This page is specific to

Microsoft Visual Studio 2010/.NET Framework 4

.NET Framework 4

**Common Language Runtime (CLR)**

The .NET Framework provides a run-time environment called the common language runtime, which runs the code and provides services that make the development process easier.

Compilers and tools expose the common language runtime's functionality and enable you to write code that benefits from this managed execution environment. Code that you develop with a language compiler that targets the runtime is called managed code; it benefits from features such as cross-language integration, cross-language exception handling, enhanced security, versioning and deployment support, a simplified model for component interaction, and debugging and profiling services.

To enable the runtime to provide services to managed code, language compilers must emit metadata that describes the types, members, and references in your code. Metadata is stored with the code; every loadable common language runtime portable executable (PE) file contains metadata. The runtime uses metadata to locate and load classes, lay out instances in memory, resolve method invocations, generate native code, enforce security, and set run-time context boundaries.

The runtime automatically handles object layout and manages references to objects, releasing them when they are no longer being used. Objects whose lifetimes are managed in this way are called managed data. Garbage collection eliminates memory leaks as well as some other common programming errors. If your code is managed, you can use managed data, unmanaged data, or both managed and unmanaged data in your .NET Framework application. Because language compilers supply their own types, such as primitive types, you might not always know (or need to know) whether your data is being managed.

The common language runtime makes it easy to design components and applications whose objects interact across languages. Objects written in different languages can communicate with each other, and their behaviors can be tightly integrated. For example, you can define a class and then use a different language to derive a class from your original class or call a method on the original class. You can also pass an instance of a class to a method of a class written in a different language. This cross-language integration is possible because language compilers and tools that target the runtime use a common type system defined by the runtime, and they follow the runtime's rules for defining new types, as well as for creating, using, persisting, and binding to types.

As part of their metadata, all managed components carry information about the components and resources they were built against. The runtime uses this information to ensure that your component or application has the specified versions of everything it needs, which makes your code less likely to break because of some unmet dependency. Registration information and state data are no longer stored in the registry where they can be difficult to establish and maintain. Instead, information about the types you define (and their dependencies) is stored with the code as metadata, making the tasks of component replication and removal much less complicated.

Language compilers and tools expose the runtime's functionality in ways that are intended to be useful and intuitive to developers. This means that some features of the runtime might be more noticeable in one environment than in another. How you experience the runtime depends on which language compilers or tools you use. For example, if you are a Visual Basic developer, you might notice that with the common language runtime, the Visual Basic language has more object-oriented features than before. The runtime provides the following benefits:

* Performance improvements.
* The ability to easily use components developed in other languages.
* Extensible types provided by a class library.
* Language features such as inheritance, interfaces, and overloading for object-oriented programming.
* Support for explicit free threading that allows creation of multithreaded, scalable applications.
* Support for structured exception handling.
* Support for custom attributes.
* Garbage collection.
* Use of delegates instead of function pointers for increased type safety and security. For more information about delegates, see [Common Type System](http://msdn.microsoft.com/en-us/library/zcx1eb1e.aspx).

This page is specific to

Microsoft Visual Studio 2010/.NET Framework 4

.NET Framework 4

**Common Type System**

The common type system defines how types are declared, used, and managed in the common language runtime, and is also an important part of the runtime's support for cross-language integration. The common type system performs the following functions:

* Establishes a framework that helps enable cross-language integration, type safety, and high-performance code execution.
* Provides an object-oriented model that supports the complete implementation of many programming languages.
* Defines rules that languages must follow, which helps ensure that objects written in different languages can interact with each other.
* Provides a library that contains the primitive data types (such as [Boolean](http://msdn.microsoft.com/en-us/library/system.boolean.aspx), [Byte](http://msdn.microsoft.com/en-us/library/system.byte.aspx), [Char](http://msdn.microsoft.com/en-us/library/system.char.aspx), [Int32](http://msdn.microsoft.com/en-us/library/system.int32.aspx), and [UInt64](http://msdn.microsoft.com/en-us/library/system.uint64.aspx)) used in application development.

