CSHARP INTERVIEW QUESTIONS

www.enosislearning.com

**Question 1. What is the .NET Framework?**

**Answer:** The .NET is a Framework, which is a collection of classes of reusable libraries given by Microsoft to be used in other .NET applications and to develop, build and deploy many types of applications on the Windows platform including the following:

* Console Applications
* Windows Forms Applications
* Windows Presentation Foundation (WPF) Applications
* Web Applications
* Web Services
* Windows Services
* Services-oriented applications using Windows Communications Foundation (WCF)
* Workflow-enabled applications using Windows Workflow Foundation(WF)

That primarily runs on the Microsoft Windows Operating System.



**Question 2. What is CLR?**

**Answer:** The CLR stands for Common Language Runtime and it is an Execution Environment. It works as a layer between Operating Systems and the applications written in .NET languages that conforms to the Common Language Specification (CLS). The main function of Common Language Runtime (CLR) is to convert the Managed Code into native code and then execute the program. The Managed Code compiled only when it is needed, that is it converts the appropriate instructions when each function is called. The Common Language Runtime (CLR)’s just in time (JIT) compilation converts Intermediate Language (MSIL) to native code on demand at application run time.

When a .NET application is executed at that time the control will go to Operating System, then Operating System create a process to load **CLR.**
The program used by the operating system for loading CLR is called runtime host, which are different depending upon the type of application that is desktop or web based application i.e.

The runtime host for **desktop applications** is API function called **CorbinToRuntime**.

The runtime host for **web based** applications is ASP.NET worker process **(aspnet-wp.exe)**.



CLR runtime engine comes with set of services, which are classified as follows

**CLR services**

* Assembly Resolver
* Assembly Loader
* Type Checker
* COM marshalled
* Debug Manager
* Thread Support
* IL to Native compiler
* Exception Manager
* Garbage Collector

**Question 3. What is CTS?**

**Answer:** The Common Type System (CTS) standardizes the data types of all programming languages using .NET under the umbrella of .NET to a common data type for easy and smooth communication among these .NET languages.



To implement or see how CTS is converting the data type to a common data type, for example, when we declare an int type data type in C# and VB.NET, then they are converted to int32. In other words, now both will have a common data type that provides flexible communication between these two languages.

**Question 4. What is CLS?**

**Answer**: One of the important goals of .NET Framework is to support Multiple Languages. This is achieved by CLS. For multiple languages to interoperate, it is necessary that they should go on in common in certain features such as Types that are used. For example, every language has its own size and range for different data types. Thus CLS is the agreement among language designers and class library designers concerning these usage conventions.



**Question 5. What is managed code?**
**Answer:** The resource, which is within your application domain is, managed code. The resources that are within domain are faster.

The code, which is developed in .NET framework, is known as managed code. This code is directly executed by CLR with help of managed code execution. Any language that is written in .NET Framework is managed code.

Managed code uses CLR which in turn looks after your applications by managing memory, handling security, allowing cross - language debugging, and so on.



**Question 6. What is MSIL?**

**Answer:** When we compile our .NET code then it is not directly converted to native/binary code; it is first converted into intermediate code known as MSIL code which is then interpreted by the CLR. MSIL is independent of hardware and the operating system. Cross language relationships are possible since MSIL is the same for all .NET languages. MSIL is further converted into native code.



**Question 7. What is JIT?**

**Answer**: A Web Service or Web Forms file must be compiled to run within the CLR. Compilation can be implicit or explicit. Although you could explicitly call the appropriate compiler to compile your Web Service or Web Forms files, it is easier to allow the file to be complied implicitly. Implicit compilation occurs when you request the .asmx via HTTP-SOAP, HTTP-GET, or HTTP-POST. The parser (xsp.exe) determines whether a current version of the assembly resides in memory or in the disk. If it cannot use an existing version, the parser makes the appropriate call to the respective compiler (as you designated in the **Class** property of the .asmx page).

When the Web Service (or Web Forms page) is implicitly compiled, it is actually compiled twice. On the first pass, it is compiled into IL. On the second pass, the Web Service (now an assembly in IL) is compiled into machine language. This process is called Just-In-Time JIT compilation because it does not occur until the assembly is on the target machine.

**JIT Types:**


**Question 8. What is portable executable (PE)?**

**Answer:** Every .NET program first compiles with an appropriate compiler like if we write a program in C# language then it gets compiled by C# compiler (i.e. csc.exe).

In .NET framework every program executes (communicate) in an operating system by using CLR (Common Language Runtime).



Managed module is standard windows Portable Executable (PE) file which contains the following parts.

* **PE Header**

It is similar to common object file format.
* **CLR Header**

This contains CLR version required to run this managed module, location & metadata. This also contains entry point of function i.e. the address of entry point of function.
* **Metadata**

This contains table information means variable with its data types and default values, functions / methods which are declared & defined in our program.

