can store data directly within an AppDomain. As shows the following example, data within the AppDomain is indexed based on a character string:

Example 4-6

```
using System;
using System.Threading;
public class Program {
   public static void Main() {
      AppDomain newDomain = AppDomain.CreateDomain("NewDomain");
      CrossAppDomainDelegate deleg = new CrossAppDomainDelegate(Fct);
      newDomain.DoCallBack(deleg);
      // Fetch from the new appdomain the data named 'AnInteger'.
      int anInteger = (int) newDomain.GetData("AnInteger");
      AppDomain.Unload(newDomain);
   }
   public static void Fct() {
      // This method is ran inside the appdomain 'NewDomain'. It creates
      // an appdomain data named 'AnInteger' which has the value '691'.
      AppDomain.CurrentDomain.SetData("AnInteger", 691);
   }
}
```

This example shows a simple case as we are storing an integer which is a type known by all AppDomains. The .NET *Remoting* technology (which is covered in chapter 22) allows us to share data between AppDomains in a more evolved yet complex way.

Loading the CLR inside a Windows process with the runtime host

mscorsvr.dll and mscorwks.dll

Each version of the CLR is published with two DLLs:

- The `mscorsvr.dll` DLL contains a version of the CLR specially optimized for multi-processor machines ('svr' is used for 'server').
- The `mscorwks.dll` DLL contains a version of the CLR optimized for machines with a single processor ('wks' is used for 'workstation').

These two DLLs are not assemblies and consequently do not contain any IL code (and cannot be analyzed with the `ildasm.exe` tool). Each process which executes one or many .NET application will contain one of these two DLLs. We say that the process is hosting the CLR. Let us explain how this DLL is loaded into a process.

The mscorlib.dll assembly

Another DLL that plays a crucial role in the execution of .NET applications is `mscorlib.dll` which is an assembly containing implementations of all base types in the .NET *framework* (such as `System.String` `System.Object` or `System.Int32`). This assembly is referenced by every other .NET assembly. This reference is automatically created by every compiler producing IL code. It is an interesting task to analyze the `mscorlib` assembly using the `ildasm.exe` tool. Let us precise that the `mscorlib` resides for execution outside of all AppDomains and thus can only be loaded/ unloaded once during the lifetime of a process.

Introduction to the runtime host

The fact that the CLR is not yet integrated into any operating system means that the loading of the CLR at the creation of a process must be handled by the *process* itself.

The task of loading the CLR into the process involves an entity called the *runtime host*. The runtime host being there to load the CLR, a portion if its code must be unmanaged since it is the CLR that will handle managed code. This portion of code takes care of loading the CLR, configuring it and then transferring the current thread into managed mode. Once the CLR is loaded, the runtime host has other responsibilities such dealing with untrapped exceptions. The illustration bellow illustrates the different layers of this architecture. As you can see, the CLR and the runtime host exchange data through an API:

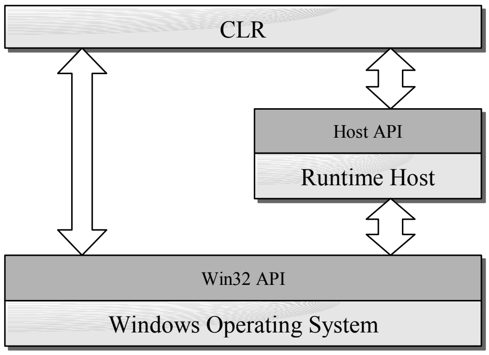

Figure 4-1: Hosting the CLR

There are several existing runtime hosts and you can even create your own. The choice of runtime host will have an impact on your application's performance and will define which functionalities are available to your application. The pre-existing runtime hosts provided by *Microsoft* are:

- The **Console** and **Winform** runtime host: The executed assembly is launched in the default AppDomain. When an assembly is implicitly loaded, it is loaded in the same domain as the referring assembly. Generally, this type of application does not need to use an AppDomain other than the default one.
- The **ASP.NET** runtime host: Create an AppDomain for each web application. A web application is identified by its ASP.NET virtual directory. If a web request is issued to an application that is already loaded, then the request is automatically routed to this AppDomain by the host.
- The **Microsoft Internet Explorer** runtime host: By default, this host creates one AppDomain for each web site visited. This allows each site to have different security levels. The CLR is only loaded once *Internet Explorer* needs to execute an assembly for the first time.
- The **SQL Server 2005** runtime host: Requests to the database can be written in IL code. The CLR is only loaded the first time that such a request must be executed. An AppDomain is created for every user/database pair. In the current chapter we will have the opportunity to come back to discuss the possibilities of this particular host that has been introduced in .NET 2.

Hosting several versions of the CLR on a single machine

The `mscorlib` assembly is strongly named meaning that many versions can co-exist side by side on the same machine. In addition, all versions of the CLR reside in the `'%windir%\Microsoft.NET\Framework'` directory. All files relating to a specific version of the .NET *framework* exist in a sub-folder named based on the version number. Since there is one directory for each version of the .NET *framework* installed on a machine, there can be multiple versions of both the `mscorsvr.dll` and `mscorwks.dll`. However, only a single version of the CLR can be loaded in each process.

The fact that you can have multiple versions of the CLR brings the existence of a small software layer which takes in as a parameter the desired version of the CLR and takes care of loading it. This code is called a *shim* and is stored in the `mscoree.dll` DLL (*MSCOREE* means *Microsoft Component Object Runtime Execution Engine*).

There can only be one shim DLL per machine and it is called through the runtime host via the `CorBindToRuntimeEx()` function. The `mscoree.dll` DLL contains COM interfaces and classes and `CorBindToRuntimeEx()` essentially creates an instance the `CorRuntimeHost` COM class for the requested version. It is through this object that you will interface with the CLR. To manipulate this object, the `CorBindToRuntimeEx()` function returns the `ICLRRuntimeHost` COM interface.

The call `CorBindToRuntimeEx()` used to create the COM objects interfacing with the CLR breaks a few fundamental COM rules: you must not call the `CoCreateInstance()` function to create your object. In addition, calling `AddRef()` and `Release()` on this interface has no effect.

Loading the CLR with the CorBindToRuntimeEx() function

Here is the prototype to the `CorBindToRuntimeEx()` which takes care of loading the shim DLL which in turns loads up the CLR:

```
HRESULT CorBindToRuntimeEx(
   LPWSTR      pwszVersion,
   LPWSTR      pwszBuildFlavor,
   DWORD       flags,
   REFCLSID    rclsid,
   REFIID      riid,
   LPVOID *    ppv);
```

This prototype is defined in the file `mscoree.h` and the code to this function is located in `mscoree.dll`.

- `PwszVersion`: Indicates the version of the CLR formatted as a string starting with the 'v' character (for example "`v2.0.50727`"). If this string is not defined (i.e. you passed null to the function), the most recent version of the CLR will be used.
- `PwszBuildFlavor`: This parameter indicates if you want to load the workstation (`mscorwks.dll`) version of the CLR by specifying the string "`wks`" or the server CLR (`mscorsvr.dll`) by specifying the string "`svr`". If you have a machine with a single processor the workstation version will be loader no matter which version you specify. *Microsoft* used a string parameter to allow future expansion with other new kinds of CLR.
- `flags`: This parameter is constructed from a set of various flags.

  By using the `STARTUP_CONCURRENT_GC` flag, you are indicating that you wish the *garbage collector* be executed in a concurrent mode. This mode allows a thread to complete a big part of the garbage collection work without interrupting the other threads of the application. In the case of a non-concurrent execution, the CLR often uses the application threads to execute garbage collection. The overall performance of the non-concurrent mode is better but may cause occasional lack of responsiveness from the user interface.

  We can also set other flags to indicate that we wish for assemblies to be loaded in a neutral way in regards to application domains or not. This means that all resources from the assembly will be present physically only once in the process even if multiple AppDomains load the same assembly (in the same spirit as the mapping of DLL across *Windows* applications). An assembly loaded neutrally does not belong to a specific AppDomain. This brings up the main disadvantage being that it cannot be unloaded without destroying the current process. However, the same set of security permissions must be used by each AppDomain or the assembly may be loaded multiple times. The constructor of each class in a neutrally loaded assembly is invoked for each AppDomain and a copy of every static field will exist for each domain. Internally, this is possible because of an indirection table that is managed by the CLR. This may result in a small performance loss because of the added indirection but considering the assembly will be loaded only a single time it will consume less system resources such as memory and may have some performance benefits as the assembly will only go through the JIT compiler once.

This means there is three possible flags:

STARTUP_LOADER_OPTIMIZATION_SINGLE_DOMAIN: No assemblies are loaded in a domain neutral fashion. This is the default behavior.

STARTUP_LOADER_OPTIMIZATION_MULTI_DOMAIN: All assemblies are loaded in a domain neutral fashion. No runtime hosting engine currently uses this option.

STARTUP_LOADER_OPTIMIZATION_MULTI_DOMAIN_HOST: Only shared assemblies found in the GAC are loaded domain neutral.

The mscorlib assembly is treated differently as it is always loaded neutrally.

- rclsid: The class ID (CLSID) of the COM class (coclass) that implements the CLR hosting interface desired. Only CLSID_CorRuntimeHost, CLSID_CLRRuntimeHost or null are valid values. The second value appeared with version 2.0 of .NET since new functionality was needed for the hosting of the CLR in *SQL Server 2005*.
- pwszBuildFlavor: The interface ID (IID) of the COM interface that you need. The only valid values are IID_CorRuntimeHost, IID_CLRRuntimeHost or null.
- ppv: A pointer towards the COM interface returned. This value is either of type ICorRuntimeHost or IClrRuntimeHost depending on what you have requested.

The shim only takes care of loading the CLR within the process. The lifecycle of the CLR is controlled by the unmanaged code part of the runtime host in the form of the ICLRRuntimeHost interface returned by the CorBindToRuntimeEx() function. This interface presents two methods, Start() and Stop() whose names are self-descriptive.

Creating a custom runtime host

In the majority of projects, you will not need to create your own runtime host. However, it is good to know how this can be done and it can be instructive to see how this is accomplished.