This page is specific to

Microsoft Visual Studio 2010/.NET Framework 4

.NET Framework 4

**Common Language Specification**

To fully interact with other objects regardless of the language they were implemented in, objects must expose to callers only those features that are common to all the languages they must interoperate with. For this reason, the Common Language Specification (CLS), which is a set of basic language features needed by many applications, has been defined. The CLS rules define a subset of the [Common Type System](http://msdn.microsoft.com/en-us/library/zcx1eb1e.aspx); that is, all the rules that apply to the common type system apply to the CLS, except where stricter rules are defined in the CLS. The CLS helps enhance and ensure language interoperability by defining a set of features that developers can rely on to be available in a wide variety of languages. The CLS also establishes requirements for CLS compliance; these help you determine whether your managed code conforms to the CLS and to what extent a given tool supports the development of managed code that uses CLS features.

If your component uses only CLS features in the API that it exposes to other code (including derived classes), the component is guaranteed to be accessible from any programming language that supports the CLS. Components that adhere to the CLS rules and use only the features included in the CLS are said to be CLS-compliant components.

Most of the members defined by types in the [.NET Framework Class Library](http://go.microsoft.com/fwlink/?LinkID=217856) are CLS-compliant. However, some types in the class library have one or more members that are not CLS-compliant. These members enable support for language features that are not in the CLS. The types and members that are not CLS-compliant are identified as such in the reference documentation, and in all cases a CLS-compliant alternative is available. For more information about the types in the .NET Framework class library, see the [.NET Framework Class Library](http://go.microsoft.com/fwlink/?LinkID=217856).

The CLS was designed to be large enough to include the language constructs that are commonly needed by developers, yet small enough that most languages are able to support it. In addition, any language construct that makes it impossible to rapidly verify the type safety of code was excluded from the CLS so that all CLS-compliant languages can produce verifiable code if they choose to do so.

**Common Language Infrastructure**

The Common Language Infrastructure (CLI) is an open specification developed by Microsoft and standardized by ISO and ECMA that describes the executable code and runtime environment that form the core of the Microsoft .NET Framework and the free and open source implementations Mono and Portable.NET. The specification defines an environment that allows multiple high-level languages to be used on different computer platforms without being rewritten for specific architectures.

**Overview**

Among other things, the CLI specification describes the following four aspects:

**The Common Type System (CTS)**

A set of data types and operations that are shared by all CTS-compliant programming languages.

**Metadata**

Information about program structure is language-agnostic, so that it can be referenced between languages and tools, making it easy to work with code written in a language you are not using.

**Common Language Specification (CLS)**

A set of base rules to which any language targeting the CLI should conform in order to interoperate with other CLS-compliant languages. The CLS rules define a subset of the Common Type System.

**Virtual Execution System (VES)**

The VES loads and executes CLI-compatible programs, using the metadata to combine separately generated pieces of code at runtime.

All compatible languages compile to Common Intermediate Language (CIL), which is an intermediate language that is abstracted from the platform hardware. When the code is executed, the platform-specific VES will compile the CIL to the machine language according to the specific hardware and operating system.

**Common Intermediate Language**

Common Intermediate Language (CIL, pronounced either "sil" or "kil") (formerly called Microsoft Intermediate Language or MSIL) is the lowest-level human-readable programming language defined by the Common Language Infrastructure (CLI) specification and is used by the .NET Framework and Mono. Languages which target a CLS-compatible runtime environment compile to CIL, which is assembled into an object code that has a bytecode-style format. CIL is an object-oriented assembly language, and is entirely stack-based. Its bytecode is translated into native code or executed by a virtual machine.

CIL was originally known as Microsoft Intermediate Language (MSIL) during the beta releases of the .NET languages. Due to standardization of C# and the Common Language Infrastructure, the bytecode is now officially known as CIL.