**Question 9. What is an application domain?**

**Answer:** An Application Domain is a logical container for a set of assemblies in which an executable is hosted. As you have seen, a single process may contain multiple Application Domains, each of which is hosting a .NET executable. The first appdomain created when the CLR is initialized is called the default AppDomain and this default one is destroyed when the Windows process is terminated.

* An AppDomain can be independently secured.
* An AppDomain can be unloaded.
* Independently configured.
* No mutual intervention by multiple appdomains.
* Performance

**
How does an AppDomain get created**

The AppDomain class is used to create and terminate Application Domains, load and unload assemblies and types and enumerates assemblies and threads in a domain. The following table shows some useful methods of the AppDomain class:

|  |  |
| --- | --- |
| **Methods** | **Description** |
| CreateDomain() | It allows us to create a new Application Domain. |
| CreateInstance() | Creates an instance of type in an external assembly. |
| ExecuteAssembly() | It executes an \*.exe assembly in the Application Domain. |
| Load() | This method dynamically loads an assembly into the current app domain. |
| UnLoad() | It allows us to unload a specified AppDomain within a given process. |
| GetCurrentThread() | Returns the ID of the active thread in the current Application Domain. |

In addition, the AppDomain class also defined as a set of properties that can be useful when you wish to monitor the activity of a given Application Domain.

|  |  |
| --- | --- |
| **Properties** | **Description** |
| CurrentDomain | Gets the Application Domain for the currently executing thread. |
| FriendlyName | Gets the friendly name of the current Application Domain. |
| SetupInformation | Get the configuration details for a given Application Domain. |
| BaseDirectory | Gets the directory path that the assembly resolver uses to probe for assemblies. |

**Question 10. What is an assembly?**

**Answer:** An Assembly is a basic building block of .NET Framework applications. It is basically compiled code that can be executed by the CLR. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. An Assembly can be a DLL or exe depending upon the project that we choose.

Assemblies are basically the following two types:

1. Private Assembly
2. Shared Assembly

**Question 11. What are the contents of assembly?**

**Answer:** Assembly

* An Assembly is a basic unit of application deployment and versioning.
* An Assembly is also called the building block of a .NET application.
* An Assembly is either a .exe or .dll file.

An Assembly structure consists of the following parts:

* Assembly manifest (name, language and version).
* CIL code (logic part).
* Type information (Datatype).
* Resources.

**Question 12. What are the different types of assembly?**

**Answer**: An Assembly contains metadata and manifest information. The reason for the emergence of assembly concept was to overcome the common "**DLL Hell**" problem in COM. The assembly contains all the code, resources, metadata and even version information. Metadata contains the details of every "type" inside the assembly. In addition to metadata, assemblies also have a special file called Manifest. It contains information about the current version of the assembly, culture information, public key token if available and other related information.

There are in all 3 different types of assemblies:

1. Private Assembly
2. Shared or Strong named assembly
3. Satellite assembly

**Question 13. What is a dynamic assembly?**

**Answer:** Technically, the act of loading external assemblies on demand is known as Dynamic Loading. Using the Assembly class, we can dynamically load both private and shared assemblies from the local location to a remote location as well as, explore its properties.

To illustrate dynamic loading, we are creating a console based application that loads an external TestLib.dll assembly. During the execution, the application asks the user to specify the dynamic loading assembly name and that reference is passed to the helper method that is responsible for loading the assembly.

**Question 14. What is GAC?**

**Answer:** The GAC is a shared location of computer where we can put an assembly so that it will be accessible from many locations, I mean it is accessible from another project or application. It's always a good practice to provide a strong name to a public assembly, I mean the assembly to be registered in the GAC,  otherwise the DLL hell problem may occur.

**Problems that occurred**
I have seen DLLs added to the GAC that you can't remove - very frustrating. I have seen registered DLLs into the cache - verified everything is there ok using ILDASM only to find the DLLs are no longer in the GAC.

**Strongly naming the assembly**
When doing this make sure you get the directory slashes \\ correct within the assembly file (assembly.cs). - if not, you will get errors whilst the code is looking for the .snk file. If you get errors which leave you scratching your head - best bet is to remove the .snk file and start over.

**Project References**
Also be careful and watch where you build projects as the referenced DLLs can easily be built to the development instead of the release folder - sometimes even when you specify the release folder. This can be very, very frustrating.

**Conclusion**
My conclusion on using the GAC was only use it if you really need to as it isn't the 'end of DLL hell' as first thought. Also only use it if you are using a DLL that is shared by other projects. Don't put it in the GAC if you don't have to.