To properly understand the creation of a runtime host, you need a good understanding of the COM technology. Do not forget that the *Windows 2000/XP* operating systems still use COM as a standard communication mechanism. The CLR has the possibility of being manipulated using non-managed code via the use of COM interfaces. The ICCLRRuntimeHost plays a predominant role in accomplishing this and can be obtained through the CorBindToRuntimeEx() function. Here is the base C++ code for a runtime host:

Example 4-7

```
// You need to link with the static lib mscoree.lib
// to compile this C++ file.
#include <mscoree.h>

// The import must be done on a single line.
#import <mscorlib.tlb> raw_interfaces_only …
  ... high_property_prefixes(“_get”,”_put”,”_putref”) …
    ... rename(“ReportEvent”,”ReportEventManaged”) rename_namespace(“CRL”)

// We use the namespace ComRuntimeLibrary.
using namespace CRL;
ICLRRuntimeHost * pClrHost = NULL;
```

Example 4-7

```
void main (void){
   // Get a COM pointer of type ICorRuntimeHost on the CLR.
   HRESULT hr = CorBindToRuntimeEx(
               NULL, // We ask for the most recent version of the CLR.
               NULL, // We ask for the workstation version of the CLR.
               0,
               CLSID_CLRRuntimeHost,
               IID_ICLRRuntimeHost,
               (LPVOID *) &pClrHost);
   if (FAILED(hr)){
      printf("Can't get a pointer of type ICLRRuntimeHost!");
      return;
   }
   printf("We got a pointer of type ICCLRRuntimeHost.\n");
   printf("Launch the CLR.\n");
   pClrHost->Start();

   // Here, we can use our COM pointer pClrHost on the CLR.

   pClrHost->Stop();
   printf("CLR stopped.\n");
   pClrHost->Release();
   printf("Bye!\n");
}
```

Let us suppose that you have an assembly named `MyManagedLib.dll` which resides in the `C:\Test` directory and is built from the following code:

Example 4-8 *MyManagedLib.cs*

```
namespace MyProgramNamespace {
   public class MyClass {
      public static int MyMethod(string s) {
         System.Console.WriteLine(s);
         return 0;
      }
   }
}
```

You can easily invoke the `Main()` method from your host as follows:

Example 4-9

```
...
   pClrHost->Start();
   DWORD retVal=0;
   hr = pClrHost->ExecuteInDefaultAppDomain(
    L"C:\\test\\MyManagedLib.dll",// Path + Asm.
    L"MyProgramNamespace.MyClass",// Full name of the type.
    L"MyMethod",                  // Name of the method to run. It must
                                  // have the signature int XXX(string).
    L"Hello from host!",          // The argument string.
    &retVal);                     // A reference to the returned value.

   pClrHost->Stop();
...
```

Tuning the CLR from a custom runtime host

One thing you need to know is that when you control the CLR through a COM object, the `ICLRRuntimeHost` is not the only interface exposed by this COM object. In fact, you can manipulate the CLR through the following COM interfaces:

- `ICLRRuntimeHost` allows you to create and unload AppDomains, manage the lifecycle of the CLR and create evidences for the CAS security mechanism.
- `ICorConfiguration` allows you to specify to the CLR certain callback interfaces to be warned of specific events. These callback interfaces are: `IGCThreadControl` `IGCHostControl` `IDebuggerThreadControl`. Events presented by these interfaces are much less fine grained than those presented by the profiling API of the CLR which we will discuss a little later.
- `ICorThreadPool` allows you to manipulate the .NET thread pool for the process and allows the modification of certain configuration parameters.
- `IGCHost` allows you to obtain information on the behavior of the *garbage collector* and to control some of its configuration parameters.
- `IValidator` allows you to validate the PE/COFF header of assemblies (used by the `peverify.exe` tool).
- `IMetaDataConverter` allows the conversion of COM metadata (i.e. tlb/tlh) into .NET metadata (used by the `tlbexp.exe` tool).

To know which methods are exposed by these interfaces, you simply need to look at the files `mscoree.h`, `ivalidator.h` and `gchost.h`. To obtain one of these interfaces from your `IClrRuntimeHost` interface, you simply need to use the well known `QueryInterface()` method as follows:

```
...
   ICorThreadpool *   pThreadPool   = NULL;
   hr = pClrHost->QueryInterface( IID_ICorThreadpool,
                                      (void**)&pThreadPool);
...
```

Characteristics of the SQL Server 2005 runtime host

As we have mentioned previously, the runtime host for *SQL Server 2005* is special because of reliability, security and performance constraints imposed by RDBMS.

The main constraint for this type of server is reliability. The three mechanisms *constrained execution region* (CER), *critical finalizer* and *critical region* (CR) have been added to the CLR to reinforce the reliability of a .NET application. We discuss them further in the current chapter.

The second constraint is security. To avoid bad user code being loaded, all user assemblies are loaded by the runtime host from the database. This implies that an initial phase involving the pre-loading of assemblies in the database by the administrator must happen. During the phase, the administrator can specify that the assembly belongs to one of the categories `SAFE`, `EXTERNAL_ACCESS` and `UNSAFE` based on the level of trust towards the code in the assembly. The set of permissions allowed to an assembly by the CAS mechanism will depend on this level of trust. Finally, certain functionalities of the .NET *framework* considered sensitive such as certain classes of the `System.Threading` namespace, cannot be used from an assembly loaded in *SQL Server 2005*.

The third constraint is performance. It is satisfied by exploiting resources in an optimal manner. In this context, the most precious resources are threads and memory pages loaded in RAM. The idea is to minimize the number of context switches between threads and to minimize the number of memory pages stored on the virtual disk.

The *context switching* is generally managed by the *preemptive multitasking* of the *Windows scheduler* which we overview at page 113. The runtime host of *SQL Server 2005* implements its own multitasking mechanism based on a *cooperative multitasking* model. In this model, the threads themselves decide when the processor can move on to another thread. One advantage is that the choice of thread switching moments are more fine since tied to the semantics of processing. Another advantage is that this model is adapted to the use of *Windows fibers*.

A fiber is a *logical thread* which also qualifies as a *lightweight thread*. A same physical thread in *Windows* can chain the execution of several fibers. The advantage is that transfer from one fiber to another is much less expensive than a regular context switch. However, when the fiber mode is used by the *SQL Server 2005* runtime host we lose the guarantee of a good relationship between the physical *Windows* threads and the threads managed by .NET. Note that a same managed thread may not be executed by the same physical thread during its whole existence. So you must remove yourself from any affinity between these two entity types. Amongst the type of affinities between a managed and physical thread controlling it are the *Thread Local Storage*, the current culture and *Windows* synchronization objects which derive from the `WaitHandle` class, mutex, semaphores or events. Note that you can communicate to the runtime host the start and end of a region which exploit this kind of affinity using the `BeginThreadAffinity()` and the `EndThreadAffinity()` methods which are part of the `Thread` class. The use of the fiber mode generally brings a performance optimization of up to 20%. (Note: *In the current version of SQL Server 2005, the fiber mode has been removed as Microsoft engineers were unsure about its reliability. This mode will be reintroduced in later versions of this product*).

The storage of memory pages is normally managed by the *Windows* virtual memory system described at page 109. The runtime host of *SQL Server 2005* wedges itself between the memory request of the CLR and the operating system mechanisms in order to maximize the available RAM. This also allows us to obtain a more predictable behavior when a memory request to the CLR fails As we will see later, this feature is essential to ensuring the reliability of servers such as *SQL Server 2005*.

All these new functionalities are accessible through an API which allows the CLR and the host to communicate. About thirty new interfaces have been planned which are listed below. It is the host responsibility to supply an object which implements the `IHostControl` interface through the `ICLRRuntimeHost.SetHostControl(IHostControl*)` method. This interface presents the method `GetHostManager(IID,[out]obj)` that is called by the CLR to obtain an object from the host for which it will delegate a responsibility such as the management of threads or the loading of assemblies. More information is available by looking at the interfaces defined in the `mscoree.h` file.

Responsibility	Interfaces implemented by the host	Interfaces implemented by the CLR
Loading of assemblies	`IHostAssemblyManager` `IHostAssemblyStore`	`ICLRAssemblyReferenceList` `ICLRAssemblyIdentityManager`
Security	`IHostSecurityManager` `IHostSecurityContext`	`ICLRHostProtectionManager`
Exception management	`IHostPolicyManager`	`ICLRPolicyManager`
Memory management	`IHostMemoryManager` `IHostMalloc`	`ICLRMemoryNotification-Callback`
Garbage collector	`IHostGCManager`	`ICLRGCManager`
Threading	`IHostTaskManager` `IHostTask`	`ICLRTaskManager` `ICLRTask`
Thread pools	`IHostThreadPoolManager`	

Responsibility	Interfaces implemented by the host	Interfaces implemented by the CLR
Synchronization	`IHostSyncManager` `IHostCriticalSection` `IHostManualEvent` `IHostAutoEvent` `IHostSemaphore`	`ICLRSyncManager`
I/O Completion	`IHostIoCompletionManager`	`ICLRIoCompletionManager`
Debugging		`ICLRDebugManager`
CLR events	`IActionOnCLREvent`	`ICLROnEventManager`

Profiling the execution of your .NET applications

This section has the goal of familiarizing you with an extremely useful functionality of the CLR allowing you to finely profile its execution. In other words, you can ask the CLR to call one of your unmanaged methods when specific events occur such as the beginning of JIT compilation or the loading of a new assembly. It is logical that these callback methods be unmanaged since they are intended to let you observe the state of the CLR and obviously you would not want the CLR to handle them.

To take advantage of the *profiling* features of the CLR, you must construct a COM class which implements the `ICorProfilerCallback` interface. This interface is defined in the file `corprof.h` which can be found in the `SDK\v2.0\Include` directory of your *Visual Studio* installation. The interface contains about 70 methods. Once you have implemented the `ICorProfilerCallback` interface, you will need to let the CLR know that you wish to use your implementation for its callbacks. To communicate the CLSID of this implementation to the CLR, you do not need to create your own runtime host. In fact, you simply need to properly set the alues of the `Cor_Enable_Profiling` and `Cor_Profiler` variables. `Cor_Enable_Profiling` must be set to a non-null values to let the CLR know that it must perform the profiling callbacks. And `Cor_Profiler` must be set to the `CLSID` or the `ProgID` of your implementation of the `ICorProfilerCallback` interface.