**General information**

During compilation of .NET programming languages, the [source code](http://en.wikipedia.org/wiki/Source_code) is translated into CIL code rather than platform or processor-specific [object code](http://en.wikipedia.org/wiki/Object_file). CIL is a [CPU](http://en.wikipedia.org/wiki/CPU)- and platform-independent instruction set that can be executed in any environment supporting the Common Language Infrastructure, such as the [.NET runtime](http://en.wikipedia.org/wiki/Common_Language_Runtime) on [Windows](http://en.wikipedia.org/wiki/Microsoft_Windows), or the [cross-platform](http://en.wikipedia.org/wiki/Cross-platform) [Mono](http://en.wikipedia.org/wiki/Mono_(software)) runtime. In theory, this eliminates the need to distribute different executable files for different platforms and CPU types. CIL code is verified for safety during runtime, providing better security and reliability than natively compiled executable files.

The execution process looks like this:

1. Source code is converted to Common Intermediate Language, CIL's equivalent to Assembly language for a CPU.
2. CIL is then assembled into a form of so called bytecode and a .NET assembly is created.
3. Upon execution of a .NET assembly, its code is passed through the runtime's JIT compiler to generate native code. Ahead-of-time compilation may also be used, which eliminates this step, but at the cost of executable file portability.
4. The native code is executed by the computer's processor.

.NET Development (General) Technical Articles

**Microsoft .NET Framework FAQ**

Microsoft Corporation

July 2001

<http://msdn.microsoft.com/en-us/library/ms973850.aspx#faq111700_assembly02>

## Conceptual Questions

#### What is the .NET Framework?

The Microsoft .NET Framework is a platform for building, deploying, and running Web Services and applications. It provides a highly productive, standards-based, multi-language environment for integrating existing investments with next-generation applications and services as well as the agility to solve the challenges of deployment and operation of Internet-scale applications. The .NET Framework consists of three main parts: the common language runtime, a hierarchical set of unified class libraries, and a componentized version of Active Server Pages called ASP.NET.

## Runtime Technical Questions

### Terminology

#### What is the common language runtime (CLR)?

The common language runtime is the execution engine for .NET Framework applications.

It provides a number of services, including the following:

* Code management (loading and execution)
* Application memory isolation
* Verification of type safety
* Conversion of IL to native code
* Access to metadata (enhanced type information)
* Managing memory for managed objects
* Enforcement of code access security
* Exception handling, including cross-language exceptions
* Interoperation between managed code, COM objects, and pre-existing DLLs (unmanaged code and data)
* Automation of object layout
* Support for developer services (profiling, debugging, and so on)

#### What is the common type system (CTS)?

The common type system is a rich type system, built into the common language runtime, that supports the types and operations found in most programming languages. The common type system supports the complete implementation of a wide range of programming languages.

#### What is the Common Language Specification (CLS)?

The Common Language Specification is a set of constructs and constraints that serves as a guide for library writers and compiler writers. It allows libraries to be fully usable from any language supporting the CLS, and for those languages to integrate with each other. The Common Language Specification is a subset of the common type system. The Common Language Specification is also important to application developers who are writing code that will be used by other developers. When developers design publicly accessible APIs following the rules of the CLS, those APIs are easily used from all other programming languages that target the common language runtime.

#### What is the Microsoft Intermediate Language (MSIL)?

MSIL is the CPU-independent instruction set into which .NET Framework programs are compiled. It contains instructions for loading, storing, initializing, and calling methods on objects.

Combined with metadata and the common type system, MSIL allows for true cross-language integration.

Prior to execution, MSIL is converted to machine code. It is not interpreted.

#### What is managed code and managed data?

Managed code is code that is written to target the services of the common language runtime (see What is the Common Language Runtime?). In order to target these services, the code must provide a minimum level of information (metadata) to the runtime. All C#, Visual Basic .NET, and JScript .NET code is managed by default. Visual Studio .NET C++ code is not managed by default, but the compiler can produce managed code by specifying a command-line switch (/CLR).