**Question 15. What is a garbage collector?**

**Answer:** The Garbage Collector (GC) is the part of the .NET Framework that allocates and releases memory for your .NET applications. The Common Language Runtime (CLR) manages allocation and deallocation of a managed object in memory. C# programmers never do this directly, there is no delete keyword in the C# language. It relies on the garbage collector.

**Example**
Assume the managed heap contains a set of objects named A, B, C, D, E, F and G. During garbage collection, these objects are examined for active roots. After the graph has been constructed, unreachable objects (that we will assume are objects C and F) are marked as garbage in reddish color in the following diagram.


**Question 16. What are generations and how are they used by the garbage collector?**

**Answer**: Basically the generation of Garbage Collection (GC) shows the life of objects, it means it defines how long an object will stay in the memory. It's categorized into the following three generations:

* Generation 0
* Generation 1
* Generation 2



**1.1 What is .NET?**

.NET is a general-purpose software development platform, similar to Java. At its core is a virtual machine that turns intermediate language (IL) into machine code. High-level language compilers for C#, VB.NET and C++ are provided to turn source code into IL. C# is a new programming language, very similar to Java. An extensive class library is included, featuring all the functionality one might expect from a contempory development platform - windows GUI development (Windows Forms), database access (ADO.NET), web development (ASP.NET), web services, XML etc.

**1.2 When was .NET announced?**

Bill Gates delivered a keynote at Forum 2000, held June 22, 2000, outlining the .NET 'vision'. The July 2000 PDC had a number of sessions on .NET technology, and delegates were given CDs containing a pre-release version of the .NET framework/SDK and Visual Studio.NET.

**1.3 What versions of .NET are there?**

The final versions of the 1.0 SDK and runtime were made publicly available around 6pm PST on 15-Jan-2002. At the same time, the final version of Visual Studio.NET was made available to MSDN subscribers.

.NET 1.1 was released in April 2003, and was mostly bug fixes for 1.0.

.NET 2.0 was released to MSDN subscribers in late October 2005, and was officially launched in early November.

**2. Terminology**

**2.1 What is the CLI? Is it the same as the CLR?**

The CLI (Common Language Infrastructure) is the definiton of the fundamentals of the .NET framework - the Common Type System (CTS), metadata, the Virtual Execution Environment (VES) and its use of intermediate language (IL), and the support of multiple programming languages via the Common Language Specification (CLS).

The CLR (Common Language Runtime) is Microsoft's primary *implementation* of the CLI.

**2.2 What is IL?**

IL = Intermediate Language. Also known as MSIL (Microsoft Intermediate Language) or CIL (Common Intermediate Language). All .NET source code (of any language) is compiled to IL during development. The IL is then converted to machine code at the point where the software is installed, or (more commonly) at run-time by a Just-In-Time (JIT) compiler.

**2.3 What is C#?**

C# is a new language designed by Microsoft to work with the .NET framework. In their "Introduction to C#" whitepaper, Microsoft describe C# as follows:

"C# is a simple, modern, object oriented, and type-safe programming language derived from C and C++. C# (pronounced “C sharp”) is firmly planted in the C and C++ family tree of languages, and will immediately be familiar to C and C++ programmers. C# aims to combine the high productivity of Visual Basic and the raw power of C++."

Substitute 'Java' for 'C#' in the quote above, and you'll see that the statement still works pretty well :-).

**2.4 What does 'managed' mean in the .NET context?**

The term 'managed' is the cause of much confusion. It is used in various places within .NET, meaning slightly different things.

Managed *code*: The .NET framework provides several core run-time services to the programs that run within it - for example exception handling and security. For these services to work, the code must provide a minimum level of information to the runtime. Such code is called managed code.

Managed *data*: This is data that is allocated and freed by the .NET runtime's garbage collector.

Managed *classes*: This is usually referred to in the context of Managed Extensions (ME) for C++. When using ME C++, a class can be marked with the \_\_gc keyword. As the name suggests, this means that the memory for instances of the class is managed by the garbage collector, but it also means more than that. The class becomes a fully paid-up member of the .NET community with the benefits and restrictions that brings. An example of a benefit is proper interop with classes written in other languages - for example, a managed C++ class can inherit from a VB class. An example of a restriction is that a managed class can only inherit from one base class.

**3. Assemblies**

**3.1 What is an assembly?**

An assembly is sometimes described as a logical .EXE or .DLL, and can be an *application* (with a main entry point) or a *library*. An assembly consists of one or more files (dlls, exes, html files etc), and represents a group of resources, type definitions, and implementations of those types. An assembly may also contain references to other assemblies. These resources, types and references are described in a block of data called a *manifest*. The manifest is part of the assembly, thus making the assembly self-describing.

