**CLR Basics**

**CLR’s execution model**

**Shared assemblies and strong named assemblies**

**Working with Types**

**Type Fundamentals**

**Primitive, Reference and Value Types**

**Designing Types**

**Type and member basics**

**Constants and Fields**

**Methods: Constructors, Operators**

**Events**

**Essential Types**

**Array**

**Strings**

**Delegates**

**Interface**

**Generics**

**Collections**

**Attributes**

**Reflection**

**CLR facilities**

**Exceptions**

**Automatic Memory Management**

**CLR Hosting and Appdomains**

**Threading**

**CLR Basics**

**CLR’s execution model**

**Shared assemblies and strong named assemblies**

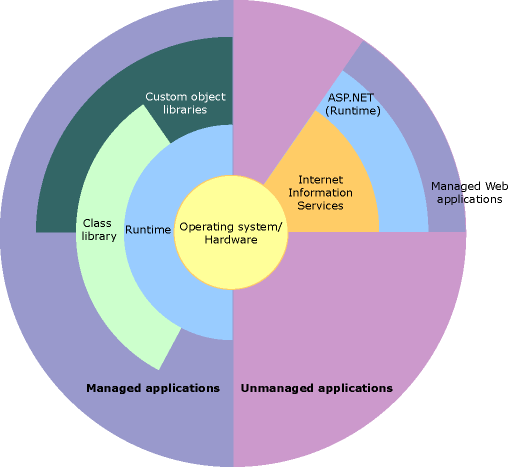
**CLR’s execution model**

#### 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.

The following illustration shows the relationship of the common language runtime and the class library to your applications and to the overall system. The illustration also shows how managed code operates within a larger architecture.

**.NET Framework in context**

****

#### What is the common language runtime (CLR)? What are it’s features?

The common language runtime is the *execution engine* for .NET Framework applications. The CLR allows programmers to ignore many details of the specific CPU that will execute the program. It also provides other important services, including the following:

* Memory management
* Thread management
* Exception handling
* Garbage collection
* Security

**Explain the CLR execution model.**

The managed execution process includes the following steps:

1. Choosing a compiler.

To obtain the benefits provided by the common language runtime, you must use one or more language compilers that target the runtime.

1. Compiling your code to Microsoft intermediate language (MSIL).

Compiling translates your source code into MSIL and generates the required metadata.

1. Compiling MSIL to native code.

At execution time, a just-in-time (JIT) compiler translates the MSIL into native code. During this compilation, code must pass a verification process that examines the MSIL and metadata to find out whether the code can be determined to be type safe. Alternatively, the CIL code can be compiled to native code in a separate step prior to runtime by using the Native Image Generator (NGEN). This speeds up all later runs of the software as the CIL-to-native compilation is no longer necessary.

1. Running code.

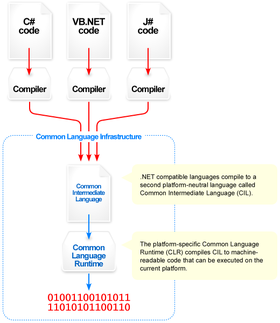
The common language runtime provides the infrastructure that enables execution to take place as well as a variety of services that can be used during execution.

#### 

#### 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 CLI is documented through ECMA - see http://msdn.microsoft.com/net/ecma/ for more details.

#### The CLR (Common Language Runtime) is Microsoft's primary *implementation* of the CLI. Microsoft also has a shared source implementation known as ROTOR, for educational purposes, as well as the .NET Compact Framework for mobile devices. Non-Microsoft CLI implementations include Mono and DotGNU Portable.NET.



#### 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.

**Figure 1-6** Languages offer a subset of the CLR/CTS and a superset of the CLS (but not necessarily the same superset)

#### 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*. *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.

**What is JIT?**

*JIT is a compiler that converts MSIL to native code. The native code consists of hardware specific instructions that can be executed by the CPU.*

Rather than converting the entire MSIL (in a portable executable [PE] file) to native code, the JIT converts the MSIL as it is needed during execution. This converted native code is stored so that it is accessible for subsequent calls.

**What are different types of JIT?**

JIT compiler is a part of the runtime execution environment. In Microsoft .NET there are three types of JIT compilers:

* **Pre -JIT**: Pre-JIT compiles complete source code into native code in a single compilation cycle. This is done at the time of deployment of the application.
* **Econo -JIT** : Econo-JIT compiles only those methods that are called at runtime. However, these compiled methods are removed when they are not required.
* **Normal -JIT** : Normal-JIT compiles only those methods that are called at runtime. These methods are compiled the first time they are called, and then they are stored in cache. When the same methods are called again, the compiled code from cache is used for execution.

**What is portable executable (PE)?***PE is the file format defining the structure that all executable files (EXE) and Dynamic Link Libraries (DLL) must use to allow them to be loaded and executed by Windows.* PE is derived from the Microsoft Common Object File Format (COFF). The EXE and DLL files created using the .NET Framework obey the PE/COFF formats and also add additional header and data sections to the files that are only used by the CLR.

#### Assembly

#### 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 static assemblies and dynamic assemlies.Differences between them?**

Assemblies can be static or dynamic. Static assemblies can include .NET Framework types (interfaces and classes), as well as resources for the assembly (bitmaps, JPEG files, resource files, and so on). Static assemblies are stored on disk in portable executable (PE) files. You can also use the .NET Framework to create dynamic assemblies, which are run directly from memory and are not saved to disk before execution. You can save dynamic assemblies to disk after they have executed.

There are several ways to create assemblies. You can use development tools, such as Visual Studio 2005, that you have used in the past to create .dll or .exe files. You can use tools provided in the Windows Software Development Kit (SDK) to create assemblies with modules created in other development environments. You can also use common language runtime APIs, such as [Reflection.Emit](http://msdn.microsoft.com/en-us/library/system.reflection.emit.aspx), to create dynamic assemblies.

**What are the assembly contents?**

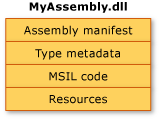
In general, a static assembly can consist of four elements:

* Assembly manifest - Contains the assembly metadata. An assembly manifest contains the information about the identity and version of the assembly. It also contains the information required to resolve references to types and resources.
* Type metadata - Binary information that describes a program.
* Microsoft intermediate language (MSIL) code.
* A set of resources.

Only the assembly manifest is required, but either types or resources are needed to give the assembly any meaningful functionality.

There are several ways to group these elements in an assembly. You can group all elements in a single physical file, which is shown in the following illustration.

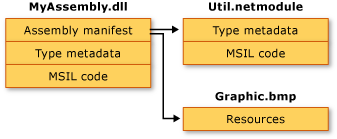
Single-file assembly



Alternatively, the elements of an assembly can be contained in several files. These files can be modules of compiled code (.netmodule), resources (such as .bmp or .jpg files), or other files required by the application. Create a multifile assembly when you want to combine modules written in different languages and to optimize downloading an application by putting seldom used types in a module that is downloaded only when needed.

In the following illustration, the developer of a hypothetical application has chosen to separate some utility code into a different module and to keep a large resource file (in this case a .bmp image) in its original file. The .NET Framework downloads a file only when it is referenced; keeping infrequently referenced code in a separate file from the application optimizes code download.

Multifile assembly



| **Note** |
| --- |
| The files that make up a multifile assembly are not physically linked by the file system. Rather, they are linked through the assembly manifest and the common language runtime manages them as a unit. |

In this illustration, all three files belong to an assembly, as described in the assembly manifest contained in MyAssembly.dll. To the file system, they are three separate files. Note that the file Util.netmodule was compiled as a module because it contains no assembly information. When the assembly was created, the assembly manifest was added to MyAssembly.dll, indicating its relationship with Util.netmodule and Graphic.bmp.

As you currently design your source code, you make explicit decisions about how to partition the functionality of your application into one or more files. When designing .NET Framework code, you will make similar decisions about how to partition the functionality into one or more assemblies.

#### 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.

**How should we add an assembly reference?**  
  
If a DLL project is referenced in an application, using Add Reference--> Projects, the referenced DLL has the source path as one of the following:

DLL in the release folder.

Note: DotNet does not considers whether the .dll is available in the release folder or not.

Incase release folder is not available then the .dll will refer the debug folder.

If the reference file is selected explicitly to point debug or release the selected folder is referenced.

**What is a strong name?**

You need to assign a strong name to an assembly to place it in the GAC and make it globally accessible. *A strong name consists of a name that consists of an assembly's identity (text name, version number, and culture information), a public key and a digital signature generated over the assembly.*  The .NET Framework provides a tool called the Strong Name Tool (Sn.exe), which allows verification and key pair and signature generation.

**What is GAC? What are the steps to create an assembly and add it to the GAC?**

*The global assembly cache (GAC) is a machine-wide code cache (a place holder or container) that stores assemblies specifically designated to be shared by several applications on the computer.* You should share assemblies by installing them into the global assembly cache only when you need to.

Steps:

- Create a strong name using sn.exe tool eg: sn -k mykey.snk

- in AssemblyInfo.cs, add the strong name eg: [assembly:assemblyKeyFile("mykey.snk")]

- recompile project, and then install it to GAC in two ways :

·         drag & drop it to assembly folder (C:\WINDOWS\assembly OR C:\WINNT\assembly) (shfusion.dll tool)

·         gacutil -i abc.dll

**Can I delete the source file which I have used to register in GAC?**

Yes, the assembly makes a copy in the GAC folder. So the source file can be deleted.

**Can I specify the space used by GAC?**

Yes, Navigate to the GAC directory, C:\winnt\Assembly in explore. In the tools menu select the cache properties; in the windows displayed you can set the memory limit in MB used by the GAC.

**How should an assembly be uninstalled?**  
  
Uninstalling an assembly should be done with care.

The following command removes the assembly hello from the global assembly cache (GAC) as long as no reference counts exist for the assembly.

gacutil /u hello

Note : If there is only one version of hello assembly the above command is fine and you are in safer side. Incase if there are more than one version of the assembly or different assembly with same name exists then the above command might remove more than one assembly from the assembly cache because the assembly name is not fully specified. For example, if both version 1.0.0.0 and 3.2.2.1 of hello are installed in the cache, the command gacutil /u hello removes both of the assemblies.

Then how to remove the assembly safely: Use the following example to avoid removing more than one assembly. This command removes only the hello assembly that matches the fully specified version number, culture, and public key.  
gacutil /u hello, Version=1.0.0.1, Culture="de", PublicKeyToken=45e343aae32233ca

**Why I should use Assemblies?**

The .NET Framework uses assemblies as the fundamental unit for several purposes:

·         Security   
·         Type Identity   
·         Reference Scope   
·         Versioning   
·         Deployment

**What is Versioning?**  
  
*Each assembly has a 128-bit version number that is presented as a set of four decimal pieces: Major.Minor.Build.Revision*

For example, an assembly might have the version number 3.5.0.126.

By default, an assembly will only use types from the exact same assembly (name and version number) that it was built and tested with. That is, if you have an assembly that uses a type from version 1.0.0.2 of another assembly, it will (by default) not use the same type from version 1.0.0.4 of the other assembly. This use of both name and version to identify referenced assemblies helps avoid the "DLL Hell" problem of upgrades to one application breaking other applications.

Tip   An administrator or developer can use configuration files to relax this strict version checking. Look for information on publisher policy in the .NET Framework Developer's Guide.

**Can my assembly span more that one file?**

Yes, assembly can have more that one file. Each file can be developed from different language too. Using the ***al.exe*** utility all the required files are grouped and made into single file.

**How do I decide whether my assembly should contain only one DLL or a collection of dlls?**

A typical assembly is a single DLL described by an assembly manifest. An assembly may contain multiple files. In cases where components have tight interdependencies such that if you make changes to one component the other would also change, it makes sense to include the group of files in the same assembly.

When deciding what to include in your assembly, keep in mind how you intend to manage these in the future, and how you want applications to use them. For example, if you include two files in an assembly, each time you ship a new version of the assembly, you would include both files, whether or not both have changed. If you have two files with a strong interdependency, and you decide to ship as separate assemblies, you need to be aware that customers have a choice of mixing versions. If you update both assemblies, a customer may choose to use the new version of one and the older version of the other, which may or may not be compatible

**Which type of reference is advantageous during development and why? Referencing the exact dll path or the project itself in a solution with more than one project, with one or more class lib project?**

Referring, project itself is right and advantages, for the following reasons:

They automatically track project configuration changes. For example, when you build using a debug configuration, any project references refer to debug assemblies generated by the referenced projects, while they refer to release assemblies in a release configuration. This means that you can automatically switch from debug to release builds across projects without having to reset references.

**What are the types of caches available in .Net ?**

1)     GAC, or Global Assembly Cache - all shared assemblies live here   
2)     Download Cache - when you execute an assembly from a URL, this is where the downloaded assemblies end up  
3)     'ZAP' Cache - this cache seems to serve as the home for pre-compiled native assembly images that are produced by NGEN

**What is called probing?**

Private assemblies are deployed within the directory structure of the application in which they are used. Private assemblies can be placed directly in the application directory, or in a subdirectory thereof. The CLR finds these assemblies through a process called probing. Probing is simply a mapping of the assembly name to the name of the file that contains the manifest.

**If the same assembly is used by two app, what will be stored in GAC and how it is stored?**

In GAC, only one master copy of the dll is stored. But if another application is installing the same assembly (with the same version) a reference with a native image is created.

So if we have a dll, which is used by three applications, there will be one master and two images in the GAC.

You can see this when you check the GAC folder.  In COM you can't have same dlls with different versions.

**What do you call as DLL Hell in COM and how was that fixed in .Net?**

Let’s consider a shared dll used by msn messenger and your own application. When you upgrade MSN Messenger, a new version of shared dll will be installed which may not be compatible with your application, which leads in failure of your application. This scenario is described as DLL Hell.

In .Net the problem is fixed using GAC - shared assembly, where you can have more than one version of the same dll. If new version of msn is installed, the new version of the dll in installed in GAC with the new version number. *The important thing in GAC is that, more than one version of same dll can co-exist, so older version of dll, which is referenced by your application, will not get affected.*

**How to override the assembly name and version combination key while selecting and using the assembly in the GAC?**

By default the assembly for the application is binded during installation.

The assembly is selected by name and the version from the GAC, sometimes we require our application to use another version of the same assembly. For example

An administrator may deploy a critical bug fix to a shared assembly and want all applications to use this new version regardless of which version they were built with. Also, the vendor of a shared assembly may have shipped a service release to an existing assembly and would like all applications to begin using the service release instead of the original version. These scenarios and others are supported in the .NET Framework through version policies.

Version policies are stated in XML files and are simply a request to load one version of assembly instead of another. For example, the following version policy directs the CLR to load version 5.0.0.1 instead of version 5.0.0.0 of an assembly called MarineCtrl:

<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">  
<dependentAssembly>  
<assemblyIdentity name="MarineCtrl" publicKeyToken="9335a2124541cfb9" />  
<bindingRedirect oldVersion="5.0.0.0" NewVersion="5.0.0.1" />  
</dependentAssembly>  
</assemblyBinding>

In addition to redirecting from a specific version number to another, you can also redirect from a range of versions to another version. For example, the following policy redirects all versions from 0.0.0.0 through 5.0.0.0 of MarineCtrl to version 5.0.0.1:

<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">  
<dependentAssembly>  
<assemblyIdentity name="MarineCtrl" publicKeyToken="9335a2124541cfb9" />  
<bindingRedirect oldVersion="0.0.0.0-5.0.0.0" newVersion="5.0.0.1" />  
</dependentAssembly>  
</assemblyBinding>

**What are the types of version policy levels?**

Application-specific Policy. Each application has an optional configuration file that can specify the application’s desire to bind to a different version of a dependent assembly. The name of the configuration file varies based on the application type. For executable files, the name of the configuration file is the name of the executable plus a ".config" extension. For example, the configuration file for "myapp.exe" would be "myapp.exe.config". Configuration files for ASP.NET applications are always "web.config".

Publisher Policy. While application-specific policy is set either by the application developer or administrator, publisher policy is set by the vendor of the shared assembly. Publisher policy is the vendor’s statement of compatibility regarding different versions of her assembly. For example, say the vendor of a shared Windows Forms control ships a service release that contains a number of bug fixes to the control. The original control was version 2.0.0.0 and the version of the service release is 2.0.0.1. Because the new release just contains bug fixes (no breaking API changes) the control vendor would likely issue publisher policy with the new release that causes existing applications that used 2.0.0.0 to now start using 2.0.0.1. Publisher policy is expressed in XML just as application and machine-wide policy are, but unlike those other policy levels, publisher policy is distributed as an assembly itself. The primary reason for this is to ensure that the organization releasing the policy for a particular assembly is the same organization that released the assembly itself. This is accomplished by requiring that both the original assembly and the policy assembly are given a strong name with the same key-pair.

Machine-wide Policy. The final policy level is machine-wide policy (sometimes referred to as Administrator policy). Machine-wide policy is stored in machine.config which is located in the "config" subdirectory under the .NET Framework install directory. The install directory is %windir%\microsoft.net\framework\%runtimeversion%. Policy statements made in machine.config affect all applications running on the machine. Machine-wide policy is used by Administrators to force all applications on a given machine to use a particular version of an assembly. The most common scenario in which this is used is when a security or other critical bug fix has been deployed to the global assembly cache. After deploying the fixed assembly, the Administrator would use machine-wide version policy to ensure that applications don’t use the old, broken version of the assembly.

**What is Native image file for an assembly?**

*A native image is a file containing compiled processor-specific machine code. Note* that the native image that Ngen.exe generates cannot be shared across Application Domains. Therefore, you cannot use Ngen.exe in application scenarios, such as ASP.NET, that require assemblies to be shared across application domains.

Pre-compiling assemblies with Ngen.exe can improve the startup time for applications, because much of the work required to execute code has been done in advance. Therefore, it is more appropriate to use Ngen.exe for client-side applications where you have determined that the CPU cycles consumed by JIT compilation cause slower performance.

**What is the caspol.exe tool used for?**

The caspol tool grants and modifies permissions to code groups at the user policy, machine policy, and enterprise policy levels.

**What is Ildasm.exe used for?**

Ildasm.exe is a tool that generates PE files from MSIL code. You can run the resulting executable to determine whether the MSIL code performs as expected.

Ildasm.exe is a tool that takes a PE file containing the MSIL code as a parameter and creates a text file that contains managed code.

**What is the ResGen.exe tool used for?**

ResGen.exe is a tool that is used to convert resource files in the form of .txt or .resx files to common language runtime binary .resources files that can be compiled into satellite assemblies.

**Can I limit the space used by GAC?**

Yes, the size of the GAC can be modified as per the requirements. To set the size, browse through the gac folder %windir%\Assembly\ in windows explorer.

Select ToolsàCache Options from the menu to set the maximum size of the disk space to be allotted for GAC.

**How do I see the physical file structure of cache in explorer?**  
It is possible to see physical file structure of GAC folder.

As such Winnt\assembly\gac cannot be browsed using windows explorer.

To view the physical file structure add a binary value named *'DisableCacheViewer'* to the registry key HKLM\Software\Microsoft\Fusion and set it to a non-zero value.

**How is the DLL Hell problem solved in .NET?**

Assembly versioning allows the application to specify not only the library it needs to run (which was available under Win32), but also the version of the assembly. 

**What are the ways to deploy an assembly?**

An MSI installer, a CAB archive, and XCOPY command. 

**Where we can use DLL made in C#.Net?**

Supporting .Net, because DLL made in C#.Net semi compiled version. It’s not a com object. It is used only in .Net Framework As it is to be compiled at runtime to byte code.

**What is a satellite assembly?**

When you write a multilingual or multi-cultural application in .NET, and want to distribute the core application separately from the localized modules, the localized assemblies that modify the core application are called satellite assemblies. 

**What is ResourceManager class?**

ResourceManager class helps us to read the resource files and get the values using key. First you need to create the object of resource manager. You need to specify the resource name and the assembly in the constructor.

private ResourceManager objResourceManager = new ResourceManager ("Globalization.resource",System.Reflection.Assembly.GetExecutingAssembly());

Once the resource manager is populated with details you can then use the GetString function to get by key. For instance in the below code snippet we are using the “cmdAddNew” key to get the value for button “cmdAddNew””.

cmdAddNew.Text = objResourceManager.GetString("cmdAddNew");

**How do we deploy satellite assemblies?**

When we deploy the assembly, the folder structure has to very organized. MainFolder is the main application folder. All satellite assemblies should be deployed in the Main application folder with in there own respective folder. The respective folder is denoted by the culture code (/en-ca, /hi, /el etc.).

If the program does not find resource file for a culture it uses the invariant culture satellite assembly. The above folder structure is a strict requirement when we deploy the satellite assembly. Any mismatch in the folder structure will lead to in appropriate results.

**How to prevent my .NET DLL to be decompiled?**

By design .NET embeds rich Meta data inside the executable code using MSIL. Anyone can easily decompile your DLL back using tools like ILDASM (owned by Microsoft) or third-party tools like Reflector for .NET. So any one can easily look in to your assemblies and reverse engineer them back in to actual source code and understand some real good logic which can make it easy to crack your application.

The process by which you can stop this reverse engineering is using “obfuscation”. It’s a technique which will foil the decompilers. There are many third parties (XenoCode, Demeanor for .NET) which provide .NET obfuscation solution. Microsoft includes one that is Dotfuscator Community Edition with Visual Studio.NET.

**If we have two version of same assembly in GAC how do we make a choice ?**

You need to specify “bindingRedirect” in your config file. For instance in the below case “ClassLibraryVersion” has two versions “1.1.1″ and “1.1.2″ from which “1.1.2 is the recent version. But using the bindingRedirect we can specify saying “1.1.1″ is the new version. So the client will not use “1.1.2″.