Closely related to managed code is managed data—data that is allocated and de-allocated by the common language runtime's garbage collector. C#, Visual Basic, and JScript .NET data is managed by default. C# data can, however, be marked as unmanaged through the use of special keywords. Visual Studio .NET C++ data is unmanaged by default (even when using the /CLR switch), but when using Managed Extensions for C++, a class can be marked as managed by using the \_\_gc keyword. As the name suggests, this means that the memory for instances of the class is managed by the garbage collector. In addition, the class becomes a full participating member of the .NET Framework community, with the benefits and restrictions that brings. An example of a benefit is proper interoperability with classes written in other languages (for example, a managed C++ class can inherit from a Visual Basic class). An example of a restriction is that a managed class can only inherit from one base class.

### Assemblies

#### What is an assembly?

An assembly is the primary building block of a .NET Framework application. It is a collection of functionality that is built, versioned, and deployed as a single implementation unit (as one or more files). All managed types and resources are marked either as accessible only within their implementation unit, or as accessible by code outside that unit.

Assemblies are self-describing by means of their manifest, which is an integral part of every assembly. The manifest:

* Establishes the assembly identity (in the form of a text name), version, culture, and digital signature (if the assembly is to be shared across applications).
* Defines what files (by name and file hash) make up the assembly implementation.
* Specifies the types and resources that make up the assembly, including which are exported from the assembly.
* Itemizes the compile-time dependencies on other assemblies.
* Specifies the set of permissions required for the assembly to run properly.

This information is used at run time to resolve references, enforce version binding policy, and validate the integrity of loaded assemblies. The runtime can determine and locate the assembly for any running object, since every type is loaded in the context of an assembly. Assemblies are also the unit at which code access security permissions are applied. The identity evidence for each assembly is considered separately when determining what permissions to grant the code it contains.

The self-describing nature of assemblies also helps makes zero-impact install and XCOPY deployment feasible.

#### What are private assemblies and shared assemblies?

A private assembly is used only by a single application, and is stored in that application's install directory (or a subdirectory therein). A shared assembly is one that can be referenced by more than one application. In order to share an assembly, the assembly must be explicitly built for this purpose by giving it a cryptographically strong name (referred to as a strong name). By contrast, a private assembly name need only be unique within the application that uses it.

By making a distinction between private and shared assemblies, we introduce the notion of sharing as an explicit decision. Simply by deploying private assemblies to an application directory, you can guarantee that that application will run only with the bits it was built and deployed with. References to private assemblies will only be resolved locally to the private application directory.

There are several reasons you may elect to build and use shared assemblies, such as the ability to express version policy. The fact that shared assemblies have a cryptographically strong name means that only the author of the assembly has the key to produce a new version of that assembly. Thus, if you make a policy statement that says you want to accept a new version of an assembly, you can have some confidence that version updates will be controlled and verified by the author. Otherwise, you don't have to accept them.

For locally installed applications, a shared assembly is typically explicitly installed into the global assembly cache (a local cache of assemblies maintained by the .NET Framework). Key to the version management features of the .NET Framework is that downloaded code does not affect the execution of locally installed applications. Downloaded code is put in a special download cache and is not globally available on the machine even if some of the downloaded components are built as shared assemblies.

The classes that ship with the .NET Framework are all built as shared assemblies.

#### If I want to build a shared assembly, does that require the overhead of signing and managing key pairs?

Building a shared assembly does involve working with cryptographic keys. Only the public key is strictly needed when the assembly is being built. Compilers targeting the .NET Framework provide command line options (or use custom attributes) for supplying the public key when building the assembly. It is common to keep a copy of a common public key in a source database and point build scripts to this key. Before the assembly is shipped, the assembly must be fully signed with the corresponding private key. This is done using an SDK tool called SN.exe (Strong Name).

Strong name signing does not involve certificates like Authenticode does. There are no third party organizations involved, no fees to pay, and no certificate chains. In addition, the overhead for verifying a strong name is much less than it is for Authenticode. However, strong names do not make any statements about trusting a particular publisher. Strong names allow you to ensure that the contents of a given assembly haven't been tampered with, and that the assembly loaded on your behalf at run time comes from the same publisher as the one you developed against. But it makes no statement about whether you can trust the identity of that publisher.