An important aspect of assemblies is that they are part of the identity of a type. The identity of a type is the assembly that houses it combined with the type name. This means, for example, that if assembly A exports a type called T, and assembly B exports a type called T, the .NET runtime sees these as two completely different types. Furthermore, don't get confused between assemblies and namespaces - namespaces are merely a hierarchical way of organising type names. To the runtime, type names are type names, regardless of whether namespaces are used to organise the names. It's the assembly plus the typename (regardless of whether the type name belongs to a namespace) that uniquely indentifies a type to the runtime.

Assemblies are also important in .NET with respect to security - many of the security restrictions are enforced at the assembly boundary.

**3.2 How can I produce an assembly?**

The simplest way to produce an assembly is directly from a .NET compiler. For example, the following C# program:

public class CTest

 {

publicCTest() { System.Console.WriteLine( "Hello from CTest" ); }

 }

can be compiled into a library assembly (dll) like this:

csc /t:libraryctest.cs

You can then view the contents of the assembly by running the "IL Disassembler" tool that comes with the .NET SDK.

Alternatively you can compile your source into modules, and then combine the modules into an assembly using the assembly linker (al.exe). For the C# compiler, the /target:module switch is used to generate a module instead of an assembly.

**3.3 What is the difference between a private assembly and a shared assembly?**

The terms 'private' and 'shared' refer to how an assembly is deployed, not any intrinsic attributes of the assembly.

A private assembly is normally used by a single application, and is stored in the application's directory, or a sub-directory beneath. A shared assembly is intended to be used by multiple applications, and is normally stored in the global assembly cache (GAC), which is a central repository for assemblies. (A shared assembly can also be stored outside the GAC, in which case each application must be pointed to its location via a codebase entry in the application's configuration file.) The main advantage of deploying assemblies to the GAC is that the GAC can support multiple versions of the same assembly side-by-side.

Assemblies deployed to the GAC must be strong-named. Outside the GAC, strong-naming is optional.

**3.4 How do assemblies find each other?**

By searching directory paths. There are several factors that can affect the path (such as the AppDomain host, and application configuration files), but for weakly named assemblies the search path is normally the application's directory and its sub-directories. For strongly named assemblies, the search path is the GAC followed by the private assembly path.

 **3.5 How does assembly versioning work?**

An assembly has a version number consisting of four parts, e.g. 1.0.350.1. These are typically interpreted as Major.Minor.Build.Revision, but this is just a convention.

The CLR applies no version constraints on weakly named assemblies, so the assembly version has no real significance.

For strongly named assemblies, the version of a referenced assembly is stored in the referring assembly, and by default only this exact version will be loaded at run-time. If the exact version is not available, the referring assembly will fail to load. It is possible to override this behaviour in the config file for the referring assembly - references to a single version or a range of versions of the referenced assembly can be redirected to a specific version. For example, versions 1.0.0.0 to 2.0.0.0 can be redirected to version 3.0.125.3. However note that there is no way to specify a range of versions to be redirected *to*. Publisher policy files offer an alternative mechanism for redirecting to a different version for assemblies deployed to the GAC - a publisher policy file allows the publisher of the assembly to redirect *all* applications to a new version of an assembly in one operation, rather than having to modify all of the application configuration files.

**4. Application Domains**

**4.1 What is an application domain?**

An AppDomain can be thought of as a lightweight process. Multiple AppDomains can exist inside a Win32 process. The primary purpose of the AppDomain is to isolate applications from each other, and so it is particularly useful in hosting scenarios such as ASP.NET. An AppDomain can be destroyed by the host without affecting other AppDomains in the process.

Win32 processes provide isolation by having distinct memory address spaces. This is effective, but expensive. The .NET runtime enforces AppDomain isolation by keeping control over the use of memory - all memory in the AppDomain is managed by the .NET runtime, so the runtime can ensure that AppDomains do not access each other's memory.

One non-obvious use of AppDomains is for unloading types. Currently the only way to unload a .NET type is to destroy the AppDomain it is loaded into. This is particularly useful if you create and destroy types on-the-fly via reflection.

**4.2 How does an AppDomain get created?**

AppDomains are usually created by hosts. Examples of hosts are the Windows Shell, ASP.NET and IE. When you run a .NET application from the command-line, the host is the Shell. The Shell creates a new AppDomain for every application.