<configuration>  
 <runtime>  
 <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">  
 <dependentAssembly>  
 <assemblyIdentity name="ClassLibraryVersion" publicKeyToken="b035c4774706cc72" culture="neutral" />  
 <bindingRedirect oldVersion= "1.1.2" newVersion= "1.1.1"/>  
 </dependentAssembly>  
 </assemblyBinding>  
 </runtime>  
</configuration>

### What is the smallest unit of execution in .NET?

### An assembly.

### 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.

**How can I make sure my C# classes will interoperate with other .Net languages?**

Make sure your C# code conforms to the Common Language Subset (CLS). To help with this, add the [assembly: CLSCompliant (true)] global attribute to your C# source files. The compiler will emit an error if you use a C# feature which is not CLS-compliant.

**Is it possible to debug the classes written in other .Net languages in a C# project?**

It is definitely possible to debug other .Net languages code in a C# project. As everyone knows .net can combine code written in several .net languages into one single assembly. Same is true with debugging.

**Does C# have its own class library?**

Not exactly. The .NET Framework has a comprehensive class library, which C# can make use of. C# does not have its own class library.

**How do you convert a value-type to a reference-type?**  
Use Boxing.

**What happens in memory when you Box and Unbox a value-type?**  
Boxing converts a value-type to a reference-type, thus storing the object on the heap.  Unboxing converts a reference-type to a value-type, thus storing the value on the stack.

**What are the differences between .NET dll and .NET exe?**

.Exe  
1. These are outbound file.  
2. Only one .exe file exists per application.  
3. Exe cannot be shared with other applications.  
4. exe is an executable program.It runs on its own

5. Exe has main Entry point available

.dll  
1. These are inbund file .  
2. Many .dll files may exists in one application.  
3. dll can be shared with other applications.

4. dll is a dynamic link library located at run time using PATH env variable. It is linked or referenced to the exe at run time

5. There is no any entry point available in a dll.

"First we need to be clear that both "exe" and "dll" are fundamentally the same but the difference lies in how windows interacts with them."   
  
When windows loads a dll, it runs the initialization code and then leaves it alone. Functions in the dll are called if they are explicitly referenced by an application. Another thing, when dll gets crashed it not only crashes itself but also the application as the dll runs in the memory of the parent application.   
  
When windows load an exe, the exe's initialization code is responsible for creating what is called as "message pump", nothing but a program loop which runs as long as the application is running. The message pump request messages from the operating system. Windows keep track of the application as a separate task. It allocates separate memory for both the exe and the application using that exe. The memory area in which each exe runs is called "Process Space".

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Working with Types**

**Type Fundamentals**

**Primitive, Reference and Value Types**

**Type Fundamentals**

**What is the use/advantage of Object class?**

It supports all classes in the .NET Framework class hierarchy and provides low-level services to derived classes. This is the ultimate base class of all classes in the .NET Framework; it is the root of the type hierarchy.

**Namespace:** System  
**Assembly:** mscorlib (in mscorlib.dll)

Languages typically do not require a class to declare inheritance from **Object** because the inheritance is implicit. Reference types inherit the object class either directly or through other reference types.  Value types inherit implicitly from the object class through System.ValueType.

**What are the members of Object class?**

The following tables list the members exposed by the Object type.

 Public Constructors

|  | **Name** | **Description** |
| --- | --- | --- |
|  | [Object](http://msdn.microsoft.com/en-us/library/system.object.object(VS.80).aspx) | Initializes a new instance of the [Object](http://msdn.microsoft.com/en-us/library/system.object(VS.80).aspx) class. |

 Public Methods

|  | **Name** | **Description** |
| --- | --- | --- |
|  | [Equals](http://msdn.microsoft.com/en-us/library/system.object.equals(VS.80).aspx) | Overloaded. Determines whether two **Object** instances are equal. |
|  | [GetHashCode](http://msdn.microsoft.com/en-us/library/system.object.gethashcode(VS.80).aspx) | Serves as a hash function for a particular type. [GetHashCode](http://msdn.microsoft.com/en-us/library/system.object.gethashcode(VS.80).aspx) is suitable for use in hashing algorithms and data structures like a hash table. |
|  | [GetType](http://msdn.microsoft.com/en-us/library/system.object.gettype(VS.80).aspx) | Gets the [Type](http://msdn.microsoft.com/en-us/library/system.type(VS.80).aspx) of the current instance. |
|  | [ReferenceEquals](http://msdn.microsoft.com/en-us/library/system.object.referenceequals(VS.80).aspx) | Determines whether the specified **Object** instances are the same instance. |
|  | [ToString](http://msdn.microsoft.com/en-us/library/system.object.tostring(VS.80).aspx) | Returns a [String](http://msdn.microsoft.com/en-us/library/system.string(VS.80).aspx) that represents the current **Object**. |

 Protected Methods

|  | **Name** | **Description** |
| --- | --- | --- |
|  | [Finalize](http://msdn.microsoft.com/en-us/library/system.object.finalize(VS.80).aspx) | Allows an **Object** to attempt to free resources and perform other cleanup operations before the **Object** is reclaimed by garbage collection. |
|  | [MemberwiseClone](http://msdn.microsoft.com/en-us/library/system.object.memberwiseclone(VS.80).aspx) | Creates a shallow copy of the current **Object**. |

**Give an example of a type that shows how to use exposed members.**

The following example defines a Point type derived from the **Object** class and overrides many of the virtual methods of the **Object** class. In addition, the example shows how to call many of the static and instance methods of the **Object** class.

using System;

// The Point class is derived from System.Object.

class Point

{

public int x, y;

public Point(int x, int y)

{

this.x = x;

this.y = y;

}

public override bool Equals(object obj)

{

// If this and obj do not refer to the same type, then they are not equal.

if (obj.GetType() != this.GetType()) return false;

// Return true if x and y fields match.

Point other = (Point) obj;

return (this.x == other.x) && (this.y == other.y);

}

// Return the XOR of the x and y fields.

public override int GetHashCode()

{

return x ^ y;

}

// Return the point's value as a string.

public override String ToString()

{

return String.Format("({0}, {1})", x, y);

}

// Return a copy of this point object by making a simple field copy.

public Point Copy()

{

return (Point) this.MemberwiseClone();

}

}

public sealed class App {

static void Main()

{

// Construct a Point object.

Point p1 = new Point(1,2);

// Make another Point object that is a copy of the first.

Point p2 = p1.Copy();

// Make another variable that references the first Point object.

Point p3 = p1;

// The line below displays false because p1 and p2 refer to two different objects.

Console.WriteLine(Object.ReferenceEquals(p1, p2));

// The line below displays true because p1 and p2 refer to two different objects that have the same value.

Console.WriteLine(Object.Equals(p1, p2));

// The line below displays true because p1 and p3 refer to one object.

Console.WriteLine(Object.ReferenceEquals(p1, p3));

// The line below displays: p1's value is: (1, 2)

Console.WriteLine("p1's value is: {0}", p1.ToString());

}

}

// This code produces the following output.

//

// False

// True

// True

// p1's value is: (1, 2)

**[Why is it important to override GetHashCode when Equals method is overriden in C#?](http://stackoverflow.com/questions/371328/why-is-it-important-to-override-gethashcode-when-equals-method-is-overriden-in-c)**

It is because the framework requires that two objects that are the same must have the same hashcode. If you override the Equals method to do a special comparison of two objects and the two objects are considered the same by the method, then the hash code of the two objects must also be the same. (Dictionaries and Hashtables rely on this principle).

**What is the advantage of overriding** [**System.Object.Equals**](http://msdn.microsoft.com/en-us/library/system.object.equals(VS.80).aspx) **and** [**System.Object.GetHashCode**](http://msdn.microsoft.com/en-us/library/system.object.gethashcode(VS.80).aspx) **methods by a public type?**

GetHashCode returns a value based on the current instance that is suited for hashing algorithms and data structures such as a hash table. The advantage is that when two objects that are the same type and are equal would return the same hash code. This would ensure that instances of System.Collections.HashTable and System.Collections.Generic.Dictionary<TKey, TValue> work correctly.

**If A.equals (B) is true then A.getHashcode & B.gethashcode must always return same hash code?**

The answer is False because it is given that A.equals(B) returns true i.e. objects are equal and now its hashCode is asked which is always independent of the fact that whether objects are equal or not. So, GetHashCode for both of the objects returns different value.

**What is the difference between == and object.Equals?**

*For value types, == and Equals() usually compare two objects by value.* For example:

int x = 10;

int y = 10;

Console.WriteLine( x == y );

Console.WriteLine ( x.Equals(y) );

will display:

True

True

However things are more complex for reference types. *Generally speaking, for reference types == is expected to perform an identity comparison, i.e. it will only return true if both references point to the same object. By contrast, Equals() is expected to perform a value comparison, i.e. it will return true if the references point to objects that are equivalent.* For example:

StringBuilder s1 = new StringBuilder("fred");

StringBuilder s2 = new StringBuilder("fred");

Console.WriteLine( s1 == s2 );

Console.WriteLine( s1.Equals(s2) );

will display:

False

True

s1 and s2 are different objects (hence == returns false), but they are equivalent (hence Equals() returns true).Unfortunately there are exceptions to these rules. The implementation of Equals() in System.Object (the one you'll inherit by default if you write a class) compares identity, i.e. it's the same as operator ==. So Equals() only tests for equivalence if the class author overrides the method (and implements it correctly). Another exception is the string class - its operator == compares value rather than identity.

*Bottom line: If you want to perform an identity comparison use the ReferenceEquals() method. If you want to perform a value comparison, use Equals() but be aware that it will only work if the type has overridden the default implementation. Avoid operator == with reference types (except perhaps strings), as it's simply too ambiguous.*

**What is the advantage of all types derived from System.Object?**

Every object in the CLR derives from System.Object. Object is the base type of every type. In C#, the object keyword is an alias for System.Object. It can be convenient that every type in the CLR and in C# derives from Object. For example, you can treat a collection of instances of multiple types homogenously simply by casting them to Object references.

**Is it possible for partial-type definitions meant to be parts of the same type be defined in the separate DLLs?** <http://msdn.microsoft.com/en-us/library/wa80x488.aspx>

All partial-type definitions meant to be parts of the same type must be defined in the same assembly and the same module (.exe or .dll file). Partial definitions cannot span multiple modules.

**Primitive, Value Types and Reference Types**

**What are the 2 types of data types available in C#?  
1.** Value Types  
**2.** Reference Types  
  
**If you define a user defined data type by using the struct keyword, Is it a a value type or reference type?**   
Value Type  
  
**If you define a user defined data type by using the class keyword, Is it a a value type or reference type?**   
Reference type

**Are Value types sealed?**Yes, Value types are sealed.  
  
**What is the base class from which all value types are derived?**   
System.ValueType  
  
**How big is the datatype int in .NET?** 32 bits.

**How big is the char?** 16 bits (Unicode).

**How do you initiate a string without escaping each backslash?**

Put an @ sign in front of the double-quoted string.

**How do you convert a string into an integer in .NET?**

Int32.Parse(string), Convert.ToInt32()

**How do you box a primitive data type variable?**

Initialize an object with its value, pass an object, cast it to an object

**Why do you need to box a primitive variable?**

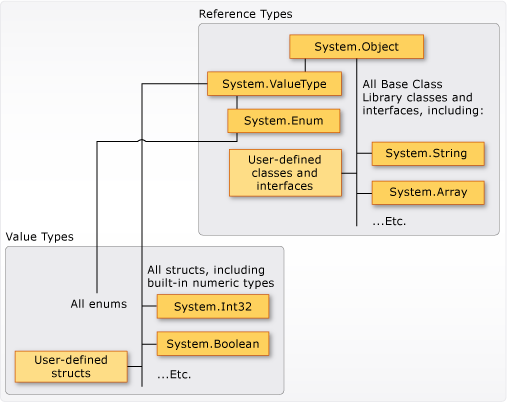
To pass it by reference or apply a method that an object supports, but primitive doesn’t.

**Give examples for value types?**Enum  
Struct  
  
**Give examples for reference types?**   
Class  
Delegate  
Array  
Interface  
  
**How value types and reference types related in the CTS?**

It is important to understand two fundamental points about the type system in the .NET Framework:

* It supports the principle of inheritance. Types can derive from other types, called base types. The derived type inherits (with some restrictions) the methods, properties, and other members of the base type. The base type can in turn derive from some other type, in which case the derived type inherits the members of both base types in its inheritance hierarchy. All types, including built-in numeric types such as [System.Int32](http://msdn.microsoft.com/en-us/library/system.int32.aspx) (C# keyword: [int](http://msdn.microsoft.com/en-us/library/5kzh1b5w.aspx)), derive ultimately from a single base type, which is [System.Object](http://msdn.microsoft.com/en-us/library/system.object.aspx) (C# keyword: [object](http://msdn.microsoft.com/en-us/library/9kkx3h3c.aspx)). This unified type hierarchy is called the [Common Type System](http://msdn.microsoft.com/en-us/library/zcx1eb1e.aspx) (CTS). For more information about inheritance in C#, see [Inheritance (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/ms173149.aspx).
* Each type in the CTS is defined as either a value type or a reference type. This includes all custom types in the .NET Framework class library and also your own user-defined types. Types that you define by using the [struct](http://msdn.microsoft.com/en-us/library/ah19swz4.aspx) keyword are value types; all the built-in numeric types are structs. Types that you define by using the [class](http://msdn.microsoft.com/en-us/library/0b0thckt.aspx) keyword are reference types. Reference types and value types have different compile-time rules, and different run-time behavior.

The following illustration shows the relationship between value types and reference types in the CTS.



**What are the differences between value types and reference types?  
1.** Value types are stored on the stack where as reference types are stored on the managed heap.  
**2.** Value type variables directly contain their values where as reference variables holds only a reference to the location of the object that is created on the managed heap.  
**3.** There is no heap allocation or garbage collection overhead for value-type variables. As reference types are stored on the managed heap, they have the over head of object allocation and garbage collection.  
**4.** Value types cannot inherit from another class or struct. Value types can only inherit from interfaces. Reference types can inherit from another class or interface.

**What do you mean by casting a data type?**   
Converting a variable of one data type to another data type is called casting. This is also called as data type conversion.  
  
**What are the 2 kinds of data type conversions in C#?  
Implicit conversions:** No special syntax is required because the conversion is type safe and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes.  
  
**Explicit conversions:** Explicit conversions require a cast operator. The source and destination variables are compatible, but there is a risk of data loss because the type of the destination variable is a smaller size than (or is a base class of) the source variable.  
  
**What is the difference between an implicit conversion and an explicit conversion?**   
**1.** Explicit conversions require a cast operator where as an implicit converstion is done automatically.  
**2.** Explicit conversion can lead to data loss where as with implicit conversions there is no data loss.  
  
**What type of data type conversion happens when the compiler encounters the following code?**ChildClass CC = new ChildClass();  
ParentClass PC = new ParentClass();  
  
Implicit Conversion. For reference types, an implicit conversion always exists from a class to any one of its direct or indirect base classes or interfaces. No special syntax is necessary because a derived class always contains all the members of a base class.  
  
**Will the following code compile?**   
double d = 9999.11;  
int i = d;  
  
No, the above code will not compile. Double is a larger data type than integer. An implicit conversion is not done automatically bcos there is a data loss. Hence we have to use explicit conversion as shown below.  
  
double d = 9999.11;  
int i = (int)d; //Cast double to int.  
  
**If you want to convert a base type to a derived type, what type of conversion do you use?**Explicit conversion as shown below.  
//Create a new derived type.  
Car C1 = new Car();  
// Implicit conversion to base type is safe.  
Vehicle V = C1;  
  
// Explicit conversion is required to cast back to derived type. The code below will compile but throw an exception at run time if the right-side object is not a Car object.  
Car C2 = (Car) V;  
  
**What operators can be used to cast from one reference type to another without the risk of throwing an exception?**   
The is and as operators can be used to cast from one reference type to another without the risk of throwing an exception.  
  
**If casting fails what type of exception is thrown?**InvalidCastException

**What is Boxing and Unboxing?**   
**Boxing** - Converting a value type to reference type is called boxing. An example is shown below.  
int i = 101;  
object obj = (object)i; // Boxing  
  
**Unboxing** - Converting a reference type to a value typpe is called unboxing. An example is shown below.  
obj = 101;  
i = (int)obj; // Unboxing  
  
**Is boxing an implicit conversion?**Yes, boxing happens implicitly.  
  
**Is un-boxing an implicit conversion?**   
No, un-boxing is an explicit conversion.  
  
**What happens during the process of boxing?**Boxing is used to store value types in the garbage-collected heap. Boxing is an implicit conversion of a value type to the type object or to any interface type implemented by this value type. Boxing a value type allocates an object instance on the heap and copies the value into the new object. Due to this boxing and un-boxing can have performance impact.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Designing Types**

**Type and member basics**

**Constants and Fields**

**Methods: Constructors, Operators**

**Type and member basics**

**What standard types does C# use?**

C# supports a very similar range of basic types to C++, including int, long, float, double, char, string, arrays, structs and classes. In C# Types The names may be familiar, but many of the details are different. For example, a long is 64 bits in C#, whereas in C++ the size of a long depends on the platform (typically 32 bits on a 32-bit platform, 64 bits on a 64-bit platform). Also classes and structs are almost the same in C++ - this is not true for C#. Finally, chars and strings in .NET are 16-bit (Unicode/UTF-16), not 8-bit like C++.

**What is the syntax to inherit from a class in C#?**

Place a colon and then the name of the base class.

Example: class DerivedClassName: BaseClassName {…}

**What is the difference about Switch statement in C#?**

No fall-throughs allowed. Unlike the C++ switch statement, C# does not support an explicit fall through from one case label to another. If you want, you can use goto a switch-case, or goto default.

case 1:

cost += 25;

break;

case 2:

cost += 25;

goto case 1;

**Assume that a class, Class1, has both instance and static constructors. Given the code below, how many times will the static and instance constructors fire?**

Class1 c1 = new Class1();

Class1 c2 = new Class1();

Class1 c3 = new Class1();

By definition, *a static constructor is fired only once when the class is loaded. An instance constructor on the other hand is fired each time the class is instantiated.* So, in the code given above, the static constructor will fire once and the instance constructor will fire three times.

**Explain when to use static class and static members.**

Static classes and class members are used to create data and functions that can be accessed without creating an instance of the class. Static class members can be used to separate data and behavior that is independent of any object identity: the data and functions do not change regardless of what happens to the object. Static members are often used to represent data or calculations that do not change in response to object state; for instance, a math library might contain static methods for calculating sine and cosine.

Static classes can be used when there is no data or behavior in the class that depends on object identity. i.e. use a static class to contain methods that are not associated with a particular object. For example, it is a common requirement to create a set of methods that do not act on instance data and are not associated to a specific object in your code. You could use a static class to hold those methods.

**What are the main features of a static class?**

The main features of a static class are:

* They only contain static members.
* They cannot be instantiated.
* They are sealed.
* They cannot contain Instance Constructors.

It is not possible to create instances of a static class using the [new](http://msdn.microsoft.com/en-us/library/51y09td4(VS.80).aspx) keyword. Static classes are loaded automatically by the .NET Framework common language runtime (CLR) when the program or namespace containing the class is loaded.

**When static members are initialized?**

Static members are initialized before the static member is accessed for the first time, and before the static constructor, if any is called.

**Is it that creating a static class is same as creating a class that contains only static members and a private constructor?**

Yes. A private constructor prevents the class from being instantiated.

**Can static class be inherited?**

Static classes are sealed and therefore cannot be inherited.

**Can static classes have constructor?**

Static classes cannot contain a constructor, although it is still possible to declare a static constructor to assign initial values or set up some static state.

**In which cases you use override and new base?**

Use the new modifier to explicitly hide a member inherited from a base class. To hide an inherited member, declare it in the derived class using the same name, and modify it with the new modifier.

**You have one base class virtual function how will you call the function from derived class?**

class a

{

public virtual int m()

{

return 1;

}

}

class b:a

{

public int j()

{

return m();

}

}

**Can we call a base class method without creating instance?**

* It is possible if it’s a static method.
* It is possible by inheriting from that class also.It is possible from derived classes using base keyword.

**How make sure C# classes will interoperate with other .NET languages?**

Ensure that the C# application code conforms to the [Common Language Specification (CLS)](http://specification). To help achieve compliance, add the [assembly:CLSCompliant(true)] global attribute to all C# source files. This attribute will cause the C# compiler to throw an error if a non-CLS-compliant feature is used.

**What are Access Modifiers in C#?**   
In C# there are 5 different types of Access Modifiers.  
*Public* The public type or member can be accessed by any other code in the same assembly or another assembly that references it.  
*Private*The type or member can only be accessed by code in the same class or struct.  
*Protected*The type or member can only be accessed by code in the same class or struct, or in a derived class.  
*Internal*The type or member can be accessed by any code in the same assembly, but not from another assembly.  
*Protected Internal*  
The type or member can be accessed by any derived class in same assembly.  
  
**What are Access Modifiers used for?**Access Modifiers are used to control the accessibilty of types and members with in the types.  
  
**Can you use all access modifiers for all types?**   
No, Not all access modifiers can be used by all types or members in all contexts, and in some cases the accessibility of a type member is constrained by the accessibility of its containing type.  
  
**Can derived classes have greater accessibility than their base types?**No, Derived classes cannot have greater accessibility than their base types. For example the following code is illegal.  
using System;  
internal class InternalBaseClass  
{  
   public void Print()  
   {  
      Console.WriteLine("I am a Base Class Method");  
   }  
}  
public class PublicDerivedClass : InternalBaseClass  
{  
   public static void Main()  
   {  
      Console.WriteLine("I am a Public Derived Class Method");  
   }  
}  
  
When you compile the above code an error will be generated stating "Inconsistent accessibility: base class InternalBaseClass is less accessible than class PublicDerivedClass".To make this simple, you cannot have a public class B that derives from an internal class A. If this were allowed, it would have the effect of making A public, because all protected or internal members of A are accessible from the derived class.  
  