#### What is the difference between a namespace and an assembly name?

A namespace is a logical naming scheme for types in which a simple type name, such as MyType, is preceded with a dot-separated hierarchical name. Such a naming scheme is completely under the control of the developer. For example, types MyCompany.FileAccess.A and MyCompany.FileAccess.B might be logically expected to have functionality related to file access. The .NET Framework uses a hierarchical naming scheme for grouping types into logical categories of related functionality, such as the Microsoft® ASP.NET application framework, or remoting functionality. Design tools can make use of namespaces to make it easier for developers to browse and reference types in their code. The concept of a namespace is not related to that of an assembly. A single assembly may contain types whose hierarchical names have different namespace roots, and a logical namespace root may span multiple assemblies. In the .NET Framework, a namespace is a logical design-time naming convenience, whereas an assembly establishes the name scope for types at run time.

### Application Deployment and Isolation

#### What options are available to deploy my .NET applications?

The .NET Framework simplifies deployment by making zero-impact install and XCOPY deployment of applications feasible. Because all requests are resolved first to the private application directory, simply copying an application's directory files to disk is all that is needed to run the application. No registration is required.

This scenario is particularly compelling for Web applications, Web Services, and self-contained desktop applications. However, there are scenarios where XCOPY is not sufficient as a distribution mechanism. An example is when the application has little private code and relies on the availability of shared assemblies, or when the application is not locally installed (but rather downloaded on demand). For these cases, the .NET Framework provides extensive code download services and integration with the Windows Installer. The code download support provided by the .NET Framework offers several advantages over current platforms, including incremental download, code access security (no more Authenticode dialogs), and application isolation (code downloaded on behalf of one application doesn't affect other applications). The Windows Installer is another powerful deployment mechanism available to .NET applications. All of the features of Windows Installer, including publishing, advertisement, and application repair will be available to .NET applications in Windows Installer 2.0.

#### I've written an assembly that I want to use in more than one application. Where do I deploy it?

Assemblies that are to be used by multiple applications (for example, shared assemblies) are deployed to the global assembly cache. In the prerelease and Beta builds, use the /i option to the GACUtil SDK tool to install an assembly into the cache:

gacutil /i myDll.dll

Windows Installer 2.0, which ships with Windows XP and Visual Studio .NET will be able to install assemblies into the global assembly cache.

#### How can I see what assemblies are installed in the global assembly cache?

The .NET Framework ships with a Windows shell extension for viewing the assembly cache. Navigating to % windir%\assembly with the Windows Explorer activates the viewer.

#### What is an application domain?

An application domain (often AppDomain) is a virtual process that serves to isolate an application. All objects created within the same application scope (in other words, anywhere along the sequence of object activations beginning with the application entry point) are created within the same application domain. Multiple application domains can exist in a single operating system process, making them a lightweight means of application isolation.

An OS process provides isolation by having a distinct memory address space. While this is effective, it is also expensive, and does not scale to the numbers required for large web servers. The Common Language Runtime, on the other hand, enforces application isolation by managing the memory use of code running within the application domain. This ensures that it does not access memory outside the boundaries of the domain. It is important to note that only type-safe code can be managed in this way (the runtime cannot guarantee isolation when unsafe code is loaded in an application domain).

### Garbage Collection

#### What is garbage collection?

Garbage collection is a mechanism that allows the computer to detect when an object can no longer be accessed. It then automatically releases the memory used by that object (as well as calling a clean-up routine, called a "finalizer," which is written by the user). Some garbage collectors, like the one used by .NET, compact memory and therefore decrease your program's working set.

#### How does non-deterministic garbage collection affect my code?

For most programmers, having a garbage collector (and using garbage collected objects) means that you never have to worry about deallocating memory, or reference counting objects, even if you use sophisticated data structures. It does require some changes in coding style, however, if you typically deallocate system resources (file handles, locks, and so forth) in the same block of code that releases the memory for an object. With a garbage collected object you should provide a method that releases the system resources deterministically (that is, under your program control) and let the garbage collector release the memory when it compacts the working set.