We will not fully detail this feature as the article **The .NET Profiling API and the DNProfiling Tool** by *Matt Pietrek* of the *December 2001* **MSDN Magazine** (free online) already does this. We strongly suggest that you download the code in this article and experiment with it on your own .NET applications. This will give you a much better idea of the extent of the work done by the CLR! The code in this article also demonstrates how you can use masks to only enable a specific subset of the callbacks. The necessary steps to use this code are easy:

- Register the COM class on your machine;
- open a command line window;
- set the proper environment variables using the supplies batch file;
- launch your application through a command line window.

Locating and loading assemblies

That we apply an assembly deployment model which is private or shared, it is the CLR which will locate and load assemblies for execution. More precisely, these tasks fall under the *assembly loader* of the CLR more commonly named *fusion*. The general idea is that the assembly locating process by the CLR is smart and configurable.

- The CLR is configurable in the sense that an administrator can easily move assemblies while ensuring that they can still be located by the CLR. Configurable also means that we can redirect the use of certain versions of an assembly towards another version (with two possible approaches, with a publisher policy assembly or using a configuration file as described a little later).

- The CLR is intelligent in the sense that when it cannot find an assembly in a folder, it applies an algorithm that allows it, for example, to probe in sub-folder with the same name as the assembly. The CLR is also intelligent in the sense that if an application who used to work now fails because of an assembly it cannot find anymore, it can simply rollback the offending changes.

These two constraints can be achieved through the use of the *fusion* location algorithm.

We have seen at page 11 that an assembly can be constructed from many files called modules. Here we will talk about the location process for an assembly, which is the process by which the main module of an assembly is located (the one which includes the manifest). Let us remind you that all the modules of the same assembly must be located in the same directory.

When does the CLR attempt to locate an assembly?

The location process of the CLR is often called upon when the code of an executing assembly needs to load another assembly. If the location fails, the process will issue a `System.IO.FileNotFoundException` exception. The location process is used when:

- One of the overloaded `AppDomain.Load()` is used to load an assembly into the AppDomain who called the method.
- When the CLR needs to implicitly load an assembly. This is described further in the next section when we will discuss how the CLR resolves types.

This algorithm is not used when:

- The `AppDomain.ExecuteAssembly()` method that we have discussed earlier in this chapter is called.
- When the static method `Assembly.LoadFrom()` is called.

Note that these methods do not fully adhere to the .NET philosophy as they take in a path to a file and the name of an assembly. It is recommended never to use `LoadFrom()` as it can be substituted with `Load()`. The use of the simple `AppDomain.ExecuteAssembly()` can prove to be an efficient shortcut under the right circumstances.

The location algorithm used by the CLR

The GAC

The algorithm will first look for the assembly in the GAC directory, as long as the name of the assembly is a strong name. During the search in the GAC, the user of the application can choose (through a configuration file) to use the publisher policy assembly relative to the assembly that needs to be located.

The CodeBase element

If the strong name for an assembly is not properly supplied or that the assembly cannot be found in the GAC, there is a possibility that the assembly will need to be loaded from a *URL* (*Unique Resource Locator*). This possibility is at the base of the *No Touch Deployment* mechanism described at page 68.

In fact, the configuration file for the application can have a `<codebase>` relative to an assembly to be located. In this case, this element defines a URL and will attempt to load the assembly from that location. If the assembly cannot be found at the specified URL, then the location process will fail. If the assembly to be found is defined with a strong name within the configuration file then the URL can be an internet address, an intranet address or a folder on the current machine. If the assembly to be found is not defined using a strong name, the URL can only be a sub-directory of where the application is located.

Here is an extract from the configuration file of an application making use of the `<codebase>` element:

```
...
     <dependentAssembly>
        <assemblyIdentity name=”Foo3” publicKeyToken=”C64B742BD612D74A”
         culture= “en-US”/>
        <codebase version=”3.0.0.0”
         href = “http://www.smacchia.com/Foo3.dll>”/>
     </dependentAssembly>
...
```

Take note that the URL contains the location of the module of the assembly containing the manifest (in this case `Foo3.dll`). If the assembly has other modules, understand that all of these modules must be downloaded from the same location (which is `http://www.smacchia.com/`).

When an assembly is downloaded from the web by using the `<codebase>` configuration element, it is stored in the download cache. Also, before you attempt to load an assembly from a URL, the *fusion* will check this cache to see if the module has already been downloaded.

The probing mechanism

If the strong name for an assembly is not properly supplied or the assembly cannot be found in the GAC folder and that the configuration file does not define a `<codebase>` for this assembly, then the locating algorithm will attempt to find the assembly by probing certain folders. The following example exposes this *probing* mechanism:

- Let’s suppose that we wish to locate an assembly named `Foo` (note that we do not supply the file extension in the name).
- And also, let us assume that a configuration file element named `<probing>` defines, for the `Foo` assembly, the following paths “`Path1`” and “`Path2\Bin`”.
- The application folder is located at “`C:\AppDir\`”.
- And finally, let’s assume that that no culture information was supplied with the assembly.

The search for the assembly will be accomplished in the following order:

```
C:\AppDir\Foo.dll
C:\AppDir\Foo\Foo.dll
C:\AppDir\Path1\Foo.dll
C:\AppDir\Path1\Foo\Foo.dll
C:\AppDir\Path2\Bin\Foo.dll
C:\AppDir\Path2\Bin\Foo\Foo.dll
C:\AppDir\Foo.exe
C:\AppDir\Foo\Foo.exe
C:\AppDir\Path1\Foo.exe
C:\AppDir\Path1\Foo\Foo.exe
C:\AppDir\Path2\Bin\Foo.exe
C:\AppDir\Path2\Bin\Foo\Foo.exe
```

If the assembly is a satellite assembly (i.e. it has a non-neutral culture, for example “`fr-FR`”) the folder search occurs in the following order:

```
C:\AppDir\fr-FR\Foo.dll
C:\AppDir\fr-FR\Foo\Foo.dll
C:\AppDir\Path1\fr-FR\Foo.dll
C:\AppDir\Path1\fr-FR\Foo\Foo.dll
C:\AppDir\Path2\Bin\fr-FR\Foo.dll
C:\AppDir\Path2\Bin\fr-FR\Foo\Foo.dll
```

```
C:\AppDir\fr-FR\Foo.exe
C:\AppDir\fr-FR\Foo\Foo.exe
C:\AppDir\Path1\fr-FR\Foo.exe
C:\AppDir\Path1\fr-FR\Foo\Foo.exe
C:\AppDir\Path2\Bin\fr-FR\Foo.exe
C:\AppDir\Path2\Bin\fr-FR\Foo\Foo.exe
```

The AppDomain.AssemblyResolve event

Finally, if the assembly is not found with all these steps, the CLR will trigger the `AssemblyResolve` event of the `AppDomain` class. The methods delegated to this event can return an object of type `Assembly`. This allows you to implement your own location mechanism for the assemblies. This features is notably used within the *ClickOnce* deployment technology to download group of files on-demand as discussed at page 65.

You can use the `fuslogvw.exe` tool to analyze the logs produced by the *fusion* mechanism. This tool is useful to determine the exact causes of failures when loading an assembly.

The <assemblyBinding> element of the configuration file

The configuration file for an `ASM` assembly can contain parameters used by the *fusion* mechanism when `ASM` triggers the location and loading of another assembly. The configuration file can contain a `<probing>` element which specifies one or many folders that the probing mechanism must look at. The file can also contain a `<dependentAssembly>` element for each assembly to be potentially located. Each of these `<dependentAssembly>` elements can contain the following information on how an assembly can be located:

- The `<publisherPolicy>` element determines if the eventual publisher policies for the assembly to be loaded need to be taken into account when locating the assembly.
- The `<codebase>`, element described in the previous section defines the URL from which the assembly must be downloaded from.
- The `<bindingRedirect>` element allows you to redirect the version number used in the same way as a publisher policy assembly. The publisher policy applies as to all client applications of a shared assembly but here only the application targeted by the configuration file is affected.

Here is what a configuration file may look like (notice the similarity with a publisher policy assembly XML file):

Diagram of the location algorithm

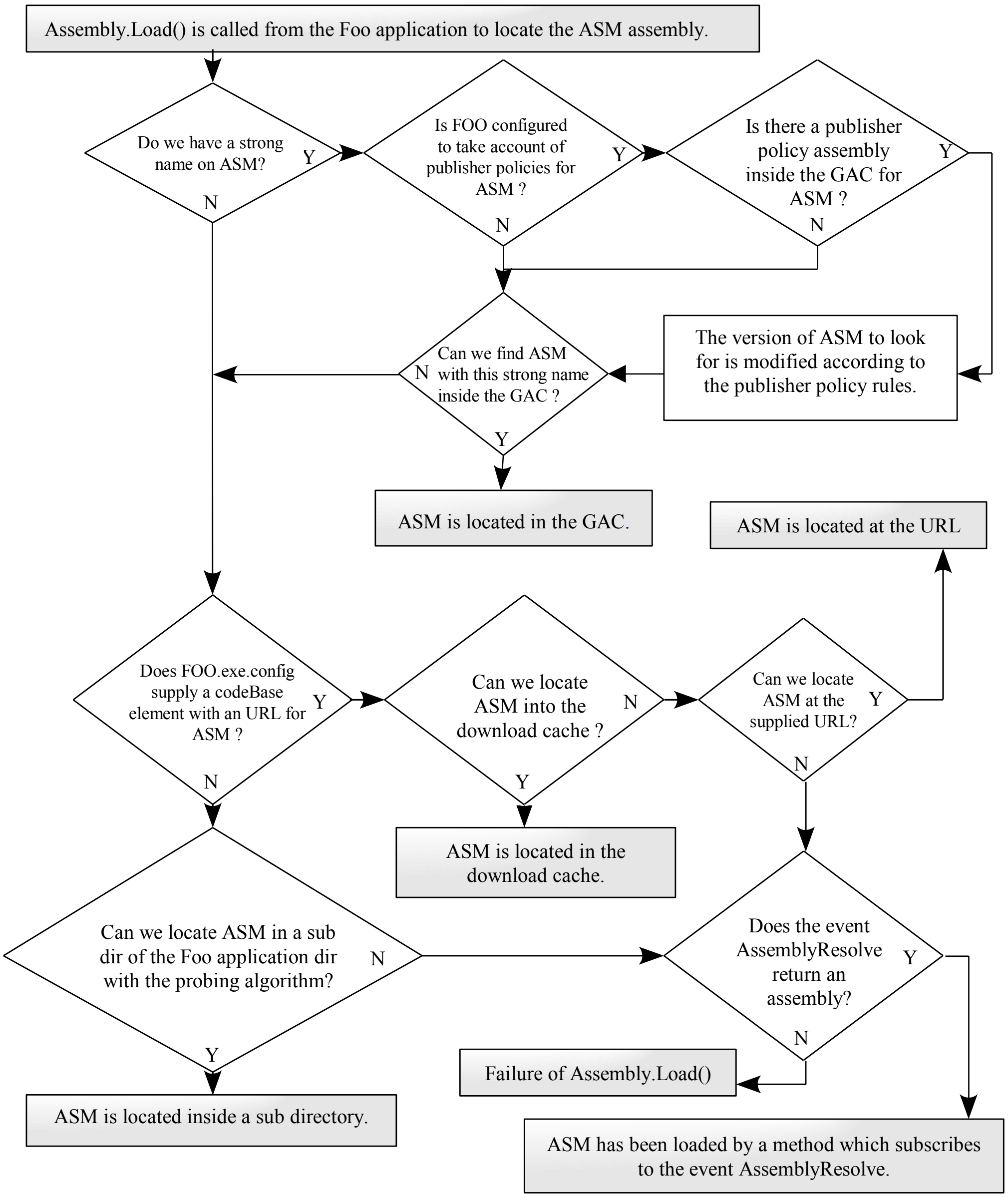
Assembly.Load() is called from the Foo application to locate the ASM assembly.
Do we have a strong name on ASM?
Y
N
Is FOO configured to take account of publisher policies for ASM ?
Y
N
Is there a publisher policy assembly inside the GAC for ASM ?
Y
N
Can we find ASM with this strong name inside the GAC ?
N
Y
The version of ASM to look for is modified according to the publisher policy rules.
ASM is located in the GAC.
ASM is located at the URL
Does FOO.exe.config supply a codeBase element with an URL for ASM ?
Y
N
Can we locate ASM into the download cache ?
N
Y
Can we locate ASM at the supplied URL?
Y
N
ASM is located in the download cache.
Can we locate ASM in a sub dir of the Foo application dir with the probing algorithm?
N
Y
Does the event AssemblyResolve return an assembly?
Y
N
Failure of Assembly.Load()
ASM is located inside a sub directory.
ASM has been loaded by a method which subscribes to the event AssemblyResolve.