**Is the following code legal?**   
using System;  
private class Test  
{  
   public static void Main()  
   {  
   }  
}  
  
No, a compile time error will be generated stating *"Namespace elements cannot be explicitly declared as private, protected, or protected internal"*  
  
**Can you declare struct members as protected?**   
No, struct members cannot be declared protected. This is because structs do not support inheritance.  
  
**Can the accessibility of a type member be greater than the accessibility of its containing type?**No, the accessibility of a type member can never be greater than the accessibility of its containing type. For example, a public method declared in an internal class has only internal accessibility.  
  
**What does protected internal access modifier mean?**The protected internal access means protected OR internal, not protected AND internal. In simple terms, a protected internal member is accessible from any class in the same assembly, including derived classes. To limit accessibility to only derived classes in the same assembly, declare the class itself internal, and declare its members as protected.  
  
**What is the default access modifier for a class, struct and an interface declared directly with a namespace?**   
internal  
  
**Will the following code compile?**using System;  
interface IExampleInterface  
{  
   public void Save();  
}  
  
No, you cannot specify access modifer for an interface member. Interface members are always public.  
  
**Can you specify an access modifier for an enumeration?**   
Enumeration members are always public, and no access modifiers can be specified.

**Describe the accessibility modifier “protected internal”.**

It is available to classes that are within the same assembly and derived from the specified base class.

**Structs are largely redundant in C++.Why does C# have them?**

In C++, a struct and a class are pretty much the same thing. The only difference is the default visibility level (public for structs, private for classes). However, in C# structs and classes are very different. In C#, structs are value types (instances stored directly on the stack, or inline within heap-based objects), whereas classes are reference types (instances stored on the heap, accessed indirectly via a reference). Also structs cannot inherit from structs or classes, though they can implement interfaces. Structs cannot have destructors. A C# struct is much more like a C struct than a C++ struct.

**Who is a protected class-level variable available to?**It is available to any sub-class (a class inheriting this class).

**Are private class-level variables inherited?**Yes, but they are not accessible.  Although they are not visible or accessible via the class interface, they are inherited. 

**Describe the accessibility modifier “protected internal”.**It is available to classes that are within the same assembly and derived from the specified base class.

**How many ways to use "New" Keyword in C#?**

In C#, the **new** keyword can be used as an operator, a modifier, or a constraint.

* [new Operator](http://msdn.microsoft.com/en-us/library/fa0ab757(v=vs.80).aspx): Used to create objects and invoke constructors.
* [new Modifier](http://msdn.microsoft.com/en-us/library/435f1dw2(v=vs.80).aspx): Used to hide an inherited member from a base class member.
* [new Constraint](http://msdn.microsoft.com/en-us/library/sd2w2ew5(v=vs.80).aspx): Used to restrict types that might be used as arguments for a type parameter in a generic declaration.

**Constants**

**What are constants?***Constants in C# are immutable values which are known at compile time and do not change for the life of the program.* Constants are declared using the const keyword. Constants must be initialized as they are declared. You cannot assign a value to a constant after it is declared. An example is shown below.  
  
using System;  
class Circle  
{  
   public const double PI = 3.14;  
   public Circle()  
   {  
      //Error : You can only assign a value to a constant field at the time of declaration  
      //PI = 3.15;  
   }  
}  
class MainClass  
{  
   public static void Main()  
   {  
      Console.WriteLine(Circle.PI);  
   }  
}

**Can you declare a class or a struct as constant?**   
*No, User-defined types including classes, structs, and arrays, cannot be const. Only the C# built-in types excluding System.Object may be declared as const. Use the readonly modifier to create a class, struct, or array that is initialized one time at runtime (for example in a constructor) and thereafter cannot be changed.***Does C# support const methods, properties, or events?**No, C# does not support const methods, properties, or events.  
  
**Can you change the value of a constant filed after its declaration?**   
No, you cannot change the value of a constant filed after its declaration. In the example below, the constant field PI is always 3.14, and it cannot be changed even by the class itself. In fact, when the [compiler](http://undefined) encounters a constant identifier in C# source code (for example, PI), it substitutes the literal value directly into the intermediate language (IL) code that it produces. Because there is no variable address associated with a constant at run time, const fields cannot be passed by reference.  
  
using System;  
class Circle  
{  
   public const double PI = 3.14;  
}  
  
**How do you access a constant field declared in a class?**   
Constants are accessed as if they were static fields because the value of the constant is the same for all instances of the type. You do not use the static keyword to declare them. Expressions that are not in the class that defines the constant must use the class name, a period, and the name of the constant to access the constant. In the example below constant field PI can be accessed in the Main method using the class name and not the instance of the class. Trying to access a constant field using a class instance will generate a compile time error.  
  
using System;  
class Circle  
{  
   public const double PI = 3.14;  
}  
class MainClass  
{  
   public static void Main()  
   {  
      Console.WriteLine(Circle.PI);  
      Circle C = new Circle();  
      // Error : PI cannot be accessed using an instance  
      // Console.WriteLine(C.PI);  
   }  
}

**Why should you override the ToString() method?**   
All types in .Net inherit from system.object directly or indirectly. Because of this inheritance, every type in .Net inherit the ToString() method from System.Object class. Consider the example below.  
  
using System;  
public class MainClass  
{  
  public static void Main()  
  {  
   int Number = 10;  
   Console.WriteLine(Number.ToString());  
  }  
}  
  
In the above example Number.ToString() method will correctly give the string representation of int 10, when you call the ToString() method.  
  
If you have a Customer class as shown in the below example and when you call the ToString() method the output does not make any sense. Hence you have to override the ToString() method, that is inherited from the System.Object class.  
  
using System;  
public class Customer  
{  
 public string FirstName;  
 public string LastName;  
}  
public class MainClass  
{  
 public static void Main()  
 {  
  Customer C = new Customer();  
  C.FirstName = "David";  
  C.LastName = "Boon";  
  Console.WriteLine(C.ToString());  
 }  
}  
  
The code sample below shows how to override the ToString() method in a class, that would give the output you want.  
  
  
using System;  
public class Customer  
{  
  public string FirstName;  
  public string LastName;  
  
  public override string ToString()  
  {  
    return LastName + ", " + FirstName;  
  }  
}  
public class MainClass  
{  
  public static void Main()  
  {  
    Customer C = new Customer();  
    C.FirstName = "David";  
    C.LastName = "Boon";  
    Console.WriteLine(C.ToString());  
  }  
}  
  
*Conclusion:* If you have a class or a struct, make sure you override the inherited ToString() method.

**Fields**

**What are the 2 broad classifications of fields in C#?**1. Instance fields  
2. Static fields  
  
**What are instance fields in C#?**Instance fields are specific to an instance of a type. If you have a class T, with an instance field F, you can create two objects of type T, and modify the value of F in each object without affecting the value in the other object.  
  
**What is a static field?**   
A static field belongs to the class itself, and is shared among all instances of that class. Changes made from instance A will be visible immediately to instances B and C if they access the field.  
  
**Will the following code compile?**using System;  
class Area  
{  
   public static double PI = 3.14;  
}  
class MainClass  
{  
   public static void Main()  
   {  
      Area a = new Area();  
      Console.WriteLine(a.PI);  
   }  
}  
No, a compile time error will be generated stating *"Static member 'Area.PI' cannot be accessed with an instance reference; qualify it with a type name instead"*. This is because PI is a static field. Static fields can only be accessed using the name of the class and not the instance of the class. The above sample program is rewritten as shown below.  
  
using System;  
class Area  
{  
   public static double PI = 3.14;  
}  
class MainClass  
{  
   public static void Main()  
   {  
      Console.WriteLine(Area.PI);  
   }  
}  
  
**Can you declare a field readonly?**   
Yes, a field can be declared readonly. *A read-only field can only be assigned a value during initialization or in a constructor.* An example is shown below.  
  
using System;  
class Area  
{  
   public readonly double PI = 3.14;  
}  
class MainClass  
{  
   public static void Main()  
   {  
      Area a = new Area();  
      Console.WriteLine(a.PI);  
   }  
}  
  
**Will the following code compile?**   
  
using System;  
class Area  
{  
   public readonly double PI = 3.14;  
}  
class MainClass  
{  
   public static void Main()  
   {  
      Area a = new Area();  
      a.PI = 3.15;  
      Console.WriteLine(a.PI);  
   }  
}  
  
No, PI is readonly. You can only read the value of PI in the Main() method. You cannot assign any value to PI.  
  
**What is wrong with the sample program below?**   
using System;  
class Area  
{  
   public const double PI = 3.14;  
   static Area()  
   {  
      Area.PI = 3.15;  
   }  
}

class MainClass  
{  
   public static void Main()  
   {  
      Console.WriteLine(Area.PI);  
   }  
}  
You cannot assign a value to the constant PI field.  
  
**What is the difference between a constant and a static readonly field?**   
A static readonly field is very similar to a constant, except that the C# compiler does not have access to the value of a static read-only field at compile time, only at run time.

The difference is that static read-only can be modified by the containing class, but const can never be modified and must be initialized to a compile time constant. To expand on the static read-only case a bit, the containing class can only modify it:

-- in the variable declaration (through a variable initializer).

-- in the static constructor (instance constructors if it's not static).

**What’s the difference between const and readonly?**

Readonly fields are delayed initalized constants. However they have one more thing different is that; When we declare a field as const it is treated as a static field. whereas the readonly fields are treated as normal class variables. const keyword used ,when u want's value constant at compile time but in case of readonly ,value constant at run timeForm the use point of view if we want a field that can have differnet values between differnet objects of same class, however the value of the field should not change for the life span of object; We should choose the Read Only fields rather than constants.Since the constants have the same value accross all the objects of the same class; they are treated as static.

**Method, Constructor and Destructor, Operators**

**How do I declare a pure virtual function in C#?**

Use the abstract modifier on the method. The class must also be marked as abstract (naturally). Note that abstract methods cannot have an implementation (unlike pure virtual C++ methods).

**What are the different ways a method can be overloaded?**

Different parameter data types, different number of parameters, different order of parameters.

**How do you mark a method obsolete?**

[Obsolete]

public int Foo()

{…}

or

[Obsolete(\”This is a message describing why this method is obsolete\”)]

public int Foo()

{…}

**Why can’t you specify the accessibility modifier for methods inside the interface?**

They all must be public, and are therefore public by default.

**What is a virtual method?**

In C#, virtual keyword can be used to mark a property or method to make it overrideable. Such methods/properties are called virtual methods/properties.By default, methods and properties in C# are non-virtual.

**Is it possible to Override Private Virtual methods?**

No, First of all you cannot declare a method as ‘private virtual’.

**How to inherit the class, but not the method inside it in C#? What is a sealed method in C#? Can a method in C# be sealed? How to create a sealed method?**

If a method is not to be inherited, but the class is, then the method is sealed. It becomes a sealed method of a class. It is important to note here that in C#, a method may not be implicitly declared as sealed. This means that a method cannot be sealed directly. A method in C# can be sealed only when the method is an overriden method. *Once the overriden method is declared as sealed, it will not be further overriding of this method.* See code sample below, where an overriden method is sealed.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace sealed\_method

{

class Program

{

public class BaseClass

{

public virtual void Display()

{

Console.WriteLine("Virtual method");

}

}

public class DerivedClass : BaseClass

{

// Now the display method have been sealed and can;t be overridden

public override sealed void Display()

{

Console.WriteLine("Sealed method");

}

//sealed method is overloaded

public void Display(int a)

{

}

}

//public class ThirdClass : DerivedClass

//{

// public override void Display()

// {

// Console.WriteLine("Here we try again to override display method which is not possible and will give error");

// }

//}

static void Main(string[] args)

{

DerivedClass ob1 = new DerivedClass();

ob1.Display(); //Sealed method

BaseClass ob2 = new DerivedClass();

ob2.Display(); //Sealed method

BaseClass ob3 = new BaseClass();

ob3.Display(); //Virtual method

Console.ReadLine();

}

}

}

**Can a sealed method be overloaded?**

Yes. See above example.

**Can I call a virtual method from a constructor/destructor?**

Yes, but it’s generally not a good idea. The mechanics of object construction in .NET are quite different from C++, and this affects virtual method calls in constructors.C++ constructs objects from base to derived, so when the base constructor is executing the object is effectively a base object, and virtual method calls are routed to the base class implementation. By contrast, in .NET the derived constructor is executed first, which means the object is always a derived object and virtual method calls are always routed to the derived implementation. (Note that the C# compiler inserts a call to the base class constructor at the start of the derived constructor, thus preserving standard OO semantics by creating the illusion that the base constructor is executed first.)The same issue arises when calling virtual methods from C# destructors. A virtual method call in a base destructor will be routed to the derived implementation.

**Does C# support variable argument on method?**

The params keyword can be applied on a method parameter that is an array. When the method is invoked, the elements of the array can be supplied as a comma separated list. So, if the method parameter is an object array,

void paramsExample(object arg1, object arg2, params object[] argsRest)

{

foreach (object arg in argsRest)

{

/\* .... \*/

}

}

then the method can be invoked with any number of arguments of any type.paramsExample(1, 0.0f, "a string", 0.0m, new UserDefinedType());

**Are all methods virtual in C#?**

No. Like C++, methods are non-virtual by default, but can be marked as virtual.

**Is it possible to restrict the scope of a field/method of a class to the classes in the same namespace?**

There is no way to restrict to a namespace. Namespaces are never units of protection. But if you’re using assemblies, you can use the ‘internal’ access modifier to restrict access to only within the assembly.

**Is it possible to have different access modifiers on the get/set methods of a property?**

No. The access modifier on a property applies to both its get and set accessors. What you need to do if you want them to be different is make the property read-only (by only providing a get accessor) and create a private/internal set method that is separate from the property.

**What’s the implicit name of the parameter that gets passed into the set method/property of a class?**Value.  The data type of the value parameter is defined by whatever data type the property is declared as.

**Can we pass method parameters either by value or by reference in C#?**

Yes. By default, value type or reference type parameters are passed by value. To pass value type or reference type parameter by reference, use the **ref** or **out** keyword.

**What is the use of the out and the ref parameters?**

The **out** and the **ref** parameters are used to return values in the same variables, that you pass an an argument of a method. These both parameters are very useful when your method needs to return more than one values. Passing parameters by reference allows function members, methods, properties, indexers, operators, and constructors, to change the value of the parameters and have that change persist.

**Explain meaning of passing value types and reference types variables by value.**

A [value-type](http://msdn.microsoft.com/en-us/library/s1ax56ch(VS.80).aspx) variable contains its data directly as opposed to a [reference-type](http://msdn.microsoft.com/en-us/library/490f96s2(VS.80).aspx) variable, which contains a reference to its data. Therefore, passing a value-type variable to a method means passing a copy of the variable to the method. Any changes to the parameter that take place inside the method have no affect on the original data stored in the variable.

Example: Passing Value Types by Value

class PassingValByVal

{

static void SquareIt(int x)

// The parameter x is passed by value.

// Changes to x will not affect the original value of x.

{

x \*= x;

System.Console.WriteLine("The value inside the method: {0}", x); //25

}

static void Main()

{

int n = 5;

System.Console.WriteLine("The value before calling the method: {0}", n); //5

SquareIt(n); // Passing the variable by value.

System.Console.WriteLine("The value after calling the method: {0}", n); //5

}

}

A variable of a [reference type](http://msdn.microsoft.com/en-us/library/490f96s2(VS.80).aspx) does not contain its data directly; it contains a reference to its data. When you pass a reference-type parameter by value, it is possible to change the data pointed to by the reference, such as the value of a class member. However, you cannot change the value of the reference itself; that is, you cannot use the same reference to allocate memory for a new class and have it persist outside the block.

Example: Passing Reference Types by Value

class PassingRefByVal

{

static void Change(int[] pArray)

{

pArray[0] = 888; // This change affects the original element.

pArray = new int[5] {-3, -1, -2, -3, -4}; // This change is local.

System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]); //-3

}

static void Main()

{

int[] arr = {1, 4, 5};

System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}", arr [0]); //1

Change(arr);

System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", arr [0]); //888

}

}

**Explain meaning of passing value types and reference types variables by reference.**

If you want the called method to change the value of the value type parameter, you have to pass it by reference, using the [ref](http://msdn.microsoft.com/en-us/library/14akc2c7(VS.80).aspx) or [out](http://msdn.microsoft.com/en-us/library/t3c3bfhx(VS.80).aspx) keyword.

Example: Passing Value Types by Reference

class PassingValByRef

{

static void SquareIt(ref int x)

// The parameter x is passed by reference.

// Changes to x will affect the original value of x.

{

x \*= x;

System.Console.WriteLine("The value inside the method: {0}", x); //5

}

static void Main()

{

int n = 5;

System.Console.WriteLine("The value before calling the method: {0}", n); //5

SquareIt(ref n); // Passing the variable by reference.

System.Console.WriteLine("The value after calling the method: {0}", n); //25

}

}

If you pass a reference type parameter using the [ref](http://msdn.microsoft.com/en-us/library/14akc2c7(VS.80).aspx) or [out](http://msdn.microsoft.com/en-us/library/t3c3bfhx(VS.80).aspx) keyword, you can change the value of the reference itself; that is, you can use the same reference to allocate memory for a new class and have it persist outside the block.

Example: Passing Reference Types by Reference

class PassingRefByRef

{

static void Change(ref int[] pArray)

{

// Both of the following changes will affect the original variables:

pArray[0] = 888;

pArray = new int[5] {-3, -1, -2, -3, -4};

System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]); //-3

}

static void Main()

{

int[] arr = {1, 4, 5};

System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}", arr[0]); //1

Change(ref arr);

System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", arr[0]); //-3

}

}

**What is the difference between out and ref?**

[ref] and [out] both allow the called method to modify a parameter. The difference between them is what happens *before* you make the call.

* [ref] means that the parameter has a value on it before going into the function. The called function can read and or change the value any time. *The parameter goes in, then comes out*
* [out] means that the parameter has no official value before going into the function. The called function must initialize it before reading it. *The parameter only goes out*

**Why often properties are preferred to methods?**

Benefits of properties.  
1) Properties can be used easier through reflection to perform automated handling. For example, the property dialog editor is based on the concept of reflection against properties, not fields. You could technically do reflection against fields as well but the standard is to use properties.  
  
2) Properties allow you to perform special processing when a value is set (or gotten). You can't do this with fields because there is no "hook" to put code in. For example, here is some C# code that will make sure a field is never null. You can't do this with fields.  
  
public string FirstName  
{  
    set   
    {  
        if (value == null) value = "";  
        this.firstName = value;  
    }  
}  
  
3) Similar #3 above, properties give you a place to hook in events. I could easily add a method to invoke a FirstNameChanged event in the above setter. You can't do this with fields. This is probably the most important difference between using fields and properties.

**Explain about Main method as entry point of an application.**

The behavior of the object is broken down into various methods within the class. In any c# application, there can have only one Main() method:  
  
public static void Main( )  
{  
..  
..  
}  
  
The definition of the *Main* method starts with its access modifier i.e. *public*, hence any object can call this method. Its followed by the keyword **static,** this keyword makes turns the method into a *class level* member. Hence you do not have to first create an object of the class to call the *Main* method. I understand things might seem a bit cloudy now, but they will get clarified as we move on.   
Then there is the return type, after every method finishes execution it returns the control back to the caller. When a method returns, it has to have a return value. In this case since this method is going to return no value the **void** keyword is used to denote that the method returns no value. Finally, there is the method name, Main with a set of empty parentheses. Like the classes, methods too use *scope operators* i.e. { } to define the scope of the method. All the code of the method has to be packaged within the scope operators for the method.  
  
There is a special significance attached with the method Main in C#. When you click on an application to run it, the runtime should know from where (which class, which method) it should start executing the code. In C#, like many other programming languages there is the Main method which the runtime uses by default to start running the application. Hence the Main method is also known as the entry point of the application.   
  
**Note:** *C / C++ / Java programmers its called the Main method with a capital M, unlike other languages.*  
**Note:** *Unlike C++ Main method is not a global function, it has to be defined as a class member.*

**In Java, the Main method cannot have any other access modifier than 'public'. How is it that in C# you can have a private access modifier to the Main method? Can anybody throw any light, comparing JVM with CLR?**

If you refer to Main method as the entry point of the application, then you have a similar situation in C# than in Java, in c# the entry point MUST be declared as static void Main() , if that is not present the compiler will give you an error.  
  
Now there is nothing that prohibits you to name a method Main () on any class, only that it will no be the entry point of the application.

But it’s not true that "in c# the entry point MUST be declared as static void Main (), if that is not present the compiler will give you an error." Infact it works even if you declare the entry point as "private static void Main ()”, the compiler gives no error, neither the CLR throws any run-time error!!! Isn’t this strange??!! That’s precisely the point of confusion. You may try this out.

Public, private, protected and others are access attributes that are enforced primarily by the compiler (at runtime only when dynamic invocation is involved). Beyond that, the runtime is free to accept or ignore the access specifiers - and in the case of Main, it kindly ignores them.  
  
Matters such as these (and this is but one example of many others) are implementation specific details for the runtime, not for a particular language.

**Why do we need static keyword for Main method in c#?**

When a program is launched, no instances of any class are present in memory. As the Main method is static, it can be called without creating an object and can then assume control of the program. It is the Main method's task to create the objects that the program requires to function correctly.  
The static modifier can be used with classes, fields, methods, properties, operators, events and constructors, but cannot be used with indexers, destructors, or types other than classes.