#### Can I avoid using the garbage collected heap?

All languages that target the runtime allow you to allocate class objects from the garbage-collected heap. This brings benefits in terms of fast allocation, and avoids the need for programmers to work out when they should explicitly 'free' each object.

The CLR also provides what are called ValueTypes—these are like classes, except that ValueType objects are allocated on the runtime stack (rather than the heap), and therefore reclaimed automatically when your code exits the procedure in which they are defined. This is how "structs" in C# operate.

Managed Extensions to C++ lets you choose where class objects are allocated. If declared as managed Classes, with the \_\_gc keyword, then they are allocated from the garbage-collected heap. If they don't include the \_\_gc keyword, they behave like regular C++ objects, allocated from the C++ heap, and freed explicitly with the "free" method.

For additional information about Garbage Collection see:

* [Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework](http://msdn.microsoft.com/en-us/library/bb985010.aspx)
* [Garbage Collection, Part 2: Automatic Memory Management in the Microsoft .NET Framework](http://msdn.microsoft.com/library/default.asp?url=/msdnmag/issues/1200/GCI2/TOC.ASP)

[Back to Top](http://msdn.microsoft.com/en-us/library/ms973850.aspx#faq111700_concepts)

### Remoting

#### How do in-process and cross-process communication work in the Common Language Runtime?

There are two aspects to in-process communication: between contexts within a single application domain, or across application domains. Between contexts in the same application domain, proxies are used as an interception mechanism. No marshaling/serialization is involved. When crossing application domains, we do marshaling/serialization using the runtime binary protocol.

Cross-process communication uses a pluggable channel and formatter protocol, each suited to a specific purpose.

* If the developer specifies an endpoint using the tool soapsuds.exe to generate a metadata proxy, HTTP channel with SOAP formatter is the default.
* If a developer is doing explicit remoting in the managed world, it is necessary to be explicit about what channel and formatter to use. This may be expressed administratively, through configuration files, or with API calls to load specific channels. Options are:

HTTP channel w/ SOAP formatter (HTTP works well on the Internet, or anytime traffic must travel through firewalls)

TCP channel w/ binary formatter (TCP is a higher performance option for local-area networks (LANs))

When making transitions between managed and unmanaged code, the COM infrastructure (specifically, DCOM) is used for remoting. In interim releases of the CLR, this applies also to serviced components (components that use COM+ services). Upon final release, it should be possible to configure any remotable component.

Distributed garbage collection of objects is managed by a system called "leased based lifetime." Each object has a lease time, and when that time expires, the object is disconnected from the remoting infrastructure of the CLR. Objects have a default renew time-the lease is renewed when a successful call is made from the client to the object. The client can also explicitly renew the lease.

### Interoperability

#### Can I use COM objects from a .NET Framework program?

Yes. Any COM component you have deployed today can be used from managed code, and in common cases the adaptation is totally automatic.

Specifically, COM components are accessed from the .NET Framework by use of a runtime callable wrapper (RCW). This wrapper turns the COM interfaces exposed by the COM component into .NET Framework-compatible interfaces. For OLE automation interfaces, the RCW can be generated automatically from a type library. For non-OLE automation interfaces, a developer may write a custom RCW and manually map the types exposed by the COM interface to .NET Framework-compatible types.

#### Can .NET Framework components be used from a COM program?

Yes. Managed types you build today can be made accessible from COM, and in the common case the configuration is totally automatic. There are certain new features of the managed development environment that are not accessible from COM. For example, static methods and parameterized constructors cannot be used from COM. In general, it is a good idea to decide in advance who the intended user of a given type will be. If the type is to be used from COM, you may be restricted to using those features that are COM accessible.

Depending on the language used to write the managed type, it may or may not be visible by default.