AppDomains can also be explicitly created by .NET applications. Here is a C# sample which creates an AppDomain, creates an instance of an object inside it, and then executes one of the object's methods:

using System;

usingSystem.Runtime.Remoting;

usingSystem.Reflection;

public class CAppDomainInfo : MarshalByRefObject

 {

public string GetName() { return AppDomain.CurrentDomain.FriendlyName; }

 }

public class App

 {

public static int Main()

 {

AppDomain ad = AppDomain.CreateDomain( "Andy's new domain" );

CAppDomainInfoadInfo = (CAppDomainInfo)ad.CreateInstanceAndUnwrap(

Assembly.GetCallingAssembly().GetName().Name, "CAppDomainInfo" );

Console.WriteLine( "Created AppDomain name = " + adInfo.GetName() );

return 0;

 }

 }

**5. Garbage Collection**

**5.1 What is garbage collection?**

Garbage collection is a heap-management strategy where a run-time component takes responsibility for managing the lifetime of the memory used by objects. This concept is not new to .NET - Java and many other languages/runtimes have used garbage collection for some time.

**5.2 Is it true that objects don't always get destroyed immediately when the last reference goes away?**

Yes. The garbage collector offers no guarantees about the time when an object will be destroyed and its memory reclaimed.

**5.3 Why doesn't the .NET runtime offer deterministic destruction?**

Because of the garbage collection algorithm. The .NET garbage collector works by periodically running through a list of all the objects that are currently being referenced by an application. All the objects that it doesn't find during this search are ready to be destroyed and the memory reclaimed. The implication of this algorithm is that the runtime doesn't get notified immediately when the final reference on an object goes away - it only finds out during the next 'sweep' of the heap.

Futhermore, this type of algorithm works best by performing the garbage collection sweep as rarely as possible. Normally heap exhaustion is the trigger for a collection sweep.

**5.4 Is the lack of deterministic destruction in .NET a problem?**

It's certainly an issue that affects component design. If you have objects that maintain expensive or scarce resources (e.g. database locks), you need to provide some way to tell the object to release the resource when it is done. Microsoft recommend that you provide a method called Dispose() for this purpose. However, this causes problems for distributed objects - in a distributed system who calls the Dispose() method? Some form of reference-counting or ownership-management mechanism is needed to handle distributed objects - unfortunately the runtime offers no help with this.

**5.5 Should I implement Finalize on my class? Should I implement IDisposable?**

This issue is a little more complex than it first appears. There are really two categories of class that require deterministic destruction - the first category manipulate unmanaged types directly, whereas the second category manipulate *managed* types that require deterministic destruction. An example of the first category is a class with an IntPtr member representing an OS file handle. An example of the second category is a class with a System.IO.FileStream member.

For the first category, it makes sense to implement IDisposable*and* override Finalize. This allows the object user to 'do the right thing' by calling Dispose, but also provides a fallback of freeing the unmanaged resource in the Finalizer, should the calling code fail in its duty. However this logic does not apply to the second category of class, with only managed resources. In this case implementing Finalize is pointless, as managed member objects cannot be accessed in the Finalizer. This is because there is no guarantee about the ordering of Finalizer execution. So only the Dispose method should be implemented. (If you think about it, it doesn't really make sense to call Dispose on member objects from a Finalizer anyway, as the member object's Finalizer will do the required cleanup.)

**5.6 Do I have any control over the garbage collection algorithm?**

A little. For example the System.GC class exposes a Collect method, which forces the garbage collector to collect all unreferenced objects immediately.

Also there is a **gcConcurrent** setting that can be specified via the application configuration file. This specifies whether or not the garbage collector performs some of its collection activities on a separate thread. The setting only applies on multi-processor machines, and defaults to true.

**5.7 How can I find out what the garbage collector is doing?**

Lots of interesting statistics are exported from the .NET runtime via the '.NET CLR xxx' performance counters. Use Performance Monitor to view them.

**5.8 What is the lapsed listener problem?**

The lapsed listener problem is one of the primary causes of leaks in .NET applications. It occurs when a subscriber (or 'listener') signs up for a publisher's event, but fails to unsubscribe. The failure to unsubscribe means that the publisher maintains a reference to the subscriber as long as the publisher is alive. For some publishers, this may be the duration of the application.

This situation causes two problems. The obvious problem is the leakage of the subscriber object. The other problem is the performance degredation due to the publisher sending redundant notifications to 'zombie' subscribers.

There are at least a couple of solutions to the problem. The simplest is to make sure the subscriber is unsubscribed from the publisher, typically by adding an Unsubscribe() method to the subscriber. Another solution, documented [here](http://www.windojitsu.com/blog/weakevent.html) by Shawn Van Ness, is to change the publisher to use weak references in its subscriber list.

**6. Serialization**

**6.1 What is serialization?**

Serialization is the process of converting an object into a stream of bytes. Deserialization is the opposite process, i.e. creating an object from a stream of bytes. Serialization/Deserialization is mostly used to transport objects (e.g. during remoting), or to persist objects (e.g. to a file or database).

**6.2 Does the .NET Framework have in-built support for serialization?**

There are two separate mechanisms provided by the .NET class library - XmlSerializer and SoapFormatter/BinaryFormatter. Microsoft uses XmlSerializer for Web Services, and SoapFormatter/BinaryFormatter for remoting. Both are available for use in your own code.

**6.3 I want to serialize instances of my class. Should I use XmlSerializer, SoapFormatter or BinaryFormatter?**

It depends. XmlSerializer has severe limitations such as the requirement that the target class has a parameterless constructor, and only public read/write properties and fields can be serialized. However, on the plus side, XmlSerializer has good support for customising the XML document that is produced or consumed. XmlSerializer's features mean that it is most suitable for cross-platform work, or for constructing objects from existing XML documents.

SoapFormatter and BinaryFormatter have fewer limitations than XmlSerializer. They can serialize private fields, for example. However they both require that the target class be marked with the [Serializable] attribute, so like XmlSerializer the class needs to be written with serialization in mind. Also there are some quirks to watch out for - for example on deserialization the constructor of the new object is not invoked.

The choice between SoapFormatter and BinaryFormatter depends on the application. BinaryFormatter makes sense where both serialization and deserialization will be performed on the .NET platform and where performance is important. SoapFormatter generally makes more sense in all other cases, for ease of debugging if nothing else.

**6.4 Can I customise the serialization process?**

Yes. XmlSerializer supports a range of attributes that can be used to configure serialization for a particular class. For example, a field or property can be marked with the [XmlIgnore] attribute to exclude it from serialization. Another example is the [XmlElement] attribute, which can be used to specify the XML element name to be used for a particular property or field.

Serialization via SoapFormatter/BinaryFormatter can also be controlled to some extent by attributes. For example, the [NonSerialized] attribute is the equivalent of XmlSerializer's [XmlIgnore] attribute. Ultimate control of the serialization process can be acheived by implementing the theISerializable interface on the class whose instances are to be serialized.

**6.5 Why is XmlSerializer so slow?**

There is a once-per-process-per-type overhead with XmlSerializer. So the first time you serialize or deserialize an object of a given type in an application, there is a significant delay. This normally doesn't matter, but it may mean, for example, that XmlSerializer is a poor choice for loading configuration settings during startup of a GUI application.

**6.6 Why do I get errors when I try to serialize a Hashtable?**

XmlSerializer will refuse to serialize instances of any class that implements IDictionary, e.g. Hashtable. SoapFormatter and BinaryFormatter do not have this restriction.

**6.7 XmlSerializer is throwing a generic "There was an error reflecting MyClass" error. How do I find out what the problem is?**

Look at the InnerException property of the exception that is thrown to get a more specific error message.

**6.8 Why am I getting an InvalidOperationException when I serialize an ArrayList?**

XmlSerializer needs to know in advance what type of objects it will find in an ArrayList. To specify the type, use the XmlArrayItemattibute like this:

public class Person

 {

public string Name;

publicint Age;

 }

public class Population

 {

 [XmlArrayItem(typeof(Person))] public ArrayList People;

 }

**7. Attributes**

**7.1 What are attributes?**

There are at least two types of .NET attribute. The first type I will refer to as a *metadata* attribute - it allows some data to be attached to a class or method. This data becomes part of the metadata for the class, and (like other class metadata) can be accessed via reflection. An example of a metadata attribute is [serializable], which can be attached to a class and means that instances of the class can be serialized.

 [serializable] public class CTest {}

The other type of attribute is a *context* attribute. Context attributes use a similar syntax to metadata attributes but they are fundamentally different. Context attributes provide an interception mechanism whereby instance activation and method calls can be pre- and/or post-processed. If you have encountered Keith Brown's universal delegator you'll be familiar with this idea.

**7.2 Can I create my own metadata attributes?**

Yes. Simply derive a class from System.Attribute and mark it with the AttributeUsage attribute. For example:

 [AttributeUsage(AttributeTargets.Class)]

public class InspiredByAttribute : System.Attribute

 {

public string InspiredBy;

publicInspiredByAttribute( string inspiredBy )

 {

InspiredBy = inspiredBy;

 }

 }

 [InspiredBy("Andy Mc's brilliant .NET FAQ")]

classCTest

 {

 }

classCApp

 {

public static void Main()

 {

object[] atts = typeof(CTest).GetCustomAttributes(true);

foreach( object att in atts )

if(att is InspiredByAttribute )

Console.WriteLine( "Class CTest was inspired by {0}", ((InspiredByAttribute)att).InspiredBy );

 }

 }

**7.3 Can I create my own context attibutes?**

Yes.

**8. Code Access Security**

**8.1 What is Code Access Security (CAS)?**

CAS is the part of the .NET security model that determines whether or not code is allowed to run, and what resources it can use when it is running. For example, it is CAS that will prevent a .NET web applet from formatting your hard disk.