**What are valid signatures for the Main function?**

* public static void Main()
* public static int Main()
* public static void Main( string[] args )
* public static int Main(string[] args )

**Does Main () always have to be public?**

No.

**What does the keyword “virtual” declare for a method or property?**The method or property can be overridden. 

**How is method overriding different from method overloading?**When overriding a method, you change the behavior of the method for the derived class.  Overloading a method simply involves having another method with the same name within the class.

**Can you modify the access modifiers while overriding a method in a class?**

No. An **override** declaration cannot change the accessibility of the **virtual** method. Both the **override** method and the **virtual** method must have the same [access level modifier](http://msdn.microsoft.com/en-us/library/wxh6fsc7(VS.80).aspx).

An **override** method provides a new implementation of a member inherited from a base class. The method overridden by an **override** declaration is known as the overridden base method. The overridden base method must have the same signature as the **override** method. You cannot override a non-virtual or static method. The overridden base method must be **virtual**, **abstract**, or **override**.

**Can you declare an override method to be static if the original method is not static?**No. The signature of the virtual method must remain the same.  (Note: Only the keyword virtual is changed to keyword override). 

**What are the different ways a method can be overloaded in C#?**Different parameter data types, different number of parameters, different order of parameters. 

**Can destructors have access modifiers?**   
No, destructors cannot have access modifiers.

**What is the difference between type/static constructor and instance constructor? What is static constructor, when it will be fired? And what is its use?**(Class constructor method is also known as type constructor or type initializer)  
Instance constructor is executed when a new instance of type is created and the class constructor is executed after the type is loaded and before any one of the type members is accessed. (It will get executed only 1st time, when we call any static methods/fields in the same class.) Class constructors are used for static field initialization. Only one class constructor per type is permitted, and it cannot use the vararg (variable argument) calling convention.  
A static constructor is used to initialize a class. It is called automatically to initialize the class before the first instance is created or any static members are referenced.

**What is Private Constructor? What is its use? Can you create instance of a class which has Private Constructor?**When a class declares only private instance constructors, it is not possible for classes outside the program to derive from the class or to directly create instances of it. (Except Nested classes)  
Make a constructor private if:  
- You want it to be available only to the class itself. For example, you might have a special constructor used only in the implementation of your class' Clone method.  
- You do not want instances of your component to be created. For example, you may have a class containing nothing but Shared utility functions, and no instance data. Creating instances of the class would waste memory.

**Can abstract classes have constructors? If the answer is yes, when do they get called?**

Yes, we can have constructor in abstract class. But we can not make instance of the abstract class. Instead, we can make a reference to that abstract class and when we make a new object of the class which extends the abstract class, the constructor of abstract class get called.

See the code for example:

public abstract class TestAbstract

{

public TestAbstract()

{

Console.WriteLine("...in abstract class' constructor");

}

public abstract void ShowAbstract();

public void Show()

{

Console.WriteLine("...in Show");

}

}

public class Test : TestAbstract

{

public static void Main(String[] args)

{

TestAbstract ta = new Test(); //constructor call

ta.ShowAbstract();

ta.Show();

Console.ReadLine();

}

public override void ShowAbstract()

{

Console.WriteLine("...in ShowAbstract");

}

}

Output

...in abstract class' constructor

...in ShowAbstract

...in Show

**If a base class has a number of overloaded constructors and an inheriting class has a number of overloaded constructors; can you enforce a call from an inherited constructor to specific base constructor?**

Yes, just place a colon, and then keyword base (parameter list to invoke the appropriate constructor) in the overloaded constructor definition inside the inherited class.

**[Is the destructor called if the constructor throws an exception?](http://stackoverflow.com/questions/188693/is-the-destructor-called-if-the-constructor-throws-an-exception)**

It does for C# (see code below) but not for C++.

using System;

class Test

{

Test()

{

throw new Exception();

}

~Test()

{

Console.WriteLine("Finalized");

}

static void Main()

{

try

{

new Test();

}

catch {}

GC.Collect();

GC.WaitForPendingFinalizers();

}

}

This prints "Finalized

**What is the order in which the destructors and the constructors are called in C++ and C#? Use the examples of some Base classes and Derived Classes.**

**The order is:**

Base constructor

Derived constructor

Derived destructor

Base destructor

**How call a virtual method from a constructor or destructor?**

In C++, objects are constructed from base class to derived class. This means that when the base class constructor is running the object is effectively a base object. So, C++ virtual method calls are directed to the base class implementation.

In .NET on the other hand, the derived constructor is executed first. This means the object will be a derived object; and, virtual method calls are directed to the derived implementation.

The C# compiler inserts a call to the base class constructor at the beginning of any derived constructor in order to maintain OO semantics, i.e. that the base class constructor is called first.

In a similar fashion, when C# destructors call virtual methods, virtual method calls from a base destructor are directed to the derived implementation.

Therefore, in C#, a virtual method can be called from a constructor or destructor. But, usually, it is a bad idea. .NET object construction is very different from C++ object construction; and, virtual method calling is affected.

Why there is no virtual constructor and destructor in C#?

**Should I make my destructor virtual?**

A C# destructor is really just an override of the System.Object Finalize method, and so is virtual by definition.

**What is the order of destructors called in a polymorphism hierarchy?**

Destructors are called in reverse order of constructors. First destructor of most derived class is called followed by its parent’s destructor and so on till the topmost class in the hierarchy.

You don’t have control over when the first destructor will be called, since it is determined by the garbage collector. Sometime after the object goes out of scope GC calls the destructor, then its parent’s destructor and so on.

When a program terminates definitely all object’s destructors are called.

**Are C# destructors the same as C++ destructors?**

No. They look the same but they are very different. The C# destructor syntax (with the familiar ~ character) is just syntactic sugar for an override of the System.Object Finalize method. This Finalize method is called by the garbage collector when it determines that an object is no longer referenced, before it frees the memory associated with the object. So far this sounds like a C++ destructor. The difference is that the garbage collector makes no guarantees about when this procedure happens. Indeed, the algorithm employed by the CLR garbage collector means that it may be a long time after the application has finished with the object. This lack of certainty is often termed ‘non-deterministic finalization’, and it means that C# destructors are not suitable for releasing scarce resources such as database connections, file handles, etc. To achieve deterministic destruction, a class must offer a method to be used for the purpose. The standard approach is for the class to implement the IDisposable interface. The user of the object must call the Dispose() method when it has finished with the object. C# offers the ‘using’ construct to make this easier.

**Are C# constructors the same as C++ constructors?**

Very similar, but there are some significant differences. First, C# supports constructor chaining. This means one constructor can call another:

class Person

{

public Person( string name, int age ) { … }

public Person( string name ) : this( name, 0 ) {}

public Person() : this( “”, 0 ) {}

}

Another difference is that virtual method calls within a constructor are routed to the most derived implementation error handling is also somewhat different. If an exception occurs during construction of a C# object, the destructor (finalizer) will still be called. This is unlike C++ where the destructor is not called if construction is not completed. Finally, C# has static constructors. The static constructor for a class runs before the first instance of the class is created. Also note that (like C++) some C# developers prefer the factory method pattern over constructors.

**What are the differences between C# and Java constructors?**

The constructor semantics and syntax for C# are identical to those of the Java language.

The following C# and Java examples are equivalent:

using System;

public class CSharpClass

{

CSharpClass ()

{

}

}

public class JavaClass

{

JavaClass()

{

}

}

**What are the differences between C# finalizers and Java destructors?**

C# features hybrid destructors which use a C++ style, destructor syntax yet share most of their semantics with Java finalizers.

The following C# and Java examples are equivalent:

using System;

public class CSharpClass

{

~CSharpClass()

{

}

}

public class JavaClass

{

public void finalize()

{

}

}

Although C# destructors automatically call the base class destructor after execution, this does not happen in the Java finalizer system.

**How to make a C# destructor virtual?**

By definition, a C# destructor *is* virtual. Because, a C# destructor is basically an override of the Finalize method of System.Object. Therefore, these is no need to make a C# destructor virtual.

**Are C# constructors inherited?**

No. C# constructors cannot be inherited.

If constructor inheritance were allowed, then necessary initialization in a base class constructor might easily be omitted. This could cause serious problems which would be difficult to track down. For example, if a new version of a base class appears with a new constructor, your class would get a new constructor automatically. This could be catastrophic.

**Why must struct constructors have at least one argument?**

The .NET [Common Language Runtime (CLR)](http://runtime) does not guarantee that parameterless constructors will be called. If structs were permitted to have default, parameterless constructors, the implication would be that default constructors would always be called. Yet, the CLR makes no such guarantee.

For instance, an array of value types will be initialized to the initial values of its members—i.e., zero for number type primitive members, null for reference types, and so forth—and not to the values provided in a default constructor. This feature makes structs perform better; because, constructor code need not be called.

So, requiring that a constructor contain a minimum of one parameter reduces the possibility that a constructor will be defined which is expected to be called every time the struct type is built.

**What is the syntax for calling an overloaded constructor from within a constructor?**

You may have noticed that this() and *constructor-name*() do not compile.

The syntax for calling another constructor is as follows:

class A

{

A (int i) { }

}

class B : A

{

B() : base (10) *// call base constructor A(10)*

{ }

B(int i) : this() *// call B()*

{ }

public static void Main() {}

}

**How do destructors work in C#?**

A destructor is a method that is called when an object is destroyed—it's memory is marked unused. C++ destructors are used to free up memory and other resources and to perform housekeeping tasks. In .NET, the Garbage Collector (GC) performs much of this type of work automatically. Generally, rather than define a destructor for manual cleanup, let the Common Language Runtime (CLR) handle it.

In C#, the Finalize method performs the operations that a standard C++ destructor might perform. Finalizers are similar to destructors except that it is not guranteed that they will be called by the CLR.

Here is a sample finalizer specification using the C++ destructor syntax which places a tilde (~) symbol before the class name:

class Test

{

~Test()

{

...

}

public static void Main() {}

}

When defining a C# finalizer, it is not named Finalize. However, finalizers do override object.Finalize(), which is called during the garbage collection process.

**Can you declare a C++ type destructor in C# like ~MyClass()?**

Yes, but what’s the point, since it will call Finalize(), and Finalize() has no guarantees when the memory will be cleaned up, plus, it introduces additional load on the garbage collector. The only time the finalizer should be implemented, is when you’re dealing with unmanaged code.

**Properties**

**What are Properties in C#? Explain with an example.**   
Properties in C# are class members that provide a flexible mechanism to read, write, or compute the values of private fields. Properties can be used as if they are public data members, but they are actually special methods called accessors. This enables data to be accessed easily and still helps promote the safety and flexibility of methods.  
  
In the example below \_firstName and \_lastName are private string variables which are accessible only inside the Customer class. \_firstName and \_lastName are exposed using FirstName and LastName public properties respectively. The get property accessor is used to return the property value, and a set accessor is used to assign a new value. These accessors can have different access levels. The value keyword is used to define the value being assigned by the set accessor. The FullName property computes the full name of the customer. Full Name property is readonly, because it has only the get accessor. Properties that do not implement a set accessor are read only.  
  
The code block for the get accessor is executed when the property is read and the code block for the set accessor is executed when the property is assigned a new value.  
  
using System;  
class Customer  
{  
   // Private fileds not accessible outside the class.  
   private string \_firstName = string.Empty;  
   private string \_lastName = string.Empty;  
   private string \_coutry = string.Empty;  
  
   // public FirstName property exposes \_firstName variable  
   public string FirstName  
   {  
      get  
      {  
         return \_firstName;  
      }  
      set  
      {  
         \_firstName = value;  
      }  
   }  
   // public LastName property exposes \_lastName variable  
   public string LastName  
   {  
      get  
      {  
         return \_lastName;  
      }  
      set  
      {  
         \_lastName = value;  
      }  
   }  
   // FullName property is readonly and computes customer full name.  
   public string FullName  
   {  
      get  
      {  
         return \_lastName + ", " + \_firstName;  
      }  
   }  
   //Country Property is Write Only  
   public string Country  
   {  
      set  
      {  
         \_coutry = value;  
      }  
   }  
  
}  
class MainClass  
{  
   public static void Main()  
   {  
      Customer CustomerObject = new Customer();  
      //This line will call the set accessor of FirstName Property  
      CustomerObject.FirstName = "David";  
      //This line will call the set accessor of LastName Property  
      CustomerObject.LastName = "Boon";  
      //This line will call the get accessor of FullName Property  
      Console.WriteLine("Customer Full Name is : " + CustomerObject.FullName);  
   }  
}  
  
**Explain the 3 types of properties in C# with an example?**   
**1.** Read Only Properties: Properties without a set accessor are considered read-only. In the above example FullName is read only property.  
**2.** Write Only Properties: Properties without a get accessor are considered write-only. In the above example Country is write only property.  
**3.** Read Write Properties: Properties with both a get and set accessor are considered read-write properties. In the above example FirstName and LastName are read write properties.  
  
**What are the advantages of properties in C#?**   
**1.** Properties can validate data before allowing a change.  
**2.** Properties can transparently expose data on a class where that data is actually retrieved from some other source such as a database.  
**3.** Properties can take an action when data is changed, such as raising an event or changing the value of other fields.  
  
**What is a static property? Give an example?**A property that is marked with a static keyword is considered as static property. This makes the property available to callers at any time, even if no instance of the class exists. In the example below PI is a static property.  
  
using System;  
class Circle  
{  
   private static double \_pi = 3.14;  
   public static double PI  
   {  
      get  
      {  
         return \_pi;  
      }  
   }  
}  
class MainClass  
{  
   public static void Main()  
   {  
      Console.WriteLine(Circle.PI);  
   }  
}

**What is an indexer in C#?**

The indexers are usually known as smart arrays in C# community. Defining a C# indexer is much like defining properties. We can say that *an indexer is a member that enables an object to be indexed in the same way as an array.*

<modifier> <return type> this [argument list]

{

get

{

// Get codes goes here

}

set

{

// Set codes goes here

}

}

Where the modifier can be private, public, protected or internal. The return type can be any valid C# types. The 'this' is a special keyword in C# to indicate the object of the current class. The formal-argument-list specifies the parameters of the indexer.

**What is a virtual property? Give an example?**   
A property that is marked with virtual keyword is considered virtual property. Virtual properties enable derived classes to override the property behavior by using the override keyword. In the example below FullName is virtual property in the Customer class. BankCustomer class inherits from Customer class and overrides the FullName virtual property. In the output you can see the over riden implementation. A property overriding a virtual property can also be sealed, specifying that for derived classes it is no longer virtual.  
  
  
using System;  
class Customer  
{  
   private string \_firstName = string.Empty;  
   private string \_lastName = string.Empty;  
  
   public string FirstName  
   {  
      get  
      {  
         return \_firstName;  
      }  
      set  
      {  
         \_firstName = value;  
      }  
   }  
   public string LastName  
   {  
      get  
      {  
         return \_lastName;  
      }  
      set  
      {  
         \_lastName = value;  
      }  
   }  
   // FullName is virtual  
   public virtual string FullName  
   {  
      get  
      {  
         return \_lastName + ", " + \_firstName;  
      }  
   }  
}  
class BankCustomer : Customer  
{  
   // Overiding the FullName virtual property derived from customer class  
   public override string FullName  
   {  
      get  
      {  
         return "Mr. " + FirstName + " " + LastName;  
      }  
   }  
}  
class MainClass  
{  
   public static void Main()  
   {  
      BankCustomer BankCustomerObject = new BankCustomer();  
      BankCustomerObject.FirstName = "David";  
      BankCustomerObject.LastName = "Boon";  
      Console.WriteLine("Customer Full Name is : " + BankCustomerObject.FullName);  
   }  
}  
  
**What is an abstract property. Give an example?**   
A property that is marked with abstract keyword is considered abstract property. An abstract property should not have any implementation in the class. The derived classes must write their own implementation. In the example below FullName property is abstract in the Customer class. BankCustomer class overrides the inherited abstract FullName property with its own implementation.  
  
using System;  
abstract class Customer  
{  
   private string \_firstName = string.Empty;  
   private string \_lastName = string.Empty;  
  
   public string FirstName  
   {  
      get  
      {  
         return \_firstName;  
      }  
      set  
      {  
         \_firstName = value;  
      }  
   }  
   public string LastName  
   {  
      get  
      {  
         return \_lastName;  
      }  
      set  
      {  
         \_lastName = value;  
      }  
   }  
   // FullName is abstract  
   public abstract string FullName  
   {  
      get;  
   }  
}  
class BankCustomer : Customer  
{  
   // Overiding the FullName abstract property derived from customer class  
   public override string FullName  
   {  
      get  
      {  
         return "Mr. " + FirstName + " " + LastName;  
      }  
   }  
}  
class MainClass  
{  
   public static void Main()  
   {  
      BankCustomer BankCustomerObject = new BankCustomer();  
      BankCustomerObject.FirstName = "David";  
      BankCustomerObject.LastName = "Boon";  
      Console.WriteLine("Customer Full Name is : " + BankCustomerObject.FullName);  
   }  
}  
  
**Can you use virtual, override or abstract keywords on an accessor of a static property?**   
No, it is a compile time error to use a virtual, abstract or override keywords on an accessor of a static property.

**Essential Types**

**Array**

**Strings**

**Delegates**

**Interface**

**Generics**

**Collections**

**Attributes**

**Reflection**

**Array**

**What is an array?**   
An array is a data structure that contains several variables of the same type.  
  
**What are the 3 different types of arrays?**1. Single-Dimensional  
2. Multidimensional  
3. Jagged  
  
**What is Jagged Array?**   
A jagged array is an array of arrays.

**Example of Single Dimension Array in .Net**

int [] intArray = new int[3];

intArray[2] = 22; // set the third element to 22

**How do you initialize a two-dimensional array that you don’t know the dimensions of?**

* int [, ] myArray; //declaration or int [][] myArray;
* myArray= new int [5, 8]; //actual initialization

**Example of Jagged Array in .Net: Variable Length Array in .Net**

int [][] myTable = new int[3][];

myTable[0] = new int[5];

myTable[1] = new int[2];

myTable[2] = new int[4];

myTable[0][2] = 11;

**Example of String Array in .Net**

string []names = new string[4];

names[2] = “God will make me win”;

**Are arrays value types or reference types?**Arrays are reference types.  
  
**What is the base class for Array types?**   
System.Array  
  
**Can you use foreach iteration on arrays in C#?**Yes, since array type implements **IEnumerable**, you can use **foreach** iteration on all arrays in C#.

**Limitation of Arrays**

The size of an array is always fixed and must be defined at the time of instantiation of an array.

Secondly, an array can only contain objects of the same data type, which we need to define at the time of its instantiation.

**How to declare a two-dimensional array in C#?**

Syntax for Two Dimensional Array in C Sharp is int[,] ArrayName;

**How can you sort the elements of the array in descending order?**

Using Array.Sort() and Array.Reverse() methods.int[] arr = new int[3];

arr[0] = 4;

arr[1] = 1;

arr[2] = 5;

Array.Sort(arr);

Array.Reverse(arr);

**Does C# do array bounds checking?**

Yes. An IndexOutOfRange exception is used to signal an error.

**What is cloning?**

Cloning is the ability to make an exact copy (a clone) of an instance of a type. Cloning may take one of two forms: a shallow copy or a deep copy.

**Why is cloning required?**

*Cloning is really required if setting up the state of an object is expensive and you just need a copy of the object to do some changes to the existing state.* Let's take a good example to understand what I have just said. Consider the DataTable class**.** Building a DataTable can involve operations like querying the database for the schema and data, adding constraints, setting the primary key and so on. So, if you want a new copy of this DataTable, just for making minor changes to the schema or adding a new row etc, it would be more wise just to clone an existing object and work on that, rather than recreating a new DataTable, which can require more time and resources.

Cloning is also widely applicable to Arrays and Collections, where you need a copy of existing elements many a time.

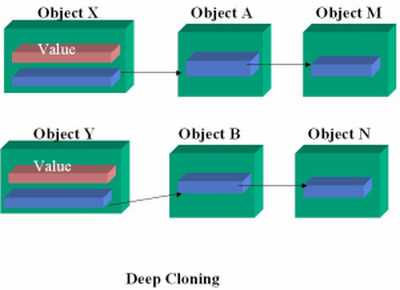
**What are the different types of cloning?**

We can classify two types of cloning based on "how much" is cloned: **Deep** and **Shallow.**

*A shallow clone is a new instance of the same type as the original object, which contains a copy of the value typed fields.* But, if the field is a reference type, the reference is copied, not the referred object. Hence, the reference in the original object and the reference in the clone point to the same object.

*A deep clone of an object, on the other hand, contains a copy of everything directly or indirectly referenced by the object.* Let's take up an example.

X is an object with references to the object A and the object A also has a reference to an object M. A shallow copy of X is an object Y, which also has references to object A. In contrast, a deep copy of X is an object Y with direct reference to object B, and an indirect reference to object N, where B is a copy of A, and N is a copy of M. This is visually depicted below to understand this better.

****

**How to implement cloning?**

System.Objectprovides a protected method MemberwiseClone which can be used to implement Shallow cloning**.** This method is marked as protected, so you can access this method in context of a derived class or within that class itself.

.NET defines an interface called IClonable which has to be implemented by classes that need functionality beyond the scope of shallow cloning (for deep cloning). We need to provide a suitable implementation in the Clone method of the interface to do the same.

There are various ways to implement deep cloning. One method is to serialize the object into a memory stream and deserialize it back into a new object. We would need to use a **Binary** formatter or **SOAP** formatter which do a deep serialization. The problem with this approach is that the class and its members (the entire object graph) have to be marked as serializable, else the formatter would through an exception.