Figure 4-2:Diagram of the location algorithm

Example 4-10

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
         <probing privatePath="Path1;Path2/bin" />
         <dependentAssembly>
            <assemblyIdentity name="Foo1" culture= "neutral"
                              publicKeyToken="C64B742BD612D74A" />
            <publiherPolicy apply="no" />
            <bindingRedirect oldVersion="1.0.0.0" newVersion="2.0.0.0"/>
         </dependentAssembly>
         <dependentAssembly>
            <assemblyIdentity name="Foo2" culture= "neutral"
                              publicKeyToken="C64B742BD612D74A" />
            <codebase version="2.0.0.0"
                      href = "file:///C:/Code/Foo2.dll>"/>
         </dependentAssembly>
         <dependentAssembly>
            <assemblyIdentity name="Foo3" culture= "fr-FR"
                              publicKeyToken="C64B742BD612D74A" />
            <codebase version="3.0.0.0"
                      href = "http://www.smacchia.com/Foo3.dll>"/>
         </dependentAssembly>
      </assemblyBinding>
   </runtime>
</configuration>
```

If you do not wish to manipulate XML files, know that this information can be configured by using the `.NET Framework Configuration` tool accessible from: *Start menu › Configuration panel › Administrative tools › Microsoft .NET Framework 2.0 Configuration › configured assembly* menu.

The Shadow Copy mechanism

When an assembly is loaded by a process, the corresponding files are automatically locked by *Windows*. This means that you cannot update or delete the file and you can only rename it. In the case of an ASP.NET server a process can potentially host multiple applications. This behavior could be cumbersome since the process would need to be restarted each time an application needs to be updated.

Fortunately, the assembly loaded of the CLR features a mechanism called *shadow copy*. This mechanism copies assembly files in a dedicated cache folder for an application before actually loading the assembly. Hence the original files can be updated while they are being executed. ASP.NET uses this feature and periodically verifies if a loaded assembly has been updated. If it is the case, the application is restarted without loosing any requests.

The `string ShadowCopyDirectories{get;set;}` property of the `AppDomainSetup` class allows you to specify the folders which will contain the shadow copy of your assemblies.

Resolving types at runtime

Loading an assembly explicitly or implicitly

We often need to use within our code types which have been declared in other assemblies. There are two techniques to use types defined in other assemblies:

- You can rely on your compiler to early bind with the type and you rely on the CLR to load the assembly automatically when needed. This technique is called *implicit loading*.
- You can also *explicitly load* the assembly and thus late bind to the type you need to use.

In both cases, it is the responsibility of a portion of the CLR named *class loader*. The implicit loading of an assembly `A` is triggered at the first JIT compilation of a method using a type defined in `A`.

The notion of implicit loading relates to the DLL loading mechanism. In a same way, the notion of explicit loading resembles the mechanism used with COM objects through *Automation* and the well known *IDispatch* interface.

Referencing an assembly at compile time

When assembly `A` executes and the JIT compiles a method of `A` which requires a type defined in the `B` assembly, the `B` assembly is implicitly loaded by the CLR if it was referenced during the compilation process for `A`. For `A` to reference `B`, one of two operations must have been completed during the compilation process:

- Either you compiled `A` using the `csc.exe` command line compiler directly or through a compilation script. You must then use the proper `/reference /r` and `/lib` compiler options.
- Either you use *Visual Studio* with the *Reference › AddReference* menu option to set the proper references. Let us remind you that the *Visual Studio* environment implicitly uses the `csc.exe` compiler to generate assemblies from your C# source code. Setting the environment options essentially forces *Visual Studio* to use the `/reference /r` and `/lib` options of the `csc.exe` compiler.

In both cases, you must specify the assembly to load by its name (strong or not). The types as well as members within the `B` assembly that are referenced by the `A` assembly are in the *TypeRef* and *MemberRef* tables of `A`. In addition, the `B` assembly is also referenced in the *AssemblyRef* table of assembly `A`. An assembly can reference many other assemblies but you must avoid at all cost cyclic references (`A` references `B` which reference `C` which in turn references `A`). *Visual Studio* will detect and prevent cyclic reference. You can also use the *NDepend* tool (described at page 857) to detect cyclic assembly references.

An example

Here is an example of the whole infrastructure put in place so that the CLR can implicitly load an assembly. The first piece of code defines the assembly referenced by the second piece of code:

Example 4-11 *Code of the referenced assembly: AsmLibrary.cs*

```
using System;
namespace MyTypes {
   public class FooClass {
      public static int Sum(int a,int b){return a+b;}
   }
}
```

Example 4-12 *Code of the referencing assembly: AsmExecutable.cs*

```
using System;
using MyTypes;
class Program {
   static void Main() {
      int i = FooClass.Sum(3,4);
   }
}
```

Note the use of the `MyTypes` namespace in the referencing assembly. We could have also put the `FooClass` and `Program` classes in the same namespace or in the anonymous namespace. In this case, we could have illustrated that namespaces can spawn multiple assemblies.

It is interesting to analyze the manifest of the referencing assembly using the `ildasm.exe` tool. You can clearly see the fact that the assembly '`AsmLibrary`' is referenced. This reference takes form as an entry in the *AssemblyRef* table.

```
...
.assembly extern 'AsmLibrary'{
  .ver 0:0:0:0
}
...
```

It is also interesting to take a look at the IL code for the `Main()` method. You can clearly see that the method named `Sum()` is located in another assembly named '`AsmLibrary`':