**9. Intermediate Language (IL)**

**9.1 Can I look at the IL for an assembly?**

Yes. MS supply a tool called Ildasm that can be used to view the metadata and IL for an assembly.

**9.2 Can source code be reverse-engineered from IL?**

Yes, it is often relatively straightforward to regenerate high-level source from IL. Lutz Roeder's [Reflector](http://www.aisto.com/roeder/dotnet/) does a very good job of turning IL into C# or VB.NET.

**9.3 How can I stop my code being reverse-engineered from IL?**

You can buy an IL obfuscation tool. These tools work by 'optimising' the IL in such a way that reverse-engineering becomes much more difficult.

Of course if you are writing web services then reverse-engineering is not a problem as clients do not have access to your IL.

**9.4 Can I write IL programs directly?**

Yes. [Peter Drayton](http://www.razorsoft.net/) posted this simple example to the DOTNET mailing list:

 .assembly MyAssembly {}

 .class MyApp {

 .method static void Main() {

 .entrypoint

ldstr "Hello, IL!"

call void System.Console::WriteLine(class System.Object)

ret

 }

 }

Just put this into a file called hello.il, and then run ilasm hello.il. An exe assembly will be generated.

**9.5 Can I do things in IL that I can't do in C#?**

Yes. A couple of simple examples are that you can throw exceptions that are not derived from System.Exception, and you can have non-zero-based arrays.

**11. Threads**

**11.2 How do I stop a thread?**

There are several options. First, you can use your own communication mechanism to tell the ThreadStart method to finish. Alternatively the Thread class has in-built support for instructing the thread to stop. The two principle methods are Thread.Interrupt() and Thread.Abort(). The former will cause a ThreadInterruptedException to be thrown on the thread when it next goes into a WaitJoinSleep state. In other words, Thread.Interrupt is a polite way of asking the thread to stop when it is no longer doing any useful work. In contrast, Thread.Abort() throws a ThreadAbortException regardless of what the thread is doing. Furthermore, the ThreadAbortException cannot normally be caught (though the ThreadStart's finally method will be executed). Thread.Abort() is a heavy-handed mechanism which should not normally be required.

**11.3 How do I use the thread pool?**

By passing an instance of a WaitCallback delegate to the ThreadPool.QueueUserWorkItem() method

Class CApp

 {

static void Main()

 {

string s = "Hello, World";

ThreadPool.QueueUserWorkItem( new WaitCallback( DoWork ), s );

Thread.Sleep( 1000 ); // Give time for work item to be executed

 }

 // DoWork is executed on a thread from the thread pool.

static void DoWork( object state )

 {

Console.WriteLine( state );

 }

 }

**11.4 How do I know when my thread pool work item has completed?**

There is no way to query the thread pool for this information. You must put code into the WaitCallback method to signal that it has completed. Events are useful for this.

**11.6 Should I use ReaderWriterLock instead of Monitor.Enter/Exit?**

Maybe, but be careful. ReaderWriterLock is used to allow multiple threads to read from a data source, while still granting exclusive access to a single writer thread. This makes sense for data access that is mostly read-only, but there are some caveats. First, ReaderWriterLock is relatively poor performing compared to Monitor.Enter/Exit, which offsets some of the benefits. Second, you need to be very sure that the data structures you are accessing fully support multithreaded read access. Finally, there is apparently a bug in the v1.1 ReaderWriterLock that can cause starvation for writers when there are a large number of readers.

**12. Tracing**

**12.1 Is there built-in support for tracing/logging?**

Yes, in the System.Diagnostics namespace. There are two main classes that deal with tracing - Debug and Trace. They both work in a similar way - the difference is that tracing from the Debug class only works in builds that have the DEBUG symbol defined, whereas tracing from the Trace class only works in builds that have the TRACE symbol defined. Typically this means that you should use System.Diagnostics.Trace.WriteLine for tracing that you want to work in debug and release builds, and System.Diagnostics.Debug.WriteLine for tracing that you want to work only in debug builds.

**12.2 Can I redirect tracing to a file?**

Yes. The Debug and Trace classes both have a Listeners property, which is a collection of sinks that receive the tracing that you send via Debug.WriteLine and Trace.WriteLine respectively. By default the Listeners collection contains a single sink, which is an instance of the DefaultTraceListener class. This sends output to the Win32 OutputDebugString() function and also the System.Diagnostics.Debugger.Log() method. This is useful when debugging, but if you're trying to trace a problem at a customer site, redirecting the output to a file is more appropriate. Fortunately, the TextWriterTraceListener class is provided for this purpose.

Here's how to use the TextWriterTraceListener class to redirect Trace output to a file:

Trace.Listeners.Clear();

FileStreamfs = new FileStream( @"c:\log.txt", FileMode.Create, FileAccess.Write );

Trace.Listeners.Add( new TextWriterTraceListener( fs ) );

Trace.WriteLine( @"This will be writen to c:\log.txt!" );

Trace.Flush();

Note the use of Trace.Listeners.Clear() to remove the default listener. If you don't do this, the output will go to the file *and*OutputDebugString(). Typically this is not what you want, because OutputDebugString() imposes a big performance hit.

**12.4 Are there any third party logging components available?**

Log4net is a port of the established log4j Java logging component.

**13. Miscellaneous**

**13.1 How does .NET remoting work?**

.NET remoting involves sending messages along channels. Two of the standard channels are HTTP and TCP. TCP is intended for LANs only - HTTP can be used for LANs or WANs (internet).

Support is provided for multiple message serializarion formats. Examples are SOAP (XML-based) and binary. By default, the HTTP channel uses SOAP (via the .NET runtime Serialization SOAP Formatter), and the TCP channel uses binary (via the .NET runtime Serialization Binary Formatter). But either channel can use either serialization format.

There are a number of styles of remote access:

* **SingleCall**. Each incoming request from a client is serviced by a new object. The object is thrown away when the request has finished.
* **Singleton**. All incoming requests from clients are processed by a single server object.
* **Client-activated object**. This is the old stateful (D)COM model whereby the client receives a reference to the remote object and holds that reference (thus keeping the remote object alive) until it is finished with it.

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 .NET runtime remoting infrastructure. 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.

**13.2 How can I get at the Win32 API from a .NET program?**

Use P/Invoke. This uses similar technology to COM Interop, but is used to access static DLL entry points instead of COM objects. Here is an example of C# calling the Win32 MessageBox function:

using System;

usingSystem.Runtime.InteropServices;

classMainApp

 {

 [DllImport("user32.dll", EntryPoint="MessageBox", SetLastError=true, CharSet=CharSet.Auto)]

public static extern intMessageBox(inthWnd, String strMessage, String strCaption, uintuiType);

public static void Main()

 {

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

 }

 }

Pinvoke.net is a great resource for off-the-shelf P/Invoke signatures.

**13.4 What is the difference between an event and a delegate?**

An event is just a wrapper for a multicast delegate. Adding a public event to a class is almost the same as adding a public multicast delegate field. In both cases, subscriber objects can register for notifications, and in both cases the publisher object can send notifications to the subscribers. However, a public multicast delegate has the undesirable property that external objects can *invoke* the delegate, something we'd normally want to restrict to the publisher. Hence events - an event adds public methods to the containing class to add and remove receivers, but does not make the invocation mechanism public.

**13.5 What size is a .NET object?**

Each instance of a reference type has two fields maintained by the runtime - a method table pointer and a sync block. These are 4 bytes each on a 32-bit system, making a total of 8 bytes per object overhead. Obviously the instance data for the type must be added to this to get the overall size of the object. So, for example, instances of the following class are 12 bytes each:

classMyInt

 {

 ...

privateint x;

 }

However, note that with the current implementation of the CLR there seems to be a minimum object size of 12 bytes, even for classes with no data (e.g. System.Object).

Values types have no equivalent overhead.

**13.6 Will my .NET app run on 64-bit Windows?**

64-bit (x64) versions of Windows support both 32-bit and 64-bit processes, and corresponding 32-bit and 64-bit versions of .NET 2.0. (.NET 1.1 is 32-bit only).

.NET 1.x apps automatically run as 32-bit processes on 64-bit Windows.

.NET 2.0 apps can either run as 32-bit processes or as 64-bit processes. The OS decides which to use based on the PE header of the executable. The flags in the PE header are controlled via the compiler /platform switch, which allows the target of the app to be specified as 'x86', 'x64' or 'any cpu'. Normally you specify 'any cpu', and your app will run as 32-bit on 32-bit Windows and 64-bit on 64-bit Windows. However if you have some 32-bit native code in your app (loaded via COM interop, for example), you will need to specify 'x86', which will force 64-bit Windows to load your app in a 32-bit process. You can also tweak the 32-bit flag in the PE header using the SDK corflags utility.

**13.7 What is reflection?**

All .NET compilers produce metadata about the types defined in the modules they produce. This metadata is packaged along with the module (modules in turn are packaged together in assemblies), and can be accessed by a mechanism called **reflection**. The System.Reflection namespace contains classes that can be used to interrogate the types for a module/assembly.

Using reflection to access .NET metadata is very similar to using ITypeLib/ITypeInfo to access type library data in COM, and it is used for similar purposes - e.g. determining data type sizes for marshaling data across context/process/machine boundaries.

Reflection can also be used to dynamically invoke methods (see System.Type.InvokeMember), or even create types dynamically at run-time (see System.Reflection.Emit.TypeBuilder).