**Reflection** may be another method to achieve the same. One good article written by **Amir Harel** caught my eye. He provides a good clone implementation using this method. The discussion on the article is good too! Here's the link

* <http://www.codeproject.com/csharp/cloneimpl_class.asp>

Understand that, for either of the methods discussed above, the member types must support cloning themselves for deep cloning to be successful. That is, the object graph must be serializable or the individual member must provide an implementation of IClonable. If this is not the case, then we would not be able to deep clone the object at all!

**How to make a shallow copy of an object?**

Shallow copying is relatively easy. It involves copying the object that the Clone method was called on. The reference type fields in the original object are copied over, as are the value-type fields. This means that if the original object contains a field of type StreamWriter, for instance, the cloned object will point to this same instance of the original object's StreamWriter; a new object is not created.  
  
There is no need to deal with static fields when performing a cloning operation. There is only one memory location reserved for each static field per class, per application domain. Besides, the cloned object will have access to the same static fields as the original.  
  
Support for shallow copying is implemented by the MemberwiseClone method of the Object class, which serves as the base class for all .NET classes. So the following code allows a shallow copy to be created and returned by the Clone method:

public object Clone( ) {  
 return (this.MemberwiseClone( ))  
}

**How to make a deep copy of an object (cloning for a user defined class)?**

Making a deep copy is the second way of cloning an object. A deep copy will make a copy of the original object just as the shallow copy does. However, a deep copy will also make separate copies of each reference type field in the original object. Therefore, if the original object contains a StreamWriter type field, the cloned object will also contain a StreamWriter type field, but the cloned object's StreamWriter field will point to a new StreamWriter object, not the original object's StreamWriter object. Support for deep copying is not automatically provided by the Clone  
method or the .NET Framework. Instead, the following code illustrates an easy way of implementing a deep copy:

For deep copy of a user defined class, a class should implement IClonable interface.

Have you ever used the Clone() method of DataSet? This method creates an empty class with same structure as original DataSet.

You can write your own clonable classes. To do so, you must implement IClonable. The following code shows a clonable Test class.

using System.IO;

using System.Runtime.Serialization.Formatters.Binary;

public Class Test : IClonable

{

    public Test()

    {

    }

    // deep copy in separeate memory space

    public object Clone()

    {

     MemoryStream ms = new MemoryStream();

     BinaryFormatter bf = new BinaryFormatter();

     bf.Serialize(ms, this);

     ms.Position = 0;

     object obj = bf.Deserialize(ms);

     ms.Close();

     return obj;

    }

}

Basically, the original object is serialized out to a memory stream using binary serialization, then it is deserialized into a new object, which is returned to the caller. Note that it is important to  
reposition the memory stream pointer back to the start of the stream before calling the Deserialize method; otherwise, an exception indicating that the serialized object contains no data will be thrown.  
Performing a deep copy using object serialization allows the underlying object to be changed without having to modify the code that performs the deep copy. If you performed the deep copy by hand, you'd have to make a new instance of all the instance fields of the original object and copy them over to the cloned object. This is a tedious chore in and of itself. If a change is made to the fields of the object being cloned, the deep copy must also change to reflect this   
modification. Using serialization, you rely on the serializer to dynamically find and serialize all fields contained in the object. If the object is modified, the serializer will still make a deep copy  
without any code modifications.

**When to avoid cloning of object?**

Cloning is a nifty thing to provide as a programmer. But, one should be aware of the need to provide the same and under some circumstances, objects should strictly not should provide this feature. Take for example the SQLTransaction class, which does not support cloning. This class represents a transaction in a SQL Server database. Cloning the object would not make sense, since we possibly cannot comprehend a clone of an active transaction in a database! Therefore, if you think the cloning the state of an object can create inconsistencies in the logic of an application, dont support cloning.

**What’s the difference between the System.Array.CopyTo() and System.Array.Clone()?**

The Clone() method returns a new array (a shallow copy) object containing all the elements in the original array. The CopyTo() method copies the elements into another existing array. Both perform a shallow copy. A shallow copy means the contents (each array element) contains references to the same object as the elements in the original array. A deep copy (which neither of these methods performs) would create a new instance of each element's object, resulting in a different, yet identical object.

**Can you store multiple data types in System.Array?**  
No. 

**How can you sort the elements of the array in descending order?**By calling Sort() and then Reverse() methods. 

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Strings**

**What is a string?**

*A string is basically a sequence of characters.* Each character is a 16 bit Unicode character in the range U+0000 to U+FFFF.

**What is string interning?**

In computer science, **string interning** *is a method of storing only one copy of each distinct string value, which must be immutable.* Interning strings makes some string processing tasks more time- or space-efficient at the cost of requiring more time when the string is created or interned. The distinct values are stored in a **string intern pool**.

String interning is supported by some modern object-oriented programming languages, including Python, Ruby (with its symbols), Java and .NET languages.

The single copy of each string is called its 'intern'.

*The string intern pool is a table that contains a single reference to each unique literal string declared or created programmatically in your application.* The Common Language Runtime (CLR) uses the intern pool to minimize string storage requirements. As a result, an instance of a literal string with a particular value only exists once in the system. For example, if you assign the same literal string to several different variables, at runtime, the CLR retrieves the unique reference to that literal string from the intern pool and assigns it to each variable.

**What are the advantages of string interning?**

* Interned strings speed up string comparisons, which are sometimes a performance bottleneck in applications (such as [compilers](http://compiler) and [dynamic programming language](http://language) runtimes) that rely heavily on hash tables with string keys. Without interning, checking that two different strings are equal involves examining every character of both strings. This is slow for several reasons: it is inherently O(n) in the length of the strings; it typically requires reads from several regions of memory, which take time; and the reads fills up the processor cache, meaning there is less cache available for other needs. With interned strings, a simple object identity test suffices after the original intern operation; this is typically implemented as a pointer equality test, normally just a single machine instruction with no memory reference at all.
* String interning also reduces memory usage if there are many instances of the same string value; for instance, it is read from a [network](http://network) or from [storage](http://storage). Such strings may include magic numbers or network protocol information.

**How string interning happens in CLR?**

Interning happens two ways in the CLR.

1) It happens when you explicitly call System.String.Intern(). Obviously the string returned from this service might be different from the one you pass in, since we might already have an intern’ed instance that has been handed out to the application.

2) It happens automatically, when you load an assembly. All the string literals in the assembly are intern’ed. This is expensive and – in retrospect – may have been a mistake. In the future we might consider allowing individual assemblies to opt-in or opt-out. Note that it is always a mistake to rely on some other assembly to have implicitly intern’ed the strings it gives you. Through versioning, that other assembly might start composing a string rather than using a literal.

The following code example uses three strings that are equal in value to determine whether a newly created string and an interned string are equal.

// Sample for String.Intern(String)

using System;

using System.Text;

class Sample {

public static void Main() {

String s1 = "MyTest";

String s2 = new StringBuilder().Append("My").Append("Test").ToString();

String s3 = String.Intern(s2);

Console.WriteLine("s1 == '{0}'", s1);

Console.WriteLine("s2 == '{0}'", s2);

Console.WriteLine("s3 == '{0}'", s3);

Console.WriteLine("Is s2 the same reference as s1?: {0}", (Object)s2==(Object)s1);

Console.WriteLine("Is s3 the same reference as s1?: {0}", (Object)s3==(Object)s1);

}

}

/\*

This example produces the following results:

s1 == 'MyTest'

s2 == 'MyTest'

s3 == 'MyTest'

Is s2 the same reference as s1?: False

Is s3 the same reference as s1?: True

\*/

**What is difference, when we declare  
String s1="hello";  
String s1=new String("hello"); ?**

**If any other object say, String s2 = new String("hello"); will it create a new object or it will reuse the existing "hello" object?**

If there is already a String literal equal to "hello" present in memory it will be referenced by s1 in the first statement String s1 = "hello"; For the second statement, even if there is a String literal in memory equal to "hello" a new object will be created and used in String s1=new String("hello").

For the first statement, it will create an object in heap and for the second statement, it will create an object in heap as well as create one more object in string pool.  
For the first statement, if we assign String s1="Hai"; then that "Hai" will overridden the "hello" but for the second statement a new object will be created with "Hai" and "hello" also retain in string pool.

No, the '**new**' keyword *always* creates a new object. You can force a new String to reuse a pooled instance:

String s2 = new String("hello").intern();

// or

String s2 = new String("hello");

...

s2.intern();

**A string is created as below:**

**String str = "abc" + "abc" + "bcd";**

**How many string objects are created in string pool?**

4 strings would be created in the string pool. From the end, these are abc, bcd, abcbcd, abcabcbcd.

**How does one compare strings in C#?**

In the past, you had to call .ToString() on the strings when using the == or != operators to compare the strings’ values. That will still work, but the C# compiler now automatically compares the values instead of the references when the == or != operators are used on string types. If you actually do want to compare references, it can be done as follows: if ((object) str1 == (object) str2) { … } Here’s an example showing how string compares work:using System;

public class StringTest

{

public static void Main(string[] args)

{

Object nullObj = null; Object realObj = new StringTest();

int i = 10;

Console.WriteLine(\”Null Object is [\” + nullObj + \”]\n\”

+ \”Real Object is [\” + realObj + \”]\n\”

+ \”i is [\” + i + \”]\n\”);

// Show string equality operators

string str1 = \”foo\”;

string str2 = \”bar\”;

string str3 = \”bar\”;

Console.WriteLine(\”{0} == {1} ? {2}\”, str1, str2, str1 == str2 );

Console.WriteLine(\”{0} == {1} ? {2}\”, str2, str3, str2 == str3 );

}

}Output:Null Object is []

Real Object is [StringTest]

i is [10]

foo == bar ? False

bar == bar ? True

**What does the term immutable mean?**  
The data value may not be changed.  Note: The *variable* value may be changed, but the original immutable data value was discarded and a new data value was created in memory. 

**What is meant by a immutable type?**

Basically, an object is immutable if its state doesn’t change once the object has been created. Consequently, a class is immutable if its instances are immutable.

**What are the benefits of immutable object?**

Immutable object have the following advantages:

* They simplify multithreaded programming.
* They can be used as hashtable keys.
* They simplify state comparison.
* They use memory more efficiently.

**Give examples of immutable types from FCL in .NET.**

**System.String**. When you think that you are modifying a string, you actually create a new string object. Often, we forget about it and we would like to write …

string str = "foofoo";

str.Replace("foo", "FOO");

…where we need to write instead:

str = str.Replace("foo", "FOO");

Of course, doing so comes at the cost of creating multiple string objects in memory when doing some intensive string computation. In this case you need to use the **System.Text.StringBuilder** class that provides a safe way to work with mutable string.

*Another cool thing about string immutability is that even though* ***System.String*** *is a class, string objects get compared with equivalence, as a value type. This is possible because we can consider that the identity of an immutable object is its state.* For example:

string str1 = "foofoo";

string strFoo = "foo";

string str2 = strFoo + strFoo;

// Even thought str1 and str2 reference 2 different objects

// the following assertion is true.

Debug.Assert(str1 == str2);

**Why .NET engineers decided that string should be immutable? Or [Why String class is immutable?](http://hikrish.blogspot.com/2006/07/why-string-class-is-immutable.html)**

There are several reasons for designing string class as immutable. Below are reasons:

* *To achieve memory efficiency:* CLR internally maintains the "String Pool". To achieve the memory efficiency, CLR will refer the String object from pool. It will not create the new String objects. So, whenever you create a new string literal, CLR will check in the pool whether it already exists or not. If already present in the pool, just give the reference to the same object or create the new object in the pool. There will be many references point to the same String objects, if someone changes the value, it will affect all the references.
* *To avoid corrupted keys in collections:* Because programmers will never get a race conditions because of a corrupted string. Also because string are well adapted to be key in hashtables (i.e **Sytem.Collections.generic.Dictionary<K,V>**). Hashtables are almost a magic way to enhance dramatically performance of your code. (*I said magic because under the hood hashtables rely on prime numbers properties and prime numbers are magic!*). The objects on which the hash values are computed must be immutable to make sure that the hash values will be constant in time. Indeed, hash value is computed from the state of the object (or eventually a sub-state of the object, then only this sub-state must be immutable).
* *To maintain security:* Look at this example: We have a file open method with login check. We pass a String to this method to process authentication which is necessary before the call will be passed to OS. If String was mutable it was possible somehow to modify its content after the authentication check before OS gets request from program then it is possible to request any file. So, if you have a right to open text file in user directory but then on the fly when somehow you manage to change the file name you can request to open "passwd" file or any other. Then a file can be modified and it will be possible to login directly to OS.

*If strings are mutable then there would be no String pool implementation and the performance gains of the String pool become lost.*

**How do we create/design an immutable class in C#? Give examples.**

In order to make a class immutable we must restrict changing the state of the class object by any means. Below are the guidelines.

* Avoid an assignment to a variable from outside the class. To further restrict the access we can use a private access modifier. Above do not provide any method where we modify the instance variables.
* Avoid sub classing. How if somebody creates a sub class from our up till now immutable class? Yes, here lies the problem. The new subclass can contain methods, which over ride our base class (immutable class) methods. Here he can change the variable values. Hence make the methods in the class also sealed. Or a better approach. Make the immutable class itself sealed. Hence cannot make any sub classes, so no question of over ridding.
* Don't allow writeable fields or properties or have methods that change the state of the type.

However, the questions started revolving around how System.String have methods to update the string but in reality it creates a new instance. All of this is really simple to achieve.

*Example:*

Let’s think of an Employee immutable class, where the class has a Name and Salary property and allows an increment to be made to the Salary. However, since Employee is immutable the Increment results in a new instance of the class to be created with an incremented Salary.

sealed class Employee

{

public Employee(string name, uint salary)

{

this.name = name;

this.salary = salary;

}

public Employee Increment(uint delta)

{

// A new instance is created

return new Employee (this.Name, this.Salary + delta);

}

readonly string name;

public string Name { get { return name; }}

readonly uint salary;

public uint Salary { get { return salary; }}

}

In the code above there is no field and the properties are get only. So there is no possibility of making state changes with these. The Increment method creates a new instance of the class with the increment without touching the class on which it is called. All of this makes Employee an immutable type.

**How to implement an immutable class that has auto-implemented properties?**

Below examples show how to create an immutable lightweight class that serves only to encapsulate a set of auto-implemented properties. Use this kind of construct instead of a struct when you must use reference type semantics.

Note that with auto-implemented properties, both a [get](http://msdn.microsoft.com/en-us/library/ms228503.aspx) and [set](http://msdn.microsoft.com/en-us/library/ms228368.aspx) accessor are required. You make the class immutable by declaring the **set** accessors as [private](http://msdn.microsoft.com/en-us/library/st6sy9xe.aspx). However, when you declare a private **set** accessor, you cannot use an object initializer to initialize the property. You must use a constructor or a factory method.

//Example 1:

// This class is immutable. After an object is created,

// it cannot be modified from outside the class. It uses a

// constructor to initialize its properties.

class Contact

{

// Read-only properties.

public string Name { get; private set; }

public string Address { get; private set; }

// Public constructor.

public Contact(string contactName, string contactAddress)

{

Name = contactName;

Address = contactAddress;

}

}

//Example 2:

// This class is immutable. After an object is created,

// it cannot be modified from outside the class. It uses a

// static method and private constructor to initialize its properties.

public class Contact2

{

// Read-only properties.

public string Name { get; private set; }

public string Address { get; private set; }

// Private constructor.

private Contact2(string contactName, string contactAddress)

{

Name = contactName;

Address = contactAddress;

}

// Public factory method.

public static Contact2 CreateContact(string name, string address)

{

return new Contact2(name, address);

}

}

**What is the difference between System.Text.StringBuilder and System.String?  
a.** Objects of type StringBuilder are mutable where as objects of type System.String are immutable.

**b.** Even after object of type StringBuilder is created, it can be able to modify length and content while once the string object is created, its length and content cannot be modified.

**c.** As StringBuilder objects are mutable, they offer better performance (faster) than string objects of type System.String.  
**d.** StringBuilder class is present in System.Text namespace where String class is present in [System](http://undefined) namespace.

**What’s the advantage of using System.Text.StringBuilder over System.String?**  
StringBuilder is more efficient in cases where there is a large amount of string manipulation.  Strings are immutable, so each time a string is changed, a new instance in memory is created.

**What is the difference between Convert.ToString and .ToString() method ?**

We can convert, say, the integer “i” using “i.ToString()” or “Convert.ToString” so what’s the difference. The basic difference between them is “Convert” function handles NULLS while “i.ToString()” does not it will throw a NULL reference exception error. So as good coding practice using “convert” is always safe.

**What namespaces are necessary to create a localized application?**System.Globalization and System.Resources.  
    
**How does one compare strings in C#?**

In the past, you had to call .ToString() on the strings when using the == or != operators to compare the strings’ values. That will still work, but the C# compiler now automatically compares the values instead of the references when the == or != operators are used on string types. If you actually do want to compare references, it can be done as follows: if ((object) str1 == (object) str2) { … } Here’s an example showing how string compares work:

using System;

public class StringTest

{

public static void Main(string[] args)

{

Object nullObj = null; Object realObj = new StringTest();

int i = 10;

Console.WriteLine("Null Object is [" + nullObj + "]\n"

+ "Real Object is [" + realObj + "]\n"

+ "i is [" + i + "]\n");

// Show string equality operators

string str1 = "foo";

string str2 = "bar";

string str3 = "bar";

Console.WriteLine("{0} == {1} ? {2}", str1, str2, str1 == str2 );

Console.WriteLine("{0} == {1} ? {2}", str2, str3, str2 == str3 );

}

}

Output:

Null Object is []

Real Object is [StringTest]

i is [10]

foo == bar ? False

bar == bar ? True

**What is the difference between string keyword and System.String class?**   
**string** keyword is an alias for **Syste.String** class. Therefore, System.String and string keyword are the same, and you can use whichever naming convention you prefer. The String class provides many methods for safely creating, manipulating, and comparing strings.  
  
**Are string objects mutable or immutable?**   
String objects are immutable.  
  
**What do you mean by String objects are immutable?***String objects are immutable means, they cannot be changed after they have been created.* All of the String methods and C# operators that appear to modify a string actually return the results in a new string object. In the following example, when the contents of s1 and s2 are concatenated to form a single string, the two original strings are unmodified. The += operator creates a new string that contains the combined contents. That new object is assigned to the variable s1, and the original object that was assigned to s1 is released for garbage collection because no other variable holds a reference to it.  
  
string s1 = "First String ";  
string s2 = "Second String";  
  
// Concatenate s1 and s2. This actually creates a new  
// string object and stores it in s1, releasing the  
// reference to the original object.  
s1 += s2;  
  
System.Console.WriteLine(s1);  
// Output: First String Second String  
  
**What will be the output of the following code?**   
*string str1 = "Hello ";  
string str2 = s1;  
str1 = str1 + "C#";  
System.Console.WriteLine(s2);*The output of the above code is "Hello" and not "Hello C#". This is because, if you create a reference to a string, and then "modify" the original string, the reference will continue to point to the original object instead of the new object that was created when the string was modified.  
  
**What is a verbatim string literal and why do we use it?**   
The "@" symbol is the verbatim string literal. Use verbatim strings for convenience and better readability when the string text contains backslash characters, for example in file paths. Because verbatim strings preserve new line characters as part of the string text, they can be used to initialize multiline strings. Use double quotation marks to embed a quotation mark inside a verbatim string. The following example shows some common uses for verbatim strings:  
  
*string ImagePath = @"C:\Images\Buttons\SaveButton.jpg";*//Output: C:\Images\Buttons\SaveButton.jpg  
  
string MultiLineText = @"This is multiline  
Text written to be in  
three lines.";  
/\* Output:  
This is multiline  
Text written to be in  
three lines.  
\*/  
  
string DoubleQuotesString = @"My Name is ""Vankat.""";  
//Output: My Name is "Vankat."

**Will the following code compile and run?**   
*string str = null;  
Console.WriteLine(str.Length);*The above code will compile, but at runtime System.NullReferenceException will be thrown.  
  
**How do you create empty strings in C#?**   
Using string.empty as shown in the example below.  
string EmptyString = string.empty;  
  
**How do you determine whether a String represents a numeric value?**To determine whether a String represents a numeric value use TryParse method as shown in the example below. If the string contains nonnumeric characters or the numeric value is too large or too small for the particular type you have specified, TryParse returns false and sets the out parameter to zero. Otherwise, it returns true and sets the out parameter to the numeric value of the string.  
  
string str = "One";  
int i = 0;  
if(int.TryParse(str,out i))  
{  
Console.WriteLine("Yes string contains Integer and it is " + i);  
}  
else  
{  
Console.WriteLine("string does not contain Integer");  
}  
  
**What is the difference between int.Parse and int.TryParse methods?**   
Parse method throws an exception if the string you are trying to parse is not a valid number where as TryParse returns false and does not throw an exception if parsing fails. Hence TryParse is more efficient than Parse.

**Delegates and Events**

**What are the differences between delegate and event?**

1. A delegate is a type-safe, object-oriented function pointer.  
2. A delegate declartion is C-sharp, f.e., public delegate void  
MyDelegate(int); defines a MulticastDelgate which contains a linked-list of  
delegates that can be added using .Combine method.  
3. An event object is an implementation of the Observer design pattern.  
4. An event is declared thus:  
public event MyDelegate MyEvent;  
5. An event is simply a wrapper around a MulticastDelegate, the C-sharp  
compiler will generate static += and -= operators to more easily add or  
subtract MulticastDelegates to the underlying MulticastDelegate.