```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       9 (0x9)
  .maxstack  2
  .locals init (int32 V_0)
  IL_0000:  ldc.i4.3
  IL_0001:  ldc.i4.4
  IL_0002:  call       int32
     ['AsmLibrary']MyTypes.FooClass::Sum(int32,int32)
  IL_0007:  stloc.0
  IL_0008:  ret
} // end of method Program::Main
```

JIT compilation (Just In Time)

Portable binary code

Let us remind you that .NET applications, independently of the source code language, are compiled into IL code. The IL code itself is stored in a special section of the assembly and modules. It stays in this form until the moment where the code needs to be executed.

As it name implies, the IL code (*Intermediate Language*) is an intermediate form of code between the high-level .NET languages (C#, VB.NET…) and machine code. The idea is that the compilation from IL code to a machine native form takes place during the execution of the application. This allows .NET applications to be distributed in a machine (and OS) independent form. The major point being that to distribute a .NET application across multiple operating systems, there is no need to recompile the high-level code. Hence we can say the .NET applications are *portable binary code*.

Introduction to Just In Time compilation

The compilation of IL code into machine language happens during the execution of the application. We could imagine two possible scenarios in regards to the compilation of the IL code:

- The application is completely compiled into language machine when it is launched.
- The IL instructions are interpreted one by one in the same way as an interpreted language.

None of these scenarios is used for the compilation of the IL code into machine language. An intermediate solution which is more efficient has been implemented. With this solution, the body of a method in IL language is compiled into native machine code just before the first call to the method. This is why the run-time compilation process is called *Just In Time* (*JIT*). The compilation to machine language occurs just in time for the execution of the method to occur.

Diagram of the types resolving algorithm

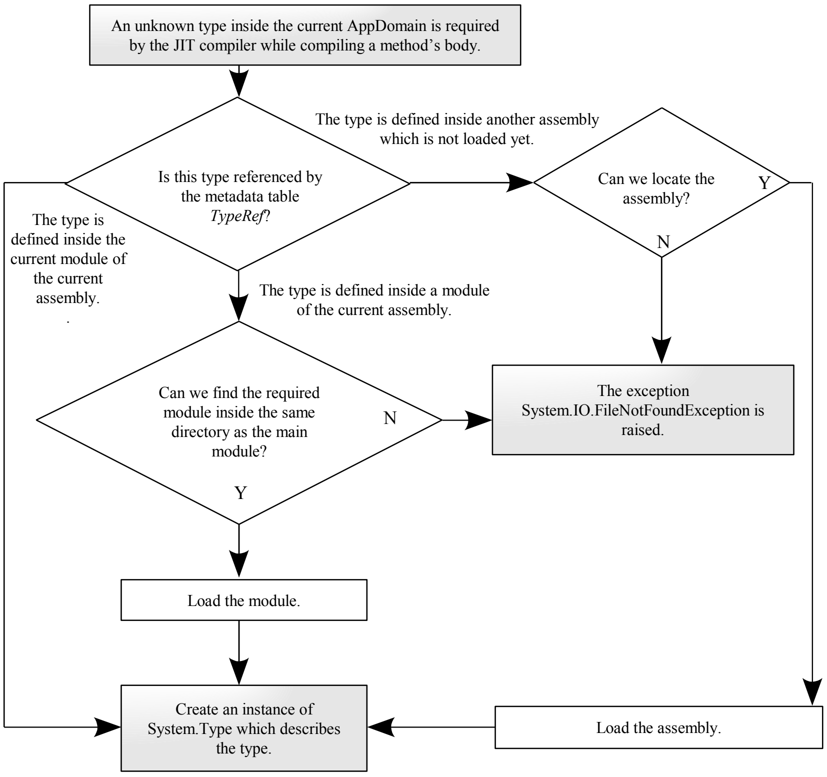
An unknown type inside the current AppDomain is required by the JIT compiler while compiling a method's body.
Is this type referenced by the metadata table *TypeRef*?
The type is defined inside another assembly which is not loaded yet.
Can we locate the assembly?
Y
N
The type is defined inside the current module of the current assembly.
The type is defined inside a module of the current assembly.
Can we find the required module inside the same directory as the main module?
N
Y
The exception System.IO.FileNotFoundException is raised.
Load the module.
Create an instance of System.Type which describes the type.
Load the assembly.

Figure 4-3 : Resolving a type

Intercepting the first call with a stub

To perform JIT compilation, the CLR uses a trick commonly used in distributed architectures such as *RPC*, *DCOM .NET Remoting* or *Corba*. Each time a class is loaded by the CLR, a *stub* is associated to each method in the class. In a distributed architecture, the stub is a small software layer on the client which intercepts method calls to convert them into remote calls (with *DCOM* the term proxy is used). In the case of the JIT, the stub is a layer of code which will intercept the first call to the method.

This stub code essentially initiates the JIT compiler which will verify the IL code and translate into machine native instructions. The native code is then stored in the memory space of the process and then executed. Of course, the stub is then replaced with a jump to the memory address of the freshly generated code, ensuring that each method is only compiled once.

Vérification of the IL code

Before a method is compiled into native machine language, the JIT compiler does a series of validations to ensure the body of the method is valid. To determine if a method is valid, the compiler checks the flow of IL instructions and verifies the content of stack to detect invalid memory accesses. If the code fails the verification, the JIT compiler will issue an exception.

Your C# code written in safe mode is automatically translated into verifiable IL code. However, at page 417 we show that under certain circumstances, the C# compiler can produce special IL instructions which are not verifiable by the JIT compiler.

Optimisations done by the JIT compiler

The JIT compiler will perform optimizations on your code. Here are a few examples to give you an idea:

- To avoid the cost of passing parameters around, the JIT compiler can insert the body of a method into the body of the calling method. This optimization is called *inlining*. So that the cost of the optimization does not exceed the performance gains, the inlined method must meet certain conditions. The body of the function once compiled must not exceed 32 bytes, it must not catch exceptions, it must not contain loops and it cannot be a virtual function.
- The JIT compiler can set any local variable that is a reference to `null` once it is last used. This reduces the reference count of this object and can then give a chance that it will be garbage collected early, leaving more resources for other objects. Note that this optimization is locally deactivated when using the `System.GC.KeepAlive()` method.
- The JIT compiler can store frequently used local variables directly into the processor's registers instead of using the stack. This constitutes an optimization as access to registers is significantly more efficient than stack access. This optimization is commonly named *enregistration*.

Introduction to pitching

The compilation of methods by the JIT consumes memory since the native code for the methods must be stored. If the CLR detects that memory is running low for the application, it has the possibility of recuperating some memory by freeing the native code for certain methods. Naturally, a stub will be regenerated for these methods. This functionally is called *pitching*.

The implementation of pitching is more complex than is seems. To be efficient, the freed code must be in contiguous memory to avoid memory fragmentation. Other important problems appear in the thread stacks which contain memory addresses referencing the native code. Fortunately, all these complex consideration are hidden to the developer.

The native code of a method generated by the JIT will be destroyed when the AppDomain of its application is unloaded

JIT and JITA acronyms

For those coming from the world of COM/DCOM/COM+ the JIT acronym may remind you of the *JITA* (*Just In Time Activation*, described at page 245) acronym. The two mechanisms are different, have different goals and are essentially used in different technologies However, the underlying idea is the same: we mobilize resources for an entity (compilation of an IL method for JIT and activation of a COM object for JITA) only when this entity is needed (i.e. just before it is used). We often use the *lazy* adjective for such type of mechanisms.

The ngen.exe tool

Microsoft supplies a tool called `ngen.exe` which is capable of compiling assemblies before their execution. This tool functions as a standard compiler and replaces the JIT mechanism. The real name for `ngen.exe` is '*Native Image Generator*'. The `ngen.exe` tool is to be used when you notice that the JIT mechanism introduces a significant drop in performance, generally during the startup of an application. However, note that the regular compiler has access to a greater number of possible optimizations than `ngen.exe`. The use of the JIT compiler is most often preferred.

Compilation using `ngen.exe` is generally done during the installation of an application. You do not need to manipulate the new files containing the native code, called *native images*. They are automatically stored in a special directory on your machine called *native image cache*. This folder is accessible from `ngen.exe` using the `/show` and `/delete` options. This folder is also visible when a user visualizes the *global assembly cache* as the native images are included in this folder. This visualization is described at page 52.

`ngen.exe` **option**	**Description**
`/show` `[assembly name |folder name]`	Allows you to visualize a list of native images in the 'Native Image Cache'. If you follow this option with the name of an assembly, you will only see images with the same name, If you follow this option with the name of a folder, you will only see the images located in this folder.
`/delete` `[assembly name | folder name]`	Deletes all images contained in the 'Native Image Cache'. If you follow this option with the name of an assembly, only images with the same name will be deleted. If you follow this option with the name of a folder, only images in this folder will be deleted.
`/debug`	Produces an image which can be used by a debugger.

Many other options exist to this compiler and you can find a complete listing on MSDN in the entry Native Image Generator (Ngen.exe). Notably, some new features have been added with .NET 2.0 to support assemblies taking advantage of reflection and to automate the update of the compiled version of an assembly when dependencies have changed. For more information, take a look online in the article named NGen Revs Up Your Performance with Powerful New Features by Reid Wilkes in the April 2005 issue of **MSDN Magazine**.

Performance counters and JIT compilation

You have access to the following six performance counters of the JIT in the `".NET CLR Jit"` category.

Name of the counter to be supplied as a string.	**Description**
`"# of IL Methods JITted"`	Counts the number of compiled methods. This number does not include the number of methods compiled by `ngen.exe`.
`"# of IL Bytes JITted"`	Counts the number of IL code bytes compiled. Pitched methods are not removed from this total.
`"Total # of IL Bytes Jitted"`	Counts the number of IL code bytes compiled with pitched methods accounted for in the total.
`"% Time in Jit"`	Percentage of time spent in the JIT compiler. This counter is updated at the end of each JIT compilation.
`"IL Bytes Jitted / sec"`	The average number of IL code compiled per second.
`"Standard Jit Failures"`	Number of times a method was determined as being non-valid and was not compiled by the JIT.

You have the choice to see the counters for all managed applications executed since the machine has booted up. This choice is done with the argument to the `PerformanceCounterCategory.GetCounters()` method. In the first case, you will need to supply the character string "`_Global_`" as an argument to this method. In the second case, you will need to supply a string equal to the name of the process you wish to observe as an argument to this method.

These counters are particularly useful to evaluate the cost of the JIT compilation process. If this cost appears too high you may need to consider the use of the `ngen.exe` tool when deploying your application. Here is an example illustrating the use of the performance counters (note that the name of the assembly is `MyAssembly.exe`):

Example 4-13 *MyAssembly.cs*

```
using System.Diagnostics;
class Program {
   static void DisplayJITCounters() {
      PerformanceCounterCategory perfCategory
                 = new PerformanceCounterCategory(".NET CLR Jit");
      PerformanceCounter[] perfCounters;
      perfCounters = perfCategory.GetCounters("MyAssembly");
      foreach(PerformanceCounter perfCounter in perfCounters)
         System.Console.WriteLine("{0}:{1}",
            perfCounter.CounterName,
            perfCounter.NextValue());
   }
   static void f() {
      System.Console.WriteLine("----> Calling  f().");
   }
   static void Main() {
      DisplayJITCounters();
      f();
      DisplayJITCounters();
   }
}
```

This program displays:

```
# of Methods Jitted:2
# of IL Bytes Jitted:108
Total # of IL Bytes Jitted:108
IL Bytes Jitted / sec:0
Standard Jit Failures:0
% Time in Jit:0
Not Displayed:0
----> Calling f().
# of Methods Jitted:3
# of IL Bytes Jitted:120
Total # of IL Bytes Jitted:120
IL Bytes Jitted / sec:0
Standard Jit Failures:0
% Time in Jit:0,02865955
Not Displayed:0
```

Let us precise that the performance counters are also available within the `perfmon.exe` tool (accessible from *Start menu › Run… › perfmon.exe*).

The garbage collector (GC) and the managed heap

Introduction to garbage collecting

In the .NET languages, the destruction of managed objects is not the responsibility of the programmer. The problem with the destruction of value types is easy to solve. The object being physically allocated on the stack of the current thread, it is automatically destroyed when the stack is emptied. The real problem exists with reference objects. To simplify the developer's task, the .NET platform provides a *garbage collector* (or commonly referred to as the *GC*) which takes care of automatically reclaiming the memory of allocated objects. This is what is called the *managed heap*.

When there are not more references to an object, it becomes inaccessible to the program since it does not need it anymore. The GC marks accessible objects. When an object is not marked this means it is not accessible by the program and must be destroyed. One problem is the fact that the GC deallocates an object only when it decides to (in general when the program needs memory). The developer has limited control over the GC. Compared to other languages such as C++, this introduces a level of uncertainty during the execution of programs. A consequence is that developers often feel frustrated by not having total control. However, with this system, recurring problems such as memory leaks are much less frequent.

Memory leaks still exist despite the presence of the GC. In fact, a developer can introduce memory leak which keeps references towards objects which are not needed anymore (for example, a collection which is never emptied). However, the most common cause of memory leaks within .NET applications is the non deletion of unmanaged resources.

Bear in mind that the GC is a software layer in the CLR where a single instance exists for each process.

Problems encountered by the garbage collecting algorithms

Our goal here is not to do an expose on all the existing garbage collection algorithms but to make you aware of the problems that the GC designers had to face. This way, you can better understand why the algorithms described in the following sections have been chosen by *Microsoft* for their garbage collection mechanism.

You may think that the task is a simple matter or releasing an object when it is not referenced anymore. But this simplistic method can easily be made to fail. For example, imagine two objects A and B; A maintains a reference to B and B maintains a reference to A; these two objects have no other references than their mutual references. Clearly, the program does not need these objects and the GC must destroy them even though there is still a reference on each of them. The GC must use reference trees and cyclic reference detection algorithms to resolve these problems

There is another problem other than the destruction of the objects. One thing that the GC must avoid is *heap fragmentation*. After a while, when multiple allocations and deallocations occur of variously sized areas of memory, the heap can become fragmented. The memory is littered with memory spaces which are unused by the program and can lead to a wasting of valuable memory. It is the GC's task to defragment the heap to limit the consequences of memory fragmentation. There are multiple algorithms which can be used to accomplish this.

Finally, common sense and empiric approach teaches us the following rule: the older an object is the longer its lifespan, the newer an object is the shorter its lifespan. If this rule is taken into account by an algorithm where we always try to deallocate newer objects than older objects, then the garbage collection mechanism using this algorithm will benefit from a performance improvement.

The algorithm used by the .NET GC takes in consideration these problems and this temporal rule in its implementation.

The .NET GC

A *collection* of objects by the GC is automatically initiated by the CLR when memory is starting to run low. The exact algorithm used to make this decision is not documented by *Microsoft*.

The term *generation* defines the time span between two collections. Each object then belongs to the generation in which it was created. In addition, the following equation is always valid:

```
{Number of generations in process} = 1 + {number of collections in the process}.
```

Generations are numbered and the number is increased at each collection until it reaches the maximum number of generations (equal to two in the current implementation of the CLR). By convention, generation 0 is always the youngest which means that every object allocated on the heap will be part of generation 0.

A consequence of this system is that objects from the same generation occupy a contiguous area in memory as shown in Figure 4-4:

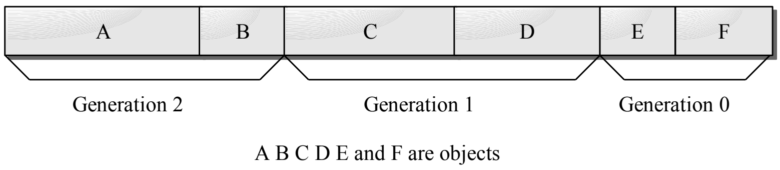

Figure 4-4: Generations

Step 1: finding the root objects

The roots of the reference tree towards the *active objects* (i.e. the objects which must not be deallocated) are objects referenced by static fields, objects referenced in the process stack and objects for which the physical address (or an offset from this address) is stored in a register of the processor.

Step 2: building the tree of active objects

The GC builds the tree based on its roots by adding the objects referenced by the ones already in the tree and by repeating this process until no more references can be added. Each object referenced by the tree is marked as being active. When the algorithm encounters an object already marked as active, it is not taken into consideration again to avoid cyclic references in the tree. The GC uses the type metadata to find references contained in the object based on its class definition.

In Figure 4-5, we see that the objects `A` and `B` are the roots of the tree and that both `A` and `B` reference `C`. `B` also references `E` and `C` references `F`. The object `D` is then not marked as active.

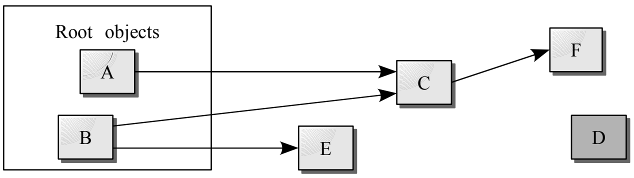

Figure 4-5: The tree of referenced objects

Step 3: deallocating inactive objects

The GC linearly traverses the heap and deallocates objects which aren't marked as being active. Certain objects need that their `Finalize()` method be called in order to be physically destroyed. This method is referred to as the *finalizer*, and is defined by the `Object` class. This method must be called right before the object is destroyed if its class overrides this method. The invocation of the finalizers is taken in charge by a dedicated thread so not to overload the thread which takes care of object collection. A consequence is that the memory location of objects with finalizers will survive through a collection.

The heap is not necessarily completely traversed, either because the GC collected enough memory or because we need to do a *partial collection*. The tree traversal algorithm is significantly impacted by partial collections because it is possible that old objects (in generation 2) references newer objects (in generation 0 or 1). Although the frequency of the triggering of collections is dependant on your application, you can base yourself on the following order of magnitude: a partial collection of generation 0 every second, a partial collection of generation 1 for each 10 partial collections of generation zero and one complete collection for each 10 partial collections of generation 1.

Figure 4-6 shows that the object `D` is not marked as active and will consequently be destroyed.

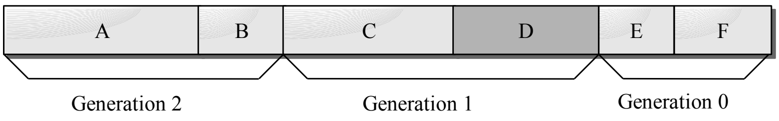

Figure 4-6: Deallocating inactive objects

Step 4: heap defragmentation

The GC defragments the heap, meaning that active objects are moved towards the bottom of heap to fill the memory voids from the deallocated objects in the previous step. The address of the top of the heap is recalculated and each generation is incremented. Older objects being towards the bottom of the heap and newer objects being at the top. In addition, active objects created at the same time are also physically close in memory. Only the new objects are often examined, meaning that it will almost always be the same memory pages which will be probed. This is done to maximize performance.

In Figure 4-7 (relating to Figure 4-6) the GC has incremented the generation number. We assume that the class for object `D` had no finalizer. This means that the memory for object `D` has been deallocated and the heap has been defragmented (`E` and `F` have been moved to fill the void left by `D`). Note that `A` and `B` did not see their generation number increase since they were already in generation 2.

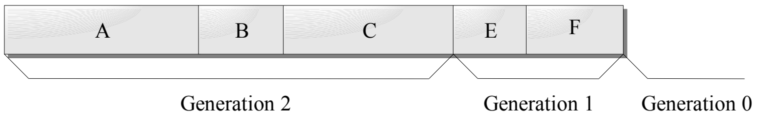

Figure 4-7: Heap defragmentation

Certain objects cannot be moved (are also said to be pinned) which means they cannot be physically moved by the GC. An object is considered to be pinned when it is referenced by unprotected or unmanaged code. For more details on this subject see page 420.

Step 5: recomputing physical addresses used by managed references

The memory addresses of certain active objects had to be modified by the previous step. The GC must then traverse the tree of active objects and update the references to the objects with their new physical memory address.

Good practices

The following good practices come from conclusions which can be drawn from the algorithm that we just described:

- Free your objects as soon as possible to avoid the promotion into a higher generation;
- Identify which objects may have a long lifespan and analyze the reasons for their lifespan so that you can attempt to reduce it. For this, we suggest that you use the *CLR Profiler* tool supplied by *Microsoft* or use the profiler supplied with *Visual Studio Team System*;
- When ever possible, avoid to reference a short lived object from a long lived object;
- Avoid implementing finalizers in your classes to that objects do not survive the collection process;
- Assign your references to `null` as soon as possible, especially before making a call to a long method.

The special heap for big objects

All objects with a size under a certain threshold are treated in the managed heap as described in the previous sections. Although this threshold is not documented by *Microsoft*, we assume an order of size in the range of 20 to 100 KB. Objects whose size is greater than this threshold are stored in a special heap for performance reasons. In fact, objects in this stack are not physically moved by the GC. A page of memory in *Windows* has a size of 4 or 8KB depending on the processor. Each of the objects in the special stack is store on an integer number of pages even if the size of the object is not an exact multiple of the page size. This leads to some minor memory waste but generally does not affect the performance. This difference in how big objects are stored is implemented transparently to the developer.

Garbage collection in a multithreaded environment

The execution of the collection process by the GC can happen with one of the application threads if it is triggered manually or using a CLR thread when it decides that collection must occur. Before starting to collect, the CLR must suspend the execution of the other application threads to avoid modifying their stacks while the collection occurs. For this, several techniques exist. We can mention the insertion of a *safe point* by the JIT compiler which allows application threads to verify if a collection is pending. Note that generation 0 of the heap is generally separated into portions (called *arenas*), one per thread, to avoid synchronization problems tied to concurrent accesses to the heap. Let us remind you that the finalizers are executed by a CLR thread dedicated to this task.

Weak references

The underlying problem

During the execution of an application, every object at a specific instant, is active, which means that the application has a reference to it or the object is inactive. When an object passes from active to inactive, which happens when the application releases the last reference to the object, there is no more possibility for the object to be accessed.

In fact, there is a third intermediate stage between active and inactive. When the object is in this stage, the application can access it but the GC can also deallocate it. There is an obvious contradiction in this stage since the object can be accessed meaning it is referenced and if referenced, it cannot be destroyed. To understand this contradiction, we have to introduce the concept of a *weak reference*. When an object is referenced by a weak reference, it is both accessible by the application and deallocatable by the GC.

Why and when should you use weak references?

A developer can use a weak reference on an object if it satisfies all of the following conditions:

- The objects might be used later but we are unsure. If we are certain to reuse it later, you should be using a strong reference.
- The object can be reconstructed if needed (from a database for example). If we potentially need this object later but cannot reconstruct it, we cannot allow the GC to destroy it.
- The object is relatively large in memory (several KB). If the object is lightweight, we can keep it in memory. However, when the two previous conditions apply to a large number of lightweight objects, it is a good idea to use weak references to each of these objects.

All of this is theoretical. In practice, we say that we are using an object cache. In fact, these conditions are all fulfilled by objects contained in a cache (we talk of cache from a conceptual point of view and not of a particular implementation).

The use of caches can be considered as a natural and automatic compromise in the use of memory, processor power and network bandwidth. When a cache is too large, a portion of its cached objects are destroyed. However, under the assumption that we will need to access one of these objects later, we will need to use processing power and bandwidth to reconstruct the object as needed

To summarize, if you need to implement a cache, we recommend the use of weak references.

How to use weak references?

The use of weak references is facilitated through the use of the `System.WeakReference` class. An example is worth more than a long discussion; take a look at the following C# code:

Example 4-14