Specifically, .NET Framework components are accessed from COM by using a COM callable wrapper (CCW). This is similar to an RCW (see previous question), but works in the opposite direction. Again, if the .NET Framework development tools cannot automatically generate the wrapper, or if the automatic behavior is not what you want, a custom CCW can be developed.

#### Can I use the Win32 API from a .NET Framework program?

Yes. Using platform invoke, .NET Framework programs can access native code libraries by means of static DLL entry points.

Here is an example of C# calling the Win32 **MessageBox** function:

using System;

using System.Runtime.InteropServices;

class MainApp

{

[DllImport("user32.dll", EntryPoint="MessageBox")]

public static extern int MessageBox(int hWnd, String strMessage, String strCaption, uint uiType);

public static void Main()

{

MessageBox( 0, "Hello, this is PInvoke in operation!", ".NET", 0 );

}

}

### Security

#### What do I have to do to make my code work with the security system?

Usually, not a thing—most applications will run safely and will not be exploitable by malicious attacks. By simply using the standard class libraries to access resources (like files) or perform protected operations (such as a reflection on private members of a type), security will be enforced by these libraries. The one simple thing application developers may want to do is include a permission request (a form of declarative security) to limit the permissions their code may receive (to only those it requires). This also ensures that if the code is allowed to run, it will do so with all the permissions it needs.

Only developers writing new base class libraries that expose new kinds of resources need to work directly with the security system. Instead of all code being a potential security risk, code access security constrains this to a very small bit of code that explicitly overrides the security system.

#### Why does my code get a security exception when I run it from a network shared drive?

Default security policy gives only a restricted set of permissions to code that comes from the local intranet zone. This zone is defined by the Internet Explorer security settings, and should be configured to match the local network within an enterprise. Since files named by UNC or by a mapped drive (such as with the NET USE command) are being sent over this local network, they too are in the local intranet zone.

The default is set for the worst case of an unsecured intranet. If your intranet is more secure you can modify security policy (with the .NET Framework Configuration tool or the CASPol tool) to grant more permissions to the local intranet, or to portions of it (such as specific machine share names).

#### How do I make it so that code runs when the security system is stopping it?

Security exceptions occur when code attempts to perform actions for which it has not been granted permission. Permissions are granted based on what is known about code; especially its location. For example, code run from the Internet is given fewer permissions than that run from the local machine because experience has proven that it is generally less reliable. So, to allow code to run that is failing due to security exceptions, you must increase the permissions granted to it. One simple way to do so is to move the code to a more trusted location (such as the local file system). But this won't work in all cases (web applications are a good example, and intranet applications on a corporate network are another). So, instead of changing the code's location, you can also change security policy to grant more permissions to that location. This is done using either the .NET Framework Configuration tool or the code access security policy utility (caspol.exe). If you are the code's developer or publisher, you may also digitally sign it and then modify security policy to grant more permissions to code bearing that signature. When taking any of these actions, however, remember that code is given fewer permissions because it is not from an identifiably trustworthy source—before you move code to your local machine or change security policy, you should be sure that you trust the code to not perform malicious or damaging actions.

#### How do I administer security for my machine? For an enterprise?

The .NET Framework includes the .NET Framework Configuration tool, an MMC snap-in (mscorcfg.msc), to configure several aspects of the CLR including security policy. The snap-in not only supports administering security policy on the local machine, but also creates enterprise policy deployment packages compatible with System Management Server and Group Policy. A command line utility, CASPol.exe, can also be used to script policy changes on the computer. In order to run either tool, in a command prompt, change the current directory to the installation directory of the .NET Framework (located in %windir%\Microsoft.Net\Framework\v1.0.2914.16\) and type **mscorcfg.msc** or **caspol.exe**.

#### How does evidence-based security work with Windows 2000 security?

Evidence-based security (which authorizes code) works together with Windows 2000 security (which is based on log on identity). For example, to access a file, managed code must have both the code access security file permission and must also be running under a log on identity that has NTFS file access rights. The managed libraries that are included with the .NET Framework also provide classes for role-based security. These allow the application to work with Windows log on identities and user groups.