**What is a delegate? How is it different from C++ function pointers?**

A **delegate** is a form of [type-safe](http://safety) [function pointer](http://pointer) used by the [.NET Framework](http://framework). Delegates specify a method to call and optionally an object to call the method on. They are used, among other things, to implement callbacks and [event listeners](http://listener). It encapsulates a reference to a method inside a delegate object. The delegate object can then be passed to code which can call the referenced method, without having to know at compile time which method will be invoked.

In other words delegates are function pointers that point to function of matching signatures. Function pointers which are extensively used in c/c++ to points to a function holds only the memory address of the function, it doesn’t carry further information about the function parameters, return type etc. On the other hand .NET framework has introduced a type-safe mechanism called delegates, with automatic verification of the signature by the compiler.

So comparatively delegates add a safety dimension in handling function pointers in .NET.

So we can say that *delegates are type-safe, object oriented, secure .NET objects which can be used to invoke methods of matching signature.*

While using delegates it is very much necessary to make sure that the functions which the delegates points has the same number of argument type and same return type. For example if we have a method that takes a single string as a parameter and another method that takes two string parameters, then we need to have two separate delegate type for each method.

**What are the different types of delegates?**

Delegates are of two types:

* Single cast delegate
* Multi cast delegate

**What is a single cast delegate? Give example.**

A delegate is called single cast delegate if it invokes a single method. In other words we can say that SingleCast Delegates refer to a single method with matching signature. SingleCast Delegates derive from the System.Delegate class

The signature of a single cast delegate is shown below. The letters in italics can be replaced with your own names and parameters.

| public delegate Boolean DelegateName (parm1, parm2) |
| --- |

When the compiler compiles the statement above, it internally generates a new class type. This class is called DelegateName and derives from System.Delegate. Just to make sure just check the ILDisassembler code to check that this is happening. As an example let us create a single cast delegate named MyDelegate which would point to a function MyFunction. The code appears as below,

| public delegate Boolean MyDelegate(Object sendingobj, Int32 x);  public class TestDelegateClass  {  Boolean MyFunction(Object sendingobj, Int32 x)  {  //Perform some processing  return (true);  }    public static void main(String [] args)  {  //Instantiate the delegate passing the method to invoke in its constructor  MyDelegate mdg = new MyDelegate(MyFunction);    // Construct an instance of this class  TestDelegateClass tdc = new TestDelegateClass();    // The following line will call MyFunction  Boolean f = mdg(this, 1);    }  } |
| --- |

**What is a multi cast delegate? Give example.**

MultiCast Delegates are nothing but a single delegate that can invoke multiple methods of matching signature. MultiCast Delegate derives from System.MulticastDelegate class which is a subclass of System.Delegate.

In Multi-Casting basically we create a single delegate that in turn invokes multiple encapsulated methods. We can use MultiCast Delegates when multiple calls to different methods are required. For example if we are required to call two methods on a single button click event or mouse over event then using MultiCast Delegates we can easily call the methods.

**How do you define a delegate?**

Defining a delegate is perhaps easier than defining a class. Generally, in fact, it is just one or two lines of code.

public delegate bool TemperatureChangeHandler(float newTemperature);

What is less obvious about the one or two lines of code, is that a delegate type definition is a shorthand syntax for defining a class.

The C# compiler produces a class that derives from System.MulticastDelegate

// ERROR: 'GreaterThanHandler' cannot  
// inherit from special class 'System.Delegate'  
public class TemperatureChangeHandler: System.MulticastDelegate  
{  
...  
}

Furthermore, System.MulticastDelegate derives from System.Delegate. However, the delegate definition syntax is not just a short hand. It turns out, the C# compiler prevents the declaration any class that derives (either directly or indirectly) from System.Delegate and therefore System.MulticastDelegate.

Each level within the delegate hierarchy provides a different set of services. System.Delegate is a container of the data for what method to call on a particular object. With System.MulticastDelegate comes the additional capability of not only invoking a method on a single object, but on a collections of objects. This enables multiple subscribers to an event.

**Why delegate types are derived from MulticastDelegate class why not it directly derive from Delegate class?**

<http://stackoverflow.com/questions/4833053/why-delegate-types-are-derived-from-multicastdelegate-class-why-not-it-directly-d>

I have a very basic question regarding delegate types. I compared the memebers of Delegate and MulticastDelegate classes in object browser and I couldn't find any new additional member present in MulticastDelegate. I also noticed that the Delegate class has GetInvocationList virtual method. So I assume that the Delegate class should have the capability to hold references to multiple methods. If my assumption is correct I wonder why not custom delegate types directly derive from the Delegate class instead of MulticastDelegate class.

Basically the split of Delegate and MulticastDelegate is for historical reasons. Originally there were going to be delegates which *couldn't* be combined and ones which could... but that turned out not to be a useful distinction. Apparently that was only discovered when it was a bit too late to rip MulticastDelegate out of the framework/CLR.

From CLR via C#, 3rd edition:

*The System.MulticastDelegate class is derived from System.Delegate, which is itself derived from System.Object. The reason why there are two delegate classes is historical and unfortunate; there should be just one delegate class in the FCL. Sadly, you need to be aware of both of these classes because even though all delegate types you create have MulticastDelegate as a base class, you'll occasionally manipulate your delegate types by using methods defined by the Delegate class instead of the MulticastDelegate class. [...]*

**What are the advantages System.Delegate type deriving from System. MulticastDelegate?**

***System.Delegate***

The purpose of a single delegate instance is very similar to a method pointer from C++. However, in C# we don’t use method pointers, rather, we save the “metadata” that identifies the target method to call. System.Delegate contains two critical data elements. Firstly, it contains an instance of System.Reflection.MethodInfo – in other words, the .NET metadata that enables method invocation using reflection.

The second aspect of System.Delegate is the object instance on which the method needs to be invoked. Given an unlimited number of objects that could support a method that matches the MethodInfo signature, we also need to be able to identify which objects to notify. The only exception is when the method identified by MethodInfo is static – in which case the object reference stored by System.Delegate is null.

**System.MulticastDelegate**

On its own, System.Delegate only identifies one subscriber to an event. This would work for a method pointer, for example a predicate used in a sort routine, but it would rarely be sufficient in a publish-subscribe/observer pattern. (In fact, in early betas of C# 1.0, Microsoft realized that using a single delegate was rare enough to avoid the need of even providing a means of defining a “single-cast” delegate.)

System.MulticastDelegate therefore, adds to delegates the support for notifying multiple subscribers. This is enabled through System.MulticastDelegate’s containment of another System.MulticastDelegate instance. When we add a subscriber to a multicast delegate, the MulticastDelegate class creates a new instance of the delegate type, stores the object reference and the method pointer for the added method into the new instance, and adds the new delegate instance as the next item in a list of delegate instances. In effect, the MulticastDelegate class maintains a linked list of delegate objects.

**Is there a Delegate which isn't a MulticastDelegate in C#?**

**<http://stackoverflow.com/questions/4220239/which-thread-does-backgroundworker-completed-event-handler-run-on>**

**How to create multicast delegates?**

A useful property of [delegate](http://msdn.microsoft.com/en-us/library/900fyy8e.aspx) objects is that multiple objects can be assigned to one delegate instance by using the + operator. The multicast delegate contains a list of the assigned delegates. When the multicast delegate is called, it invokes the delegates in the list, in order. Only delegates of the same type can be combined.

The - operator can be used to remove a component delegate from a multicast delegate.

using System;

// Define a custom delegate that has a string parameter and returns void.

delegate void CustomDel(string s);

class TestClass

{

// Define two methods that have the same signature as CustomDel.

static void Hello(string s)

{

System.Console.WriteLine(" Hello, {0}!", s);

}

static void Goodbye(string s)

{

System.Console.WriteLine(" Goodbye, {0}!", s);

}

static void Main()

{

// Declare instances of the custom delegate.

CustomDel hiDel, byeDel, multiDel, multiMinusHiDel;

// In this example, you can omit the custom delegate if you

// want to and use Action<string> instead.

//Action<string> hiDel, byeDel, multiDel, multiMinusHiDel;

// Create the delegate object hiDel that references the

// method Hello.

hiDel = Hello;

// Create the delegate object byeDel that references the

// method Goodbye.

byeDel = Goodbye;

// The two delegates, hiDel and byeDel, are combined to

// form multiDel.

multiDel = hiDel + byeDel;

// Remove hiDel from the multicast delegate, leaving byeDel,

// which calls only the method Goodbye.

multiMinusHiDel = multiDel - hiDel;

Console.WriteLine("Invoking delegate hiDel:");

hiDel("A");

Console.WriteLine("Invoking delegate byeDel:");

byeDel("B");

Console.WriteLine("Invoking delegate multiDel:");

multiDel("C");

Console.WriteLine("Invoking delegate multiMinusHiDel:");

multiMinusHiDel("D");

}

}

/\* Output:

Invoking delegate hiDel:

Hello, A!

Invoking delegate byeDel:

Goodbye, B!

Invoking delegate multiDel:

Hello, C!

Goodbye, C!

Invoking delegate multiMinusHiDel:

Goodbye, D!

\*/

**What is the problem of invoking each delegate instance inside multicast delegate in a sequential manner?**

When invoking the multicast delegate, each delegate instance in the linked list is called sequentially. (Generally, delegates are called in the order they were added but this behavior is not specified within the CLI specification and furthermore, it can be overridden. Therefore, programmers should not depend on an invocation order.) This sequential invocation, however, leads to problems if the invoked method throws an exception or if the delegate itself returns data.

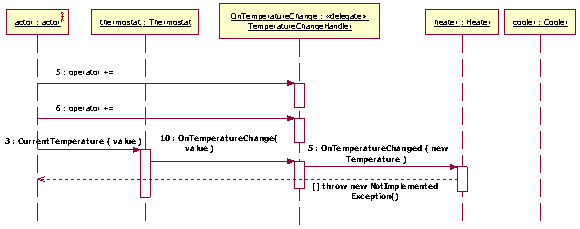
Error handling makes awareness of the sequential notification critical. If one subscriber throws an exception then later subscribers in the chain do not receive the notification.

Consider, for example, a heater class with an OnTemperatureChanged() method that threw an exception as shown in Listing 3.

Listing 3: OnTemperatureChanged() Throwing an Exception

class Heater  
{  
...  
public voidOnTemperatureChanged( floatnewTemperature)  
{  
throw new NotImplementedException();  
}  
...  
}

Figure 1 shows the effect in a sequence diagram where cooler is an additional subscriber that also has a method matching the TemperatureChangeHandler() signature.



**Figure 1: Delegate Invocation with Exception Sequence Diagram**

Even though cooler subscribed to receive messages, heater exception terminates the chain and prevents the cooler object from receiving notification.

To avoid this problem, so that all subscribers receive notification regardless of the behavior of earlier subscribers, you must manually enumerate through the list of subscribers and call them individually. Listing 4shows an implementation for a CurrentTemperature property that fires change notifications:

Listing 4: Handling Exceptions from Subscribers

public class Thermometer  
{  
 // Define the delegate data type  
 public delegatevoid TemperatureChangeHandler(float newTemperature);  
 // Define the event publisher  
 public event TemperatureChangeHandler OnTemperatureChange;

private float \_CurrentTemperature;  
 public float CurrentTemperature  
 {  
 get{return \_CurrentTemperature;}  
 set  
 {  
 if (value != CurrentTemperature)  
 {  
 \_CurrentTemperature = value;  
 if(OnTemperatureChange != null)  
 {  
 foreach(TemperatureChangeHandler handler in  
 OnTemperatureChange.GetInvocationList() )  
 {  
 try  
 {  
 handler(value);  
 }  
 catch(Exception exception)  
 {  
 Console.WriteLine(exception.Message);  
 }  
 }  
 }  
 }  
 }  
 }  
}

Output:

Enter temperature: 45

The method or operation is not implemented.

Cooler: Off

This listing demonstrates that we can retrieve a list of subscribers from a delegates GetInvocationList() method. Enumerating over each item in this list returns the individual subscribers. If we then place each invocation of a subscriber within a try catch block we can handle any error conditions before continuing on with the enumeration loop. In our sample, even though heater.OnTemperatureChanged() throws an exception, cooler still receives the notification of the temperature change.

**Delegate object is pointing to 5 methods? How the methods would get called? If there is an exception thrown by 2nd method, what is going to happen? How do you ensure that delegate continues to execute remaining methods?**

**Or, how to handle exception in multicast delegate in C#?**

<http://stackoverflow.com/questions/5740650/how-to-handle-exception-in-multicast-delegate-in-c>

You can loop through all the delegates’ registered in the multicast list and call each of them in turn while wrapping each call in a try - catch block.

Otherwise the invocations of the subsequent delegates in the multicast after the delegate with the exception will be aborted. Consider the code using GetInvocationList

foreach (var singleDelegate in theEvent.GetInvocationList()) {  
 try {  
 singleDelgate.DynamicInvoke(new object[] { sender, eventArg });  
 } catch (Exception ex) {  
 // uck  
 }  
}

which individually calls each delegate that would have been invoked with

theEvent.Invoke(sender, eventArg)

Remember to do the standard null-guard and copy on theEvent as required.

**In C#, how can we define the multicast delegate which accepts a DateTime object and return a boolean. How do you get all the return values from a multicast delegate? Or, how do you get the return types from multiple methods called by multicast delegate?**

public delegate bool Foo(DateTime timestamp);

This is how to declare a delegate with the signature you describe. All delegates are potentially multicast, they simply require initialization. Such as:

public bool IsGreaterThanNow(DateTime timestamp)  
{  
 return DateTime.Now < timestamp;  
}  
  
public bool IsLessThanNow(DateTime timestamp)  
{  
 return DateTime.Now > timestamp;  
}  
  
Foo f1 = IsGreaterThanNow;  
Foo f2 = IsLessThanNow;  
Foo fAll = f1 + f2;

Calling fAll, in this case would call both IsGreaterThanNow() and IsLessThanNow().

What this doesn't do is give you access to each return value. All you get is the last value returned. If you want to retrieve each and every value, you'll have to handle the multicasting manually like so:

foreach(Foo f in fAll.GetInvocationList())  
{  
 List<bool> returnValues = new List<bool>();  
 returnValues.Add(f(timestamp));  
}

**Here is my scenario. I have class with 10 methods, those methods are atomic, and are only 10 lines of code max. So I was thinking, instead of handling the exceptions in each function, would it be possible, to create a delegate (? not sure its the right word here)**

**That so called wrapped function would execute those atomic functions and handle their exceptions, thus allowing me to centralize exception handling.**

**Is this possible in C# using delegate or Func, or maybe there is another way to centralize error handling that I might have missed?**

Yes, it is possible here is a code example.

using System;  
namespace ConsoleApp  
{  
 class Example {  
 public void Run() {  
 catchy(crashA); // Calling defined functions  
 catchy(crashB);  
 catchy(()=> {  
 throw new ArgumentException("Anonymous function...");  
 });  
 }  
 void crashA() {  
 //...  
 throw new ArgumentException("another error");  
 }  
 void crashB() {  
 //...  
 throw new ArgumentException("another error");  
 }  
 void catchy(Action action) {  
 try {  
 action();  
 } catch (Exception ex) {  
 Console.WriteLine(ex);  
 // do something  
 }  
 }  
 }  
  
 class MainClass  
 {  
 public static void Main (string[] args)  
 {  
 new Example().Run();  
 Console.ReadLine();  
 Console.WriteLine ("Hello World!");

}  
 }

}

**What's the difference between Invoke() and BeginInvoke()**

* Delegate.Invoke: Executes synchronously, on the same thread.
* Delegate.BeginInvoke: Executes asynchronously, on a thread pool thread.
* Control.Invoke: Executes on the UI thread, but calling thread waits for completion before continuing.
* Control.BeginInvoke: Executes on the UI thread, and calling thread doesn't wait for completion.

Control.Invoke/BeginInvoke executes on the thread that owns the Control's handle (most of time it is the UI thread).

For Windows Forms apps, I would suggest that you should *usually* use BeginInvoke. That way you don't need to worry about deadlock, for example - but you need to understand that the UI may not have been updated by the time you next look at it! In particular, you shouldn't modify data which the UI thread might be about to use for display purposes. For example, if you have a Person with FirstName and LastName properties, and you did:

person.FirstName = "Kevin"; // person is a shared reference  
person.LastName = "Spacey";  
control.BeginInvoke(UpdateName);  
person.FirstName = "Keyser";  
person.LastName = "Soze";

then the UI may well end up displaying "Keyser Spacey". (There's an outside chance it could display "Kevin Soze" but only through the weirdness of the memory model.)

Unless you have this sort of issue, however, Control.BeginInvoke is easier to get right, and will avoid your background thread from having to wait for no good reason. Note that the Windows Forms team has guaranteed that you can use Control.BeginInvoke in a "fire and forget" manner - i.e. without ever calling EndInvoke. This is not true of async calls in general: normally every BeginXXX should have a corresponding EndXXX call, usually in the callback.

**What are the uses of delegates?**

* In .NET you can call any method asynchronously by defining a delegate for the method and calling the delegate's asynchronous methods. This is beneficial to your application because when a synchronous call is made, the calling thread is blocked until the method completes whereas an asynchronous call is made on a different thread, and this allows the original thread to continue its work while the asynchronous call is in progress.
* Events are declared using delegates.

**Why use delegates? Or what are the advantages of delegates?**

**Why would I use a delegate?**To implement callbacks and [event listeners](http://listener).

**Can anyone provide a real-world example of when I should use a delegate?**For example I have a threading that can be launched by a variety of different objects and it takes in a callback function to notify the calling object when certain stages are complete etc. The callback is a function of the object and has it's own implementation, but the thread just knows to call it at certain points. Progress bars etc can make good use of callback functions.   
  
**An example of a task that can't be done without the use of a delegate?**   
Event driven programming in .NET. Delegates are used to wire everything up.   
  
**What is the relationship between Delegates and Events?**Events use delegates to hookup actions. For instance, the onClick event for a button is hooked up to the button via a delegate of the onclick method you add.

**What is the main difference between delegate and an event in c#?**

**Can we declare delegates in class and can we declare delegates in Interface?**

Yes and no. The thing to remember about delegates is they are in themselves  
classes that inherit from System.Delegate. Since nested classes are allowed  
in classes you might declare a delegate in a class. For example the following  
is legal:  
  
public class MyClass  
{  
public delegate void MyDelegate();  
}  
  
To reference that delegate outside of MyClass you would write the following:  
  
MyClass.MyDelegate myDel;  
  
Since a delegate is a type that inherits from delegate you cannot declare a  
delegate within an interface, because you cannot declare nested types in an  
interface. That being said it is recommended to declare delegate type outside of a class  
and inside a namespace, because delegates really are just classes.

**What is an event?**

*An event in C# is a way for a class to provide notifications to clients of that class when some interesting thing happens to an object.* The most familiar use for events is in graphical user interfaces; typically, the classes that represent controls in the interface have events that are notified when the user does something to the control (for example, click a button).

Events, however, need not be used only for graphical interfaces. *Events provide a generally useful way for objects to signal state changes that may be useful to clients of that object.* Events are an important building block for creating classes that can be reused in a large number of different programs.

*Events are declared using delegates.* An event is a way for a class to allow clients to give it delegates to methods that should be called when the event occurs. When the event occurs, the delegate(s) given to it by its clients are invoked.

**Can we have events in an interface? Give example. What is the difference between events and fields?**

Yes. Difference between events and fields is that an event can be placed in an interface while a field cannot. When implementing the interface, the implementing class must supply a corresponding event in the class that implements the interface.

This example shows that it is possible to declare an event in an interface and implement it in a class:

// event\_keyword.cs

using System;

public delegate void MyDelegate(); // delegate declaration

public interface I

{

event MyDelegate MyEvent;

void FireAway();

}

public class MyClass: I

{

public event MyDelegate MyEvent;

public void FireAway()

{

if (MyEvent != null)

MyEvent();

}

}

public class MainClass

{

static private void f()

{

Console.WriteLine("This is called when the event fires.");

}

static public void Main ()

{

I i = new MyClass();

i.MyEvent += new MyDelegate(f);

i.FireAway();

}

}

**How to invoke events from derived class?**

When creating a general component that can be derived from, what seems to be a problem sometimes arises with events. Since events can only be invoked from within the class that declared them, derived classes cannot directly invoke events declared within the base class. Although this is sometimes what is desired, often it is appropriate to give the derived class the freedom to invoke the event. This is typically done by creating a protected invoking method for the event. By calling this invoking method, derived classes can invoke the event. For even more flexibility, the invoking method is often declared as virtual, which allows the derived class to override it. This allows the derived class to intercept the events that the base class is invoking, possibly doing its own processing of them.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Generics**

**What is a generic type?**

A generic type is a type that uses generic type parameters. For example, the type LinkedList<K,T>, defined as:

public class LinkedList<K,T>

{...}

is a generic type, because it uses the generic type parameters K and T, where K is the list's key and T is the type of the data item stored in the list. What is special about generic types is that you code them once, yet you can use them with different parameters. Doing so has significant benefits—you reuse your development and testing efforts, without compromising type safety and performance, and without bloating your code.

**What are the benefits of using generics?**

Generics provide the following big benefits to developers as exhibited by the code just shown:

• **Source code protection** The developer using a generic algorithm doesn't need to have

access to the algorithm's source code. With C++ templates or Java's generics, however,

the algorithm's source code must be available to the developer who is using the

algorithm.

• **Type Safety** When a generic algorithm is used with a specific type, the compiler and

the CLR understand this and ensure that only objects compatible with the specified

data type are used with the algorithm. Attempting to use an object of an incompatible

type will result in either a compiler error or a run-time exception being thrown. In the

example, attempting to pass a String object to the Add method results in the compiler

issuing an error.

• **Cleaner Code** Since the compiler enforces type safety, fewer casts are required in your

source code, meaning that your code is easier to write and maintain. In the last line of

SomeMethod, a developer doesn't need to use a (DateTime) cast to put the result of the

indexer (querying element at index 0) into the dt variable.

• **Better Performance** Before generics, the way to define a generalized algorithm was to

define all of its members to work with the Object data type. If you wanted to use the

algorithm with value type instances, the CLR had to box the value type instance prior to

calling the members of the algorithm. As discussed in Chapter 5, "Primitive, Reference,

and Value Types," boxing causes memory allocations on the managed heap, which

causes more frequent garbage collections, which, in turn, hurt an application's performance.

Since a generic algorithm can now be created to work with a specific value

type, the instances of the value type can be passed by value, and the CLR no longer

has to do any boxing. In addition, since casts are not necessary (see the previous

bullet), the CLR doesn't have to check the type safety of the attempted cast, and this

results in faster code too.

•**Reuse development and testing efforts** You code them once, yet you can use them with different parameters.

**What Is a Generic Type Parameter?**

A generic type parameter is the place holder a generic type uses. For example, the generic type LinkedList<K,T>, defined as:

public class LinkedList<K,T>

{...}

uses two type parameters - K and T, where K is the list's key and T is the type of the data item stored in the list. Using generic type parameters allows the linked list to defer the decision on the actual types to use. In fact, it is up to the client of the generic linked list to specify the generic type parameters to use.

**What Is a Generic Type Argument?**

A generic type argument is the type the client specifies to use instead of the type parameter. For example, given this generic type definition and declaration:

public class MyClass<T>

{...}

MyClass<int> obj = new MyClass<int>();

T is the type parameter, while integer is the type argument.

**What Is a Constructed Type?**

A constructed type is any generic type that has at least one type argument. For example, given this generic linked list definition:

public class LinkedList<T>

{...}

Then the following is a constructed generic type:

LinkedList<string>

To qualify as a constructed type you can also specify type parameters to the generic type:

public class MyClass<T>

{

LinkedList<T> m\_List; //Constructed type

}

**What Is an Open Constructed Type?**

A open constructed type is any generic type that which contains at least one type parameter used as a type argument. For example, given this definition:

public class LinkedList<K,T>

{...}

Then the following declarations of LinkedList<K,T> member variables are all open constructed types:

public class MyClass<K,T>

{

LinkedList<K,T> m\_List1; //Open constructed type

LinkedList<K,string> m\_List2; //Open constructed type

LinkedList<int,T> m\_List3; //Open constructed type

}

**What Is a Closed Constructed Type?**

A closed constructed type is a generic type that which contains no type parameters as type arguments. For example, given this definition:

public class LinkedList<K,T>

{...}

Then the following declarations of LinkedList<K,T> member variables are all closed constructed types:

LinkedList<int,string> list1; //Closed constructed type

LinkedList<int,int> list2; //Closed constructed type

**How Do I Use a Generic Type?** Or **How Do I Initialize a Generic Type Parameter?**

When declaring a generic type, you need to specify the types that will replace the type parameters in the declaration. These are known as type arguments to the generic type. Type arguments are simply types. For example, when using this generic linked list:

public class LinkedList<K,T>

{

public void AddHead(K key,T item);

//Rest of the implementation

}

You need to specify which types to use for K, the list's key, and T, the data items stored in the list. You specify the types in two places: when declaring the list's variable and when instantiating it:

LinkedList<int,string> list = new LinkedList<int,string>();

list.AddHead(123,"ABC");

Once you specify the types to use, you can simply call methods on the generic type, providing appropriate values of the previously specified types.

A generic type that has type arguments already, such as LinkedList<int,string> is called a constructed type.

When specifying type arguments for generic types, you can actually provide type parameters. For example, consider this definition of the Node<K,T> class, which is used as a node in a linked list:

class Node<K,T>

{

public K Key;

public T Item;

public Node<K,T> NextNode;

public Node(K key,T item,Node<K,T> nextNode)

{

Key = key;

Item = item;

NextNode = nextNode;

}

}

The Node<K,T> class contains as a member variable a reference to the next node. That member must be provided with the type to use instead of its generic type parameters. The node specifies its own type parameters in this case.

Another example of specifying generic type parameters to a generic type is how the linked list itself may declare and use the node:

public class LinkedList<K,T>

{

Node<K,T> m\_Head;

public void AddHead(K key,T item)

{...}

}

Note that the use of K and T in the linked list as the names of the type arguments is purely for readability purposes, to make the use of the node more consistent. You could have defined the linked list with any other generic type parameter names, in which case, you need to pass them along to the node as well:

public class LinkedList<Key,Item>

{

Node<Key,Item> m\_Head;

public void AddHead(Key key,Item item)

{...}

}

**Why Can't I Use Type-Specific Data Structures Instead of Generics?**

To avoid the type-safety problem without generics, you might be tempted to use type-specific interfaces and data structure, for example:

public interface IIntegerList

{

int Add(int value);

bool Contains(int value);

int IndexOf(int value);

void Insert(int index, int value);

void Remove(int value);

int this[int index]{ get; set; }

//Additional members

}

The problem with that approach is that you will need a type-specific interface and implementation per data type you need to interact with, such as a string or a Customer. If you have a defect in your handling of the data items, you will need to fix it in as many places as types, and that is simply error-prone and impractical. With generics, you get to define and implement your logic once, yet use it with any type you want.

**When Should I Use Generics?**

You should use generics whenever you have the option to. Meaning, if a data structure or a utility class offers a generic version, you should use the generic version, not the object-based methods. The reason is that generics offer significant benefits, including productivity, type safety and performance, at literally no cost to you. Typically, collections and data structures such as linked lists, queues, binary trees etc will offer generics support, but generics are not limited to data structures. Often, utility classes such as class factories or formatters also take advantage of generics. The one case where you should not take advantage of generics is cross-targeting. If you develop your code to target .NET 1.1 or earlier, then you should not use any of the new .NET 2.0 features, including generics. In C# 2.0, you can even instruct the compiler in the project settings (under Build | Advanced) to use only C# 1.0 syntax (ISO-1).

**Are Generics Covariant, Contra-Variant, or Invariant?**

Generic types are not covariant. Meaning, you cannot substitute a generic type with a specific type argument, with another generic type that uses a type argument that is the base type for the first type argument. For example, the following statement does not compile:

class MyBaseClass

{}

class MySubClass : MyBaseClass

{}

class MyClass<T>

{}

//Will not compile

MyClass<MyBaseClass> obj = new MyClass<MySubClass>();

Using the same definition as in the example above, it is also true that MyClass<MyBaseClass> is not the base type of MyClass<MySubClass>:

Debug.Assert(typeof(MyClass<MyBaseClass>) != typeof(MyClass<MySubClass>).BaseType);

This would not be the case if the generic types were contra-variant.

Because generics are not covariant, when overriding a virtual method that returns a generic type parameter, you cannot provide a subtype of that type parameter as the definition of the overriding method:

For example, the following statement does not compile:

class MyBaseClass<T>

{

public virtual T MyMethod()

{...}

}

class MySubClass<T,U> : MyBaseClass<T> where T : U

{

//Invalid definition:

public override U MyMethod()

{...}

}

That said, constraints are covariant. For example, you can satisfy a constraint using a sub type of the constraint's type:

class MyBaseClass

{}

class MySubClass : MyBaseClass

{}

class MyClass<T> where T : MyBaseClass

{}

MyClass<MySubClass> obj = new MyClass<MySubClass>();

MyClass<MySubClass ^> ^obj = gcnew MyClass<MySubClass ^>;

You can even further restrict constraints this way:

class BaseClass<T> where T : IMyInterface

{}

interface IMyOtherInterface : IMyInterface

{}

class SubClass<T> : BaseClass<T> where T : IMyOtherInterface

{}

Finally, generics are invariant, because there is no relationship between two generic types with different type arguments, even if those type arguments do have an is-as relationship, for example, List<int> has nothing to do with List<object>, even though an int is an object.

**What Can Define Generic Type Parameters? What Types Can Be Generic?**

Classes, interfaces, structures and delegates, can all be generic types. Here are a few examples from the .NET Framework:

public interface IEnumerator<T> : IEnumerator,IDisposable

{

T Current{get;}

}

public class List<T> : IList<T> //More interfaces

{

public void Add(T item);

public bool Remove(T item);

public T this[int index]{get;set;}

//More members

}

public struct KeyValuePair<K,V>

{

public KeyValuePair(K key,V value);

public K Key;

public V Value;

}

public delegate void EventHandler<E>(object sender,E e) where E : EventArgs;

In addition, both static and instance methods can rely on generic type parameters, independent of the types that contain them:

public sealed class Activator : \_Activator

{

public static T CreateInstance<T>();

//Additional memebrs

}

Enumerations on the other hand cannot define type parameters, and the same goes for attributes.

**Can Methods Define Generic Type Parameters? How Do I Call Such Methods?**

Yes. Both instance and static methods can define generic type parameters, and do so independently of their containing class. For example:

public class MyClass

{

public void MyInstanceMethod<T>(T t)

{...}

public static void MyStaticMethod<T>(T t)

{...}

}

The benefit of a method that defines generic type parameters is that you can call the method passing each time different parameter types, without ever overloading the method. When you call a method that defines generic type parameters, you need to provide the type arguments at the call site:

MyClass obj = new MyClass();

obj.MyInstanceMethod<int>(3);

obj.MyInstanceMethod<string>("Hello");

MyClass.MyStaticMethod<int>(3);

MyClass.MyStaticMethod<string>("Hello");

If type-inference is available, you can omit specifying the type arguments at the call site:

MyClass obj = new MyClass();

obj.MyInstanceMethod(3);

obj.MyInstanceMethod("Hello");

MyClass.MyStaticMethod(3);

MyClass.MyStaticMethod("Hello");

**Can I Derive From a Generic Type Parameter?**

You cannot define a class that derives from its own generic type parameter:

public class MyClass<T> : T //Does not compile

{...}

**What Is a Generic Type Inference?**

Generic type inference is the compiler's ability to infer which type arguments to use with a generic method, without the developer having to specify it explicitly. For example, consider the following definition of generic methods:

public class MyClass

{

public void MyInstanceMethod<T>(T t)

{...}

public static void MyStaticMethod<T>(T t)

{...}

}

When invoking these methods, you can omit specifying the type arguments for both the instance and the static methods:

MyClass obj = new MyClass();

obj.MyInstanceMethod(3); //Compiler infers T as int

obj.MyInstanceMethod("Hello");//Compiler infers T as string

MyClass.MyStaticMethod(3); //Compiler infers T as int

MyClass.MyStaticMethod("Hello");//Compiler infers T as string

Note that type inferring is possible only when the method takes an argument of the inferred type arguments. For example, in the CreateInstance<T>() method of the Activator class, defined as:

public sealed class Activator : \_Activator

{

public static T CreateInstance<T>();

//Additional memebrs

}

type inference is not possible, and you need to specify the type arguments at the call site:

class MyClass

{...}

MyClass obj = Activator.CreateInstance<MyClass>();

Note also that you cannot rely on type inference at the type level, only at the method level. In the following example, you must still provide the type argument T even though the method takes a T parameter:

public class MyClass<T>

{

public static void MyStaticMethod<U>(T t,U u)

{...}

}

MyClass<int>.MyStaticMethod(3,"Hello");//No type inference for the integer

**What Are Constraints?**

Constraints allow additional contextual information to be added to the type parameters of generic types. The constraints limit the range of types that are allowed to be used as type arguments, but at the same time, they add information about those type parameters. Constraints ensure that the type arguments specified by the client code are compatible with the generic type parameters the generic type itself uses. Meaning, constraints prevent the client from specifying types as type arguments that do not offer the methods, properties, or members of the generic type parameters that the generic type relies upon.

After applying a constraint you get IntelliSense reflecting the constraints when using the generic type parameter, such as suggesting methods or members from the base type.

There are three types of constraints:

Derivation constraint indicates to the compiler that the generic type parameters derives from a base type such an interface or a particular base class. For example, in the following example, the linked list applies a constraint of deriving from IComparable<T> on its generic type parameter. This is required so that you could implement a search. sorting or indexing functionality on the list:

class Node<K,T>

{

public K Key;

public T Item;

public Node<K,T> NextNode;

}

public class LinkedList<K,T> where K : IComparable<K>

{

Node<K,T> m\_Head;

public T this[K key]

{

get

{

Node<K,T> current = m\_Head;

while(current.NextNode != null)

{

if(current.Key.CompareTo(key) == 0)

break;

else

current = current.NextNode;

}

return current.Item;

}

}

//Rest of the implementation

}

You can provide constraints for every generic type parameter that your class declares, for example:

public class LinkedList<K,T> where K : IComparable<K>

where T : ICloneable

You can have a base class constraint, meaning, stipulating that the generic type parameter derives from a particular base class:

public class MyBaseClass

{...}

public class MyClass<T> where T : MyBaseClass

{...}

However, you can only use one base class at most in a constraint because neither C#, Visual Basic or managed C++ support multiple inheritance of implementation. Obviously, the base class you constrain to cannot be a sealed class, and the compiler enforces that. In addition, you cannot constrain System.Delegate or System.Array as a base class.

You can constrain both a base class and one or more interfaces, but the base class must appear first in the derivation constraint list:

public class LinkedList<K,T> where K : MyBaseKey,IComparable<K>

{...}

The constructor constraint indicates to the compiler that the generic type parameter exposes a default public constructor (a public constructor with no parameters). For example:

class Node<K,T> where K : new()

where T : new()

{

public K Key;

public T Item;

public Node<K,T> NextNode;

public Node()

{

Key = new K(); //Compiles because of the constraint

Item = new T(); //Compiles because of the constraint

NextNode = null;

}

//Rest of the implementation

}

You can combine the default constructor constraint with derivation constraints, provided the default constructor constraint appears last in the constraint list:

public class LinkedList<K,T> where K : IComparable<K>,new()

where T : new()

{...}

The reference and value type constraint is used to constrain the generic type parameter to be a value or a reference type. For example, you can constrain a generic type parameter to be a value type (such as an int, a bool, and enum, or any structure):

public class MyClass<T> where T : struct

{...}

Similarly, you can constrain a generic type parameter to be a reference type (a class):

public class MyClass<T> where T : class

{...}

The reference and value type constraint cThe value/reference type constraint cannot be used with a base class constraint, but it can be combined with any other constraint. When used, the value/reference type constraint must appear first in the constraint list.

It is important to note that although constraints are optional, they are often essential when developing a generic type. Without constraints, the compiler follows the more conservative, type-safe approach and only allows access to object-level functionality in your generic type parameters. Constraints are part of the generic type metadata so that the client-side compiler can take advantage of them as well. The client-side compiler only allows the client developer to use types that comply with the constraints, thus enforcing type safety.

**What Can I Not Use Constraints With?**

You can only place a derivation constraint on a type parameter (be it an interface derivation or a single base class derivation). In C# and Visual Basic, you can also use a default constructor constraint and a value or reference type constraint. While everything else is implicitly not allowed, it is worth mentioning the specific cases that are not possible:

* You cannot constrain a generic type to have any specific parameterized construct.
* You cannot constrain a generic type to derive from a sealed class.
* You cannot constrain a generic type to derive from a static class.
* You cannot constrain a public generic type to derive from another internal type.
* You cannot constrain a generic type to have a specific method, be it a static or an instance method.
* You cannot constrain a generic type to have a specific public event.
* You cannot constrain a generic type parameter to derive from System.Delegate or System.Array.
* You cannot constrain a generic type parameterfundamentals\_topic2 to be serializable.
* You cannot constrain a generic type parameterfundamentals\_topic2 to be COM-visible.
* You cannot constrain a generic type parameter to have any particular attribute.
* You cannot constrain a generic type parameter to support any specific operator. There is therefore no way to compile the following code:

public class Calculator<T>

{

public T Add(T argument1,T argument2)

{

return argument1 + argument2; //Does not compile

}

//Rest of the methods

}

**Why Can I Not Use Enums, Structs, or Sealed Classes as Generic Constraints?**

You cannot constraint a generic type parameter to derive from a non-derivable type. For example, the following does not compile:

public sealed class MySealedClass

{...}

public class MyClass<T> where T : MySealedClass //Does not compile

{...}

The reason is simple: The only type arguments that could possibly satisfy the above constraint is the type MySealedClass itself, making the use of generics redundant. For this very reason, all other non-derivable types such as structures and enums are not allowed in constraints.

**Is Code that Uses Generics Faster than Code that Does Not?**

The answer depends on the way the non-generic code is written. If the code is using objects as the amorphous containers to store items, then various benchmarks have shown that in intense calling patterns, generics yield on average 100 percent performance improvement (that is, three times as fast) when using value types, and some 50 percent performance improvement when using reference types.

If the non-generic code is using type-specific data structures, then there is no performance benefit to generics. However, such code is inherently very fragile. Writing a type-specific data structure is a tedious, repetitive, and error-prone task. When you fix a defect in the data structure, you have to fix it not just in one place, but in as many places as there are type-specific duplicates of what essentially is the same data structure.

**Is an Application That Uses Generics Faster than an Application That Does Not?**

Depending on the application of course, but generally speaking, in most real-life applications, bottle necks such as I/O will mask out any performance benefit from generics. The real benefit of generics is not performance but rather type safety and productivity.

**What Is the Difference Between Using Generics and Using Interfaces (or Abstract Classes)?**

Interfaces and generics serve different purposes. Interfaces are about defining a contract between a service consumer and a service provider. As long as the consumer programs strictly against the interface (and not a particular implementation of it), it can use any other service provider that supports the same interface. This allows switching service providers without affecting (or with minimum effect on) the client's code. The interface also allows the same service provider to provide services to different clients. Interfaces are the cornerstone of modern software engineering, and are used extensively in past and future technologies, from COM to .NET to Indigo and SOA.

Generics are about defining and implementing a service without committing to the actual types used. As such, interfaces and generics are not mutually exclusive. Far from it, they compliment each other. You can and you should combine interfaces and generics.

For example, the interface ILinkedList<T> defined as:

public interface ILinkedList<T>

{

void AddHead(T item);

void RemoveHead(T item);

void RemoveAll();

}

Can be implemented by any linked list:

public class LinkedList<T> : ILinkedList<T>

{...}

public class MyOtherLinkedList<T> : ILinkedList<T>

{...}

You can now program against ILinkedList<T>, using both different implementations and different type arguments:

ILinkedList<int> numbers = new LinkedList<int>();

ILinkedList<string> names = new LinkedList<string>();

ILinkedList<int> moreNumbers = new MyOtherLinkedList<int>();

**How Are Generics Implemented?**

Generics have native support in IL and the CLR itself. When you compile generic server-side code, the compiler compiles it into IL, just like any other type. However, the IL only contains parameters or place holders for the actual specific types. In addition, the metadata of the generic server contains generic information such as constraints.

The client-side compiler uses that generic metadata to support type safety. When the client provides a type arguments, the client's compiler substitutes the generic type parameter in the server metadata with the specified type. This provides the client's compiler with type-specific definition of the server, as if generics were never involved. At run time, the actual machine code produced depends on whether the specified types are value or reference type. If the client specifies a value type, the JIT compiler replaces the generic type parameters in the IL with the specific value type, and compiles it to native code. However, the JIT compiler keeps track of type-specific server code it already generated. If the JIT compiler is asked to compile the generic server with a value type it has already compiled to machine code, it simply returns a reference to that server code. Because the JIT compiler uses the same value-type-specific server code in all further encounters, there is no code bloating.

If the client specifies a reference type, then the JIT compiler replaces the generic parameters in the server IL with object, and compiles it into native code. That code will be used in any further requests for a reference type instead of a generic type parameter. Note that this way the JIT compiler only reuses actual code. Instances are still allocated according to their size off the managed heap, and there is no casting.

**Why can't I use Operators on Naked Generic Type Parameters?**

The reason is simple—since there is no way to constrain a generic type parameter to support an operator, there is no way the compiler can tell whether the type specified by the client of the generic type will support the operator.

Consider for example the following code:

class Node<K,T>

{

public K Key;

public T Item;

public Node<K,T> NextNode;

}

public class LinkedList<K,T>

{

Node<K,T> m\_Head;

public T this[K key]

{

get

{

Node<K,T> current = m\_Head;

while(current.NextNode != null)

{

if(current.Key == key)) //Does not compile

break;

else

current = current.NextNode;

}

return current.Item;

}

}

//Rest of the implementation

}

The compiler will refuse to compile this line:

if(current.Key == key))

Because it has no way of knowing whether the type the consumer will specify will support the == operator.

**When Can I use Operators on Generic Type Parameters?**

You can use an operator (or for that matter, any type-specific method) on generic type parameters if the generic type parameter is constrained to be a type that supports that operator. For example:

class MyOtherClass

{

public static MyOtherClass operator+(MyOtherClass lhs,MyOtherClass rhs)

{

MyOtherClass product = new MyOtherClass();

product.m\_Number = lhs.m\_Number + rhs.m\_Number;

return product;

}

int m\_Number;

//Rest of the class

}

class MyClass<T> where T : MyOtherClass

{

MyOtherClass Sum(T t1,T t2)

{

return t1 + t2;

}

}

**Can I Use Generic Attributes?**

You cannot define generic attributes:

//This is not possible:

class MyAttribute<T>: Attribute

{...}

However, nothing prevents you from using generics internally, inside the attribute's implementation.

**Are Generics CLS Compliant?**

Yes. With the release of .NET 2.0, generics will become part of the CLS.

**Collections**

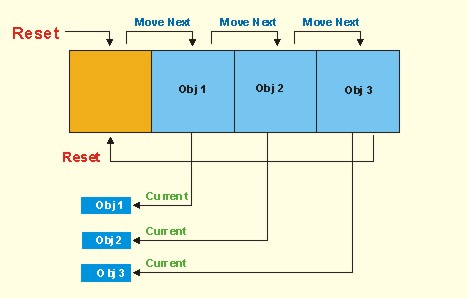
[**What is the difference between IEnumerator and IEnumerable?**](http://stackoverflow.com/questions/619564/what-is-the-difference-between-ienumerator-and-ienumerable-closed) **Why do we need to use it?**

The IEnumerable interface contains an abstract member function called GetEnumerator() and return an interface IEnumerator on any success call. This IEnumerator interface will allow us to iterate through any custom collection.

**Note:** IEnumerator interface is meant to be used as accessors and is not helpful to make any changes in the collection or elements of the collection.

Presumably, any element in a collection can be retrieved through its index property. But instead of element index, the IEnumerator provides two abstract methods and a property to pull a particular element in a collection. And they are Reset(), MoveNext() and Current.

See the figure. This is how it works.



Implementing IEnumerable enables you to get an IEnumerator for a list.

IEnumerator allows foreach style sequential access to the items in the list, using the yield keyword. *An object implementing IEnumerable allows itself to visit each of its items through an enumerator. An object implementing IEnumerator is the doing the iteration. It's looping over an enumerable object.*

Think of enumerable objects as of lists, stacks, etc.

Before foreach implementation (in Java 1.4, for example), the way to iterate a list was to get an enumerator from the list, then ask it for the "next" item in the list, for as long as the value returned as the next item is not null. Foreach simply does that implicitly as a language feature, in the same way that lock() implements the Monitor class behind the scenes.

*Example:*

Inheriting from IEnumerable means your class returns an IEnumerator object:

public class People : IEnumerable

{

IEnumerator IEnumerable.GetEnumerator()

{

// return a PeopleEnumerator

}

}

Inheriting from IEnumerator means your class returns the methods and properties for iteration:

public class PeopleEnumerator : IEnumerator

{

public void Reset()...

public bool MoveNext()...

public object Current...

}

That's the difference anyway.

**Give an example of implementing IEnumerable and IEnumerator.**

Working with a foreach loop is the primary reason to implement the IEnumerable and IEnumerator interfaces. You’ll want one of each of these to work with the loop.

I am going to do an example DateRange class which will implement IEnumerable<DateTime> and will allow us to iterate through a non-existent collection of DateTime objects.

*Note: I am aware of the fact that I could achieve the same result with a for loop. I find the foreach loop more readable.*

First we need to create a basic DateRange class. A range can be defined as a StartDate and an EndDate, so I’ll start there.

public class DateRange

{

public DateRange(DateTime startDate, DateTime endDate)

{

StartDate = startDate;

EndDate = endDate;

}

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

}

So this DateRange could be useful on its own, but we want to be able to iterate this collection using a foreach. So to start we need to implement the IEnumerable<DateTime> interface.

public class DateRange : IEnumerable<DateTime>

{

public DateRange(DateTime startDate, DateTime endDate)

{

StartDate = startDate;

EndDate = endDate;

}

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

IEnumerator IEnumerable.GetEnumerator()

{

return GetEnumerator();

}

public IEnumerator<DateTime> GetEnumerator()

{

return new DateRangeEnumerator(this);

}

}

Notice here that we now need to get the IEnumerator<DateTime> object in the GetEnumerator() method. I jumped the gun a bit and I’ve called a class that doesn’t exist yet. I’ll make another class and implement the required methods for the IEnumerator interface.

public class DateRangeEnumerator : IEnumerator<DateTime>

{

private int \_index = -1;

private readonly DateRange \_dateRange;

public DateRangeEnumerator(DateRange dateRange)

{

\_dateRange = dateRange;

}

public void Dispose()

{

}

public bool MoveNext()

{

\_index++;

if (\_index > (\_dateRange.EndDate - \_dateRange.StartDate).Days)

return false;

return true;

}

public void Reset()

{

\_index = -1;

}

public DateTime Current

{

get { return \_dateRange.StartDate.AddDays(\_index); }

}

object IEnumerator.Current

{

get { return Current; }

}

}

These are the handful of methods we implement for the IEnumerator<DateTime> interface. These are all about moving to the next object and getting the current object. Resetting and Disposal of the object are less important, so make sure you read MoveNext and Current.

Keep in mind here that I could have used a collection for this, but I didn’t because I don’t need one. The calculation to get the items was easy enough.

var dateRange = new DateRange(DateTime.Today.AddDays(-6), DateTime.Today);

foreach (DateTime date in dateRange)

{

Console.WriteLine(date.ToShortDateString());

}

Output:

10/20/2009   
10/21/2009   
10/22/2009   
10/23/2009   
10/24/2009   
10/25/2009   
10/26/2009

**[IEnumerable<T> provides two GetEnumerator methods - what is the difference between them?](http://stackoverflow.com/questions/560678/ienumerablet-provides-two-getenumerator-methods-what-is-the-difference-betwee)**

If you are implementing the IEnumerable<T> generic interface, you will pretty much always have to use the generic GetEnumerator method - unless you cast your object explicitly to (non-generic) IEnumerable.

The reason is backwards compatability with .NET 1.0/1.1 which didn't support generics.

**[Why was GetEnumerator () stored in a separate interface from IEnumerator?](http://stackoverflow.com/questions/1993140/why-was-getenumerator-stored-in-a-separate-interface-from-ienumerator)**

If all the enumeration methods were on a single interface, how could two callers enumerate the same list at the same time?

There are two interfaces because one says, "You can enumerate me," while the other says, "here's an object that keeps track of a given enumeration task."

The IEnumerable interface is a factory that creates as many IEnumerator objects as you want. How and when those enumerators get used is up to the consumer.

*IEnumerable implies that the object is a collection or source of data which can be iterated over in a linear fashion. IEnumerator is the interface for the actual implementation which performs the iteration.*

[**Is Yield Return == IEnumerable and IEnumerator?**](http://stackoverflow.com/questions/288808/is-yield-return-ienumerable-ienumerator) **Or is yield return a shortcut for implementing IEnumerable and IEnumerator? Or What is an iterator?**

Yes.

An iterator is a method, [get](http://msdn.microsoft.com/en-us/library/ms228503(VS.80).aspx) accessor or operator that enables you to support [foreach](http://msdn.microsoft.com/en-us/library/ttw7t8t6(VS.80).aspx) iteration in a [class](http://msdn.microsoft.com/en-us/library/0b0thckt(VS.80).aspx) or [struct](http://msdn.microsoft.com/en-us/library/ah19swz4(VS.80).aspx) without having to implement the entire [IEnumerable](http://msdn.microsoft.com/en-us/library/system.collections.ienumerable(VS.80).aspx) interface. Instead, you provide just an iterator, which simply traverses the data structures in your class. When the compiler detects your iterator, it will automatically generate the Current, MoveNext and Dispose methods of the **IEnumerable** or [IEnumerable](http://msdn.microsoft.com/en-us/library/9eekhta0(VS.80).aspx)<T> interface.

* **Iterators Overview**
* An iterator is a section of code that returns an ordered sequence of values of the same type.
* An iterator can be used as the body of a method, an operator, or a **get** accessor.
* The iterator code uses the **yield return** statement to return each element in turn. **yield break** ends the iteration.
* Multiple iterators can be implemented on a class. Each iterator must have a unique name just like any class member, and can be invoked by client code in a **foreach** statement as follows: foreach(int x in SampleClass.Iterator2){}
* The return type of an iterator must be **IEnumerable**, [IEnumerator](http://msdn.microsoft.com/en-us/library/system.collections.ienumerator(VS.80).aspx), **IEnumerable**<T>, or [IEnumerator](http://msdn.microsoft.com/en-us/library/78dfe2yb(VS.80).aspx)<T>.

The [yield](http://msdn.microsoft.com/en-us/library/9k7k7cf0(VS.80).aspx) keyword is used to specify the value, or values, returned. When the **yield return** statement is reached, the current location is stored. Execution is restarted from this location the next time the iterator is called.

Iterators are especially useful with collection classes, providing an easy way to iterate non-trivial data structures such as binary trees.

*Example:*

In this example, the class DaysOfTheWeek is a simple collection class that stores the days of the week as strings. After each iteration of a **foreach** loop, the next string in the collection is returned.

public class DaysOfTheWeek : System.Collections.IEnumerable

{

string[] m\_Days = { "Sun", "Mon", "Tue", "Wed", "Thr", "Fri", "Sat" };

public System.Collections.IEnumerator GetEnumerator()

{

for (int i = 0; i < m\_Days.Length; i++)

{

yield return m\_Days[i];

}

}

}

class TestDaysOfTheWeek

{

static void Main()

{

// Create an instance of the collection class

DaysOfTheWeek week = new DaysOfTheWeek();

// Iterate with foreach

foreach (string day in week)

{

System.Console.Write(day + " ");

}

}

}

**Why do we need custom collection and custom enumeration by implementing IEnumerable and IEnumerator ( I Know what IEnumerable and IEnumerator does and how to implement them).The Question is as anything can be done with collections like Dictionary, Hashtable, List, do we use custom enumeration to make non-enumerable items to enumerable collections?**

The question has two parts that are related but not the same.. To answer second part first, now that we have the yield syntax, and the plethora (I love that word!) of built-in collection class options that you can derive from, you should very very rarely have to implement IEnumerator or IEnumerable yourself anymore. So, in the great majority of cases, a custom collection class should not implement IEnumerable and IEnumerator itself, it should derive from one of the existing built-in collection classes that already do that for you.

The cases where you might need to create a custom collection are:

* In order to encapsulate logic/functionality that you wish to reuse in multiple places in client code, in a single place instead of having to repeat it everywhere. Say you have a need to represent collections of Invoices. List<Invoice> will do it, but if you need only the overdue Invoices in two or three places in one module, and only the invoices for a a specified state in another module, and only a specific customer's invoices in a billing moodule, and let's say you need to find and extract a specific invoice by it's Invoice Number in several places in code. Well if all you have is List<Invoice> all the functionality to do what I described would have to be repeated in every place in client code where you wanted to perform those functions... If you encapsulate this functionality in a custom collection class, you only write it once, and maintain it in one and only place, and access it from anywhere through the collection class methods on the instance...

public Invoices: List<Invoice>

{

// extra functionality as required

}

* When you are trying to implement a special data structure that does not yet exist in the Framework. Graphs, trees, and other special data structures come to mind. They can be very useful in many situations.
* When you want a variant of an existing collection type, you must also create a custom collection. It could be that you want the collection to host only certain type(s) and that the specific implementation of this collection is dependent on these types.
* You may need a custom collection for performance; if you have special performance needs that cannot be addressed with one of the provided general-purpose collections, writing a custom collection may be your only option. For example, you may need a list where you can insert items into the middle very quickly while at the same time you can find items just as quickly without having to iterate over the list.

**When to Use a Collection?**

You should use a collection when at least one of following statements holds true:

* Individual elements serve similar purposes and are of equal importance.
* The number of elements is unknown or is not fixed at compile time.
* You need to support iteration over all elements.
* You need to support sorting of the elements.
* You need to expose the elements from a library where a consumer will expect a collection type.

**Which collection would you use when you want additions, removals, and lookups to be very quick, and when you are not concerned about the order of the items in the collections?**

System.Collections.Generic.Dictionary<TKey, TValue> (or if you’re using the .NET Framework 1.x, a Hashtable). The three basic operations (Add, Remove, and Contains) all operate quickly even if the collection contains millions of items. On the other hand, with a List<T> (or ArrayList in the .NET Framework 1.x), inserting and removing items can take a variable amount of time. (*Both List<T> and ArrayList store items in an underlying array that maintains order. Adding items may require existing items in the underlying array be moved to make room. Adding items at the end will not require any moves and will be very quick).*

**If your usage pattern requires few deletions and mostly additions, and if it is important for you to keep the collection in order, which collection would you choose?**

You may choose a List<T>. Lookup could be slow (since the underlying array will need to be traversed while searching for the target item), but you are guaranteed that your collection maintains a specific order. Alternatively, you can choose Queue<T> to implement a first-in-first-out (FIFO) order or a Stack<T> to implement a last-in-first-out (LIFO) order. While both Queue<T> and Stack<T> support enumeration of all items in the collection, the former only supports insertion at the end and removal from the beginning, while the latter only supports insertion and removal from the beginning.

**Which collection can potentially help you improve performance in scenarios where you need to maintain order yet still achieve fast inserts?**

LinkedList<T> collection. Unlike List<T>, LinkedList<T> is implemented as a chain of dynamically allocated objects. In comparison to List<T>, inserting an object in the middle only requires updating two connections and adding the new item. The downside of a linked list from a performance point of view is increased activity by the garbage collector as it has to traverse the entire list to make sure objects were not disposed of. Additionally, performance problems can arise with large linked lists due to the overhead associated with each node as well as where in memory each node lives. And while the actual act of inserting an item into a LinkedList<T> is much faster than doing so into a List<T>, finding the particular location at which you want to insert a new value still requires traversing the list to find the correct location.

**For scenarios where you need fast insertions and lookups, a Dictionary<TKey, TValue> may be most appropriate. However, lookup is only fast in the average case, and certain datasets could cause that performance to degrade quickly, which collection best suits in such cases?**

A different implementation, SortedDictionary<TKey,TValue>, uses a balanced tree implementation as the underlying data store; this provides relatively fast lookups and maintains items in a sorted order, but insertions will most likely be slower (and will vary based on the number of items in the collection). Alternatively, you can also use SortedList<Tkey,TValue>, which uses two separate arrays to maintain the keys and the values separately and to maintain them both in order (in the worse case you would have to shift all the keys and values).

**How do we create a C# .NET strongly typed custom collection without using generics (ex. in .net 1.0/1.1)?**

The following sample code shows how to implement a strongly typed custom collection using C# and .NET v1.1.

using System;  
using System.Collections;

namespace My.Project.ORM.Collections {  
 public class ProductCollection : CollectionBase {  
  public ProductCollection() {  
  }

  public int Add(Product item) {  
   return List.Add(item);  
  }

  public void Insert(int index, Product item) {  
   List.Insert(index, item);  
  }

  public void Remove(Product item) {  
   List.Remove(item);  
  }

  public bool Contains(Product item) {  
   return List.Contains(item);  
  }

  public int IndexOf(Product item) {  
   return List.IndexOf(item);  
  }

  public void CopyTo(Product[] array, int index) {  
   List.CopyTo(array, index);  
  }

//A useful addition to this collection class is a FindByID() method for finding specific items in the //collection by passing an item ID, as follows:

public Product FindByID(int ID) {  
 foreach(Product item in List) {  
  if(item.ID == ID) {  
   return item;  
  }  
 }  
 return null;  
}

  public Product this[int index] {  
   get {  
    return (Product)List[index];  
   }  
   set {  
    List[index] = value;  
   }  
  }  
 }  
}

**How do we create custom collection in .NET 2.0 and 3.5?**

*If you decide you want to write a custom collection, you should first look into extending an existing collection type. Start by looking at the System.Collections.ObjectModel namespace.* These collections already implement the most needed interfaces and give you the baseline functionality you expect. It’s easy to add methods to inherited classes that enable your specific needs.

As an example, let’s say you want to implement a simple collection of System.Double values, but you also want to support an Add(string) method that uses Double.TryParse, only adding the result if the value can be parsed to a Double successfully. The *System.Collections.ObjectModel.Collection<T> class implements all of the standard collection interfaces, and your custom collection type can simply derive from Collection<Double> in order to provide the additional overload of Add:*

class DoubleCollection : Collection<double>

{

public void Add(string st)

{

Double d;

if (Double.TryParse(st, out d)) base.Add(d);

else throw new ArgumentException(

“Cannot parse string to a double. “ +

“Item was not added to collection.”);

}

}

The .NET Framework 3.5 with its introduction of extension methods provides an additional mechanism for creating methods that extend existing types. For example, you could rewrite this method as an extension method in C# 3.0 as follows:

class CollectionExtensions

{

public static void Add(this Collection<Double> c, string st)

{

Double d;

if (Double.TryParse(st, out d)) c.Add(d);

else throw new ArgumentException(

“Cannot parse string to a double. “ +

“Item was not added to collection.”);

}

}

Given a Collection<Double> that holds named values, this static method can be used to add a new value to the collection using traditional syntax:

CollectionExtensions.Add(values, “3.14”);

However, as this is an extension method (evident from the "this" keyword attributing the first parameter to the static Add method), you can rewrite this as follows:

values.Add(“3.14”);

This is the exact same method call you would write if you’d extended Collection<Double> with a new type, but you can now use this new Add method with any instance of Collection<Double> (or any type that derives from it), rather than just with your custom type. Alternatively, you can start building your own collection without inheriting from any of the existing ones. In that case you can take advantage of the appropriate interfaces that are provided in the .NET Framework.

**How do we create custom collection in .net from scratch?**

If you do decide that you want to write your own custom collection from scratch, you should implement all of the appropriate interfaces provided with the .NET Framework. These interfaces expose common ways to interact with collections that most people find very intuitive and easy to use. Here are the various interfaces you can choose from.

* IEnumerable
* IEnumerable<T>
* IEnumerator
* IEnumerator<T>
* ICollection
* ICollection<T>
* IList and IList<T>
* IDictionary and IDictionary<TKey, TValue>

**How to provide default sort order to an object?**

The role of **IComparable** is to provide a method of comparing two objects of a particular type. This is necessary if you want to provide any ordering capability for your object. Think of IComparable as providing a default sort order for your objects. For example, if you have an array of objects of your type, and you call the Sort method on that array, IComparable provides the comparison of objects during the sort. When you implement the IComparable interface, you must implement the CompareTo method, as follows:

// Implement IComparable CompareTo method - provide default sort order.

int IComparable.CompareTo(object obj)

{

car c=(car)obj;

return String.Compare(this.make,c.make);

}

The comparison in the method is different depending on the data type of the value that is being compared. String.Compare is used in this example because the property that is chosen for the comparison is a string.

**How to provide ordering of your class on several fields or properties, ascending and descending order on the same field, or both?**

The role of **IComparer** is to provide additional comparison mechanisms. For example, you may want to provide ordering of your class on several fields or properties, ascending and descending order on the same field, or both.   
  
Using IComparer is a two-step process. First, declare a class that implements IComparer, and then implement the Compare method:

private class sortYearAscendingHelper : IComparer

{

int IComparer.Compare(object a, object b)

{

car c1=(car)a;

car c2=(car)b;

if (c1.year > c2.year)

return 1;

if (c1.year < c2.year)

return -1;

else

return 0;

}

}

Note that the IComparer.Compare method requires a tertiary comparison. 1, 0, or -1 is returned depending on whether one value is greater than, equal to, or less than the other. The sort order (ascending or descending) can be changed by switching the logical operators in this method.  
  
The second step is to declare a method that returns an instance of your IComparer object:

public static IComparer sortYearAscending()

{

return (IComparer) new sortYearAscendingHelper();

}

In this example, the object is used as the second argument when you call the overloaded Array.Sort method that accepts IComparer. The use of IComparer is not limited to arrays. It is accepted as an argument in a number of different collection and control classes.

**Give an example to demonstrate use of IComparer and IComparable in a class.**

The following example demonstrates the use of these interfaces. To demonstrate **IComparer** and **IComparable**, a class named **car** is created. The **car** object has the **make** and **year** properties. An ascending sort for the **make** field is enabled through the **IComparable** interface, and a descending sort on the **make** field is enabled through the **IComparer** interface. Both ascending and descending sorts are provided for the **year** property through the use of **IComparer**.

using System;

using System.Collections;

namespace ConsoleEnum

{

public class car : IComparable

{

// Beginning of nested classes.

// Nested class to do ascending sort on year property.

private class sortYearAscendingHelper: IComparer

{

int IComparer.Compare(object a, object b)

{

car c1=(car)a;

car c2=(car)b;

if (c1.year > c2.year)

return 1;

if (c1.year < c2.year)

return -1;

else

return 0;

}

}

// Nested class to do descending sort on year property.

private class sortYearDescendingHelper: IComparer

{

int IComparer.Compare(object a, object b)

{

car c1=(car)a;

car c2=(car)b;

if (c1.year < c2.year)

return 1;

if (c1.year > c2.year)

return -1;

else

return 0;

}

}

// Nested class to do descending sort on make property.

private class sortMakeDescendingHelper: IComparer

{

int IComparer.Compare(object a, object b)

{

car c1=(car)a;

car c2=(car)b;

return String.Compare(c2.make,c1.make);

}

}

// End of nested classes.

private int year;

private string make;

public car(string Make,int Year)

{

make=Make;

year=Year;

}

public int Year

{

get {return year;}

set {year=value;}

}

public string Make

{

get {return make;}

set {make=value;}

}

// Implement IComparable CompareTo to provide default sort order.

int IComparable.CompareTo(object obj)

{

car c=(car)obj;

return String.Compare(this.make,c.make);

}

// Method to return IComparer object for sort helper.

public static IComparer sortYearAscending()

{

return (IComparer) new sortYearAscendingHelper();

}

// Method to return IComparer object for sort helper.

public static IComparer sortYearDescending()

{

return (IComparer) new sortYearDescendingHelper();

}

// Method to return IComparer object for sort helper.

public static IComparer sortMakeDescending()

{

return (IComparer) new sortMakeDescendingHelper();

}

}

}

Below code shows how to use car class.

using System;

namespace ConsoleEnum

{

class host

{

[STAThread]

static void Main(string[] args)

{

// Create an arary of car objects.

car[] arrayOfCars= new car[6]

{

new car("Ford",1992),

new car("Fiat",1988),

new car("Buick",1932),

new car("Ford",1932),

new car("Dodge",1999),

new car("Honda",