```
class Program {
   public static void Main() {
      // ‘obj’ is a strong reference on the object
      // created by this line.
      object obj = new object();

      // ‘wobj’ is a weak reference on our object.
      System.WeakReference wobj = new System.WeakReference(obj);

      obj = null; // Discard the strong reference ‘obj’.
      // ...
      // Here, our object might potentially be deallocated by the GC.
      // ...
      // Build a strong reference from the weak reference.
      obj = wobj.Target;
      if (obj == null) {
         // If the thread pass here, it means that the object
         // has been deallocated by the GC!
      }
      else {
         // If the thread pass here, it means that the object
```

Example 4-14

```
         // hasn’t been deallocated by the GC. We can thus use it.
      }
   }
}
```

The `WeakReference` class presents the method `bool IsAlive()` which returns `true` if the object part of the weak reference has been destroyed. It is also recommended to set all strong references to `null` as soon as a weak reference has been created to ensure that the strong reference is destroyed.

Short weak references and long weak references

There are two constructors for the `WeakReference` class:

```
WeakReference(object target);
WeakReference(object target, bool trackResurrection);
```

In the first constructor the `trackResurrection` parameter is implicitly set to `false`. If this parameter is set to `true`, the application can still access the object between the moment when the `Finalize()` method has been called and the moment where the memory of the object is really modified (in general by the copy of another object into this memory location during defragmentation of the heap). In this case, we say that it is a *long weak reference*. If the parameter is set to `false` the application cannot access the object as soon as the `Finalize()` method has been called. In this case we say that it is a *short weak reference*.

Although the potential gain from using long weak references, it is better to avoid using them as they are difficult to maintain. In fact, an implementation of the `Finalize()` method which does not account for long weak references could lead to the resurrection of an object in an invalid state.

Using the System.GC class to influence the GC behaviour

We can use the static methods of the `System.GC` class to modify or analyze the behavior of the GC. The idea is to improve performance of your applications. However, *Microsoft* has invested a lot of resources into optimizing the .NET GC. We recommend using the services of the `System.GC` class only when you are certain that a performance gain can be achieved by your modifications. Here are the properties and static methods of this class:

- `static int MaxGeneration{ get; }`

  This property returns the maximum number of generations in the managed heap. By default, in the current .NET implementation, this property returns 2 and is guaranteed constant during the lifetime of the application.

- `static void WaitForPendingFinalizers()`

  This method suspends the current thread until all pending finalizers have been executed.

- `static void Collect()`
  `static void Collect( int generation )`

  This method initiates the collection process by the GC. You have the possibility of triggering a partial collection since the GC will take care of objects with a generation number between `generation` and 0. The `generation` parameter can not exceed `MaxGeneration`. When a call is made to the overloaded version without a parameter, the garage collector will collect all generations.

This method is generally invoked with a bad intent since developers hope it will solve memory problems due to bad application design (too many allocated objects, too many references between objects, memory leaks…).

It is however interesting to trigger the collection process just before the call to a critical functions that might be bothered with running low on memory or a performance drop due to an unexpected collection. In this case, it is recommended to call the methods in this order to ensure that our process will have the maximum amount of memory available:

```
// Trigger a first collection.
GC.Collect()
// Wait for all pending finalisers.
GC.WaitForPendingFinalizers()
// Trigger a second collection to get back the memory
// used by discarded objects.
GC.Collect()
```

- `static int CollectionCount( int generation )`

  Returns the number of collections done for the specified generation. This method can be used to detect if a collection has occurred during a certain time interval.

- `static int GetGeneration( object obj )`
  `static int GetGeneration( WeakReference wo )`

  Returns the generation number of an object references by either a strong reference or a weak reference.

- `static void AddMemoryPressure( long pressure )`
  `static void RemoveMemoryPressure ( long pressure )`

  The problem behind the reason for this method is that that fact that the GC does not account for unmanaged memory in its algorithms. Imagine that you have 32 instances of the `Bitmap` class which each use 32 bytes and that each of them maintain a reference to a bitmap of 6MB. Without the use of this method, the GC behaves as if only 32x32 bytes were allocated and thus might not decide that a collection is necessary. A good practice is to use theses methods during the constructors and finalizers of classes who maintain large regions of unmanaged memory. Let us mention that the class `HandleCollector` discussed at page 229 offers similar services.

- `static long GetTotalMemory( bool forceFullCollection )`

  Returns an estimate of the current managed heap size in bytes. You can refine the estimate by putting the `forceFullCollection` parameter to `true`. In this case, the method becomes blocking if a collection is currently executing. The method returns when the collection is complete or before if the wait exceeds a certain amount of time. The value returned by this function will be more exact when the GC completes its task.

- `static void KeepAlive( object obj )`

  Guarantees that `obj` will not be destroyed by the GC during the execution of the method calling `KeepAlive()`. `KeepAlive()` must be called at the end of the body of the method. You may think that this method is useless since it requires a strong reference to the object which guarantees its existence. In fact, the JIT compiler optimizes the native code by placing a local reference variable to null after its last use in a method. The `KeepAlive()` simply disables this optimization.

- `static void SuppressFinalize(object obj)`

  The `Finalize()` will not be called by the GC for the object passed as a parameter to this method. Remember however that the call to the `Finalize()` method is guaranteed by the GC for every object which has finalizer support, before the end of the process.

  The `Finalize()` method should logically contain code to deallocate resources held by the object. However, we have no control over the moment of the call to `Finalize()`. Consequently, we often create a specialized method dedicated to the deallocation of resources that we call when we wish. In general, we use the `IDisposable.Dispose()` method for this exact purpose. It is within this method that you should call `SuppressFinalize()` since after its invocation, there will be no use to call `Finalize()`. More details on this subject can be found at page 350.

- `static void ReRegisterForFinalize(object obj)`

  The `Finalize()` method of the object passed in parameter will be called by the GC. In general, we use this method in two cases.

  1st case: When the `SuppressFinalize()` has already been called and that we change our mind (this should to be avoided as it indicates poor design).

  2nd case: If we call `ReRegisterForFinalize()` in the code of the `Finalize()` method, the object will survive the collection process. This can be used to analyze the behavior of the GC. However, if we keep calling `Finalize()` the program will never terminate and you must allow for a condition in which `ReRegisterForFinalize()` will not be called so that the program may terminate. In addition, if the code of such a `Finalize()` method references other objects, you must be careful as they may be destroyed by the GC without noticing it. In fact, this 'indestructible' object is not considered as active so its references to other objects are not necessarily accounted for during the construction of the reference tree.

Facilities to make your code more reliable

Asynchronous exceptions and managed code reliability

We hope that the content of the current chapter has convinced you that the execution of managed code is a major advancement. It is now time to look at the dark side of managed environment. In such an environment, expensive processes can be triggered implicitly by the CLR at almost any time during execution. This fact implies that it is impossible to predict when a lack of resources will occur. Here are a few classic examples of such expensive processes triggered by the CLR (and this list is far from being complete):

- Loading of an assembly.
- Execution of a class constructor.
- Object collection by the GC.
- JIT compilation of a class or method.
- Traversal of the CAS stack.
- Implicit boxing.
- Creation of static fields for a class whose assembly is shared by all AppDomains of a process.

A lack of resources generally translates itself by the CLR raising an exception of type `OutOfMemoryException`, `StackOverflowException` or `ThreadAbortException` on the thread executing the request. Here, we talk about *asynchronous exceptions.* This notion opposes itself to the notion of *application exception*. When an application exception is raised, it is the current code's responsibility to trap the exception and process it. For example, when you attempt to access a file, you have to be ready to catch an exception of type `FileNotFoundException.` However, when an asynchronous exception is raised by the CLR, the code currently executing cannot be held responsible. Consequently, it is not recommended to be paranoid and litter your code with `try/catch/finally` blocks to attempt to catch the side effects of asynchronous exceptions. The entity responsible for dealing with such exceptions is the runtime host which we have already covered in this chapter.

In the case of a console or windowed application, the raise of an asynchronous event is a rare event which generally comes from an algorithm problem (memory leak, abusive recursive calls).

Consequently, the runtime host for this application will terminate the process entirely when such an exception is not caught by the application.

The behavior is similar in the case of ASP.NET applications. In fact we notice that the various abnormal behavior detection algorithms generally detect and recycle the process often before asynchronous exceptions occur. These mechanisms are discussed at page 739.

Until the integration of the CLR in *SQL Server 2005*, asynchronous exceptions were not too problematic. The *2005* version of *SQL Server* must have a reliability of 99.999%. To be efficient, the process for *SQL Server 2005* must load a maximum amount of data in memory and limit loads of memory pages from the hard disk. The process will often flirt with the 2 or 3GB memory limit for a process. Finally, the *time out* mechanisms on the execution of request are based on the raise of the `ThreadAbortException` exception. In conclusion, when dealing with this type of server which pushes the system to its limit, asynchronous exceptions become a common thing and must not cause the process to crash.

Faced with such constraints, the CLR designers had to come up with new techniques. They constitute the subject of the current section. **Keep in mind that these techniques must be used with great wisdom, only when you develop a wide scale server that requires its own runtime host and that is likely to face asynchronous exceptions.**

Constrained Execution Regions (CER)

To avoid a whole process going down from an asynchronous exception, we have to be able to unload a specific AppDomain when such an exception happens. Here, we talk about the *application domain recycling*. The main difficulty with such recycling is to achieve it properly without leaking memory or without corrupting the general state of the process. We then need a mechanism which allows us to protect ourselves from asynchronous exceptions. Without such a mechanism is it almost impossible to guarantee that unmanaged resources held during such an exception will be deallocated properly.

The .NET 2 framework allows you to tell the CLR which portions of code where the raise of an asynchronous exception would be catastrophic. If an asynchronous exception must happen, the idea is to force the CLR to raise the exception either before or after the execution of this portion of code but not during. We call these portions of code *Constrained Execution Region*s or *CER*).

To avoid the raise of an asynchronous exception during a CER, the CLR must make a certain number of preparations before it starts the execution of this portion of code. The idea is to attempt to trigger a lack of resources if it has to happen before the execution of the CER. Typically, the CLR compiles into native code all methods which are susceptible to be executed. To know these methods, it traverses statically the graph of calls which have the CER as a root. Moreover, the CLR knows how to withhold an exception of type `ThreadAbortException` which might occur during the execution of the CER until the end of its execution.

Developers must be cautious not to allocate memory during a CER. This constraint is particularly strong as many implicit conditions can trigger the allocation of memory. Some notable examples are the use of boxing instructions, access to a multi-dimensional array or the manipulation of synchronization objects.

How to define your own CERs?

A CER is defined by a call to the static method `PrepareConstrainedRegion()` of the `System.Runtime.CompilerServices.RuntimeHelpers` class right before the declaration of a `try`/`catch`/`finally` block. All the code which can be reached by the `catch` and `finally` blocks represents a CER.

The static method `ProbeForSufficientStack()` of this class is eventually called during the call to `PrepareConstrainedRegion()`. This depends on the fact that the CER implementation of your runtime host must manage cases where the execution may exceed the maximum size of the stack for the current thread (*stack overflow*). On a x86 processor, this method attempts to reserve 48KB.

Even though the reserved memory, there is still the possibility that a *stack overflow* may occur. Also, you can follow your call to `PrepareConstrainedRegion()` by a call to `ExecuteCodeWithGuaranteedCleanup()` to indicate a method containing cleanup code to invoke if case of an overflow. Such a method is marked with the `PrePrepareMethodAttribute` attribute to specify to `ngen.exe` its special use.

The `RuntimeHelpers` class offers methods which allow developers to assist the CLR in preparing to execute a CER. You can, for example, call them in your class constructors.

- `static void PrepareMethod(RuntimeMethodHandle m)`
  The CLR traverses the graph of calls made in a CER to compile them in native code. Such a traversal cannot detect which version of a virtual method is called. Also, it is the developer's responsibility to force the compilation of the version to be called within the CER.
- `static void PrepareDelegate(Delegate d)`
  Delegates that are to be called during the execution of a CER must be prepared ahead of time by using this method.
- `static void RunClassConstructor(type t)`
  You can force the execution of a class constructor with this method. Naturally, the execution will only take place if it has not already been invoked previously.

Memory Gates

In the same spirit as the `ProbeForSufficientStack()` static method, you may need to use the class `System.Runtime.MemoryFailPoint` which is used as follows:

```
// We are about to complete an operation which required
// at most 15MB of memory.
using( new MemoryFailPoint(15) ) {
    // Complete the operation ...
}
```

Contrarily to the `ProbeForSufficientStack()` method, the quantity of memory indicated is not reserved. The constructor to this class only evaluates during its execution if such a memory request can be satisfied by the operating system without raising an `OutOfMemoryException` exception. Note that between the moment where this request is done and the execution of your code, conditions may have evolved and for example another thread may have requested a big amount of memory. Considering this weakness, this technique is generally considered to be efficient.

If it happens that the amount of memory requested is unavailable, the constructor of the `MemoryFailPoint` class will raise an exception of type `System.InsufficientMemoryException`. For this reason, we often refer to *memory gates* to designate this feature. Understand that the key distinction here is that an `InsufficientMemoryException` indicates that we're out of memory, but no corruption has occurred, so therefore this exception is safe to catch and your app can safely continue.

Reliability contracts

The .NET 2 framework exposes the `System.Runtime.ConstrainedExecution.ReliabilityContractAttribute` attribute which only applies to methods. This attribute allows you to document the level of maximal severity we can expect if an asynchronous exception happens during the execution of a marked method. These severity levels are defined by the values in the `System.Runtime.ConstrainedExecution.Consistency` enumeration:

Consistency	Description
`MayCorruptProcess`	The marked method may corrupt the state of the process and thus cause a crash.
`MayCorruptAppDomain`	The marked method may corrupt the state of an AppDomain causing it to be unloaded.
`MayCorruptInstance`	The marked method may corrupt the instance which it belongs to and may cause its destruction.
`WillNotCorruptState`	The marked method may not corrupt any state.

A second value of type `System.Runtime.ConstrainedExecution.Consistency` applies to each `ReliabilityContractAttribute` attribute. This value allows you to document if the method can fault any guarantees of an eventual CER which might execute it. Of course, if the method can corrupt the state of the process or AppDomain, it can fault these guarantees and you must then use the `Cer.None` value.

Reliability contracts constitute a means by which you can document your code. Know that they are also exploited by the CLR when it traverses the static call graph to prepare the execution of a CER. If a method without a sufficient reliability contact is encountered the tree traversal will be stopped (since we know the code is unreliable) however, the execution of the CER will still happen. This potentially dangerous behavior has been decided by *Microsoft* engineers since too few framework methods are currently annotated with a reliability contract. Note that only the following three reliability contracts are considered sufficient for CER execution:

```
[ReliabilityContract(Consistency.MayCorruptInstance, Cer.MayFail)]
[ReliabilityContract(Consistency.WillNotCorruptState, Cer.MayFail)]
[ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
```

Critical finalizers

The CLR considers that the class finalizer code of a class deriving from `System.Runtime.ConstrainedExecution.CriticalFinalizerObject` is a CER. Here, we are referring to a *critical finalizer*. In addition of being a CER, critical finalizers will happen together during the execution of collection, following the execution of normal finalizers. This guarantees that the most critical resources for which manage objects depend are freed last.

The mechanism of critical finalizers is exploited by the framework classes responsible for the life-cycle of a win32 handle (`System.Runtime.InteropServices.CriticalHandle` and `System.Runtime.InteropServices.SafeHandle` described at page 229).

Other classes of the framework which derive from the `CriticalFinalizerObject` class are `System.Security.SecureString` (see page 188) and `System.Threading.ReaderWriterLock` (see page 130).

Critical regions

A second reliability mechanism is in place in addition to CERs. The idea is to supply information to the CLR to it knows when a resource shared amongst multiple threads is updated. A portion of code is responsible for the update and is called a *Critical Region* or *CR*. To define the beginning and end of a critical region, you simply need to call the `BeginCriticalRegion()` and `EndCriticalRegion()` which belong to the `Thread` class.

If an asynchronous exception occurs during a critical region, the state of the shared resource may be potentially corrupted. The current thread will be destroyed but this isn't sufficient to ensure that the application can continue its execution. In fact, other threads may access the corrupted data and yield unpredictable execution. The only possible solution is to unload the whole current AppDomain and this is effectively what will happen. The behavior of propagating a thread local problem to the AppDomain is called *escalation policy*. Another inherent effect of the notion of critical region is that it will force the CLR to unload the current AppDomain if a memory allocation request fails.

If you make use of the locking classes of the *framework* (the `Monitor` class and the `lock`, keywords of class `ReaderWriterLock` etc) to synchronize access to your shared resources, you do not need to explicitly define critical regions within your code as these classes make use of the `BeginCriticalRegion()` and `EndCriticalRegion()` methods. **Consequently, critical regions are mostly to be used when you develop a synchronization mechanism which isn't that common.**

CLI and CLS

Under these two acronyms are hidden the magic which allows .NET to support multiple languages. The *CLI* (*Common Language Infrastructure*) is a specification which describes the constraints followed by the CLR and assemblies. A software layer which supports the constraints of the CLI is capable of executing .NET applications. This specification was produced by the ECMA and can be found online at: `http://www.ecma-international.org/publications/standards/ECMA-335.HTM`

Constraints which must be satisfied by .NET languages

So that assemblies compiled from a language be managed by the CLR (or by a software layer supporting the CLI) and use all the classes and tools of the .NET framework, the language and its compiler must respect a set of constraints called the *CLS* (*Common Language Specification*). Amongst these constraints we can include the support for CTS types but this is not the only constraint. Constraints imposed to the languages and compilers by the CLS are numerous. Here are a few of the most common ones:

- The language must provide a syntax allowing the resolution of conflicts when a class implements two interfaces for which a method conflict exists. There is a method definition conflict when two methods, one in each interface, with the same name and signature. The CLS impose that the class must implement two distinct methods.
- Only a few primitive types are compatible with the CLS. For example, the `ushort` type present in C# is not compatible with the CLS.
- Parameter types to public methods must be CLS compliant. The notion of CLS compliant is presented a few lines below.
- An object launched in an exception must be an instance of the `System.Exception`, class or must be derived from it.

A complete list of all constraints can be found on **MSDN** in the article named '**Common Language Specification**'.

The compatibility of a language to the CLS does not need to be complete and we can observe two levels of compatibility:

- A language is said to be a *CLS consumer* if it can instantiate and use public members of classes contained within CLS compatible assemblies.
- A language is said to be a *CLS extender* if it can produce classes deriving from public classes contained within CLS compatible assemblies. This compatibility also implies consumer compatibility.

The C#, VB.NET and C++/CLI support both of theses levels of compatibility.

CLI and CLS from developer's perspective

An assembly is said to be CLS compliant if the following elements are compatible with the CLS:

- The definition of public types;
- The definition of public and protected members within public types;
- The parameters of public and protected methods.

This also means that the code of private classes and members do not need to be CLS compliant.

Developers have a vested interest into developing libraries which are CLS compliant since it makes it much easier to reuse them in the future. Fortunately, developers do not need to be a specialist about the CLS constraints to verify if their classes are CLS compatible. Using the `System.CLSCompliantAttribute` attribute, you can have the compiler verify if the elements of your applications (assemblies, classes, methods…) are compatible with the CLS. For example:

Example 4-15

```
using System;
[assembly : CLSCompliantAttribute(true)]
namespace CLSCompliantTest {
   public class Program {
      public static void Fct(ushort i ) { ushort j = i; }
      static void Main(){}
   }
}
```

The compiler will generate a warning as the type `ushort`, which is not CLS compatible, is used for the parameter of a public method within a public class. However, the use of the `ushort` type inside of the body of the `Fct()` method will not cause a warning.

Using the `CLSCompliantAttribute`, attribute you can indicate to the compiler not to test for CLS compliance for the `Fct()` method while testing the CLS compliance of the rest of your application as follows:

Example 4-16

```
using System;
[assembly : CLSCompliantAttribute(true)]
namespace CLSCompliantTest {
   public class Program {
      [CLSCompliantAttribute(false)]
      public static void Fct(ushort i ) { ushort j = i; }
      static void Main(){}
   }
}
```

Understand that the `Fct()` may not be called from code which does not know the `ushort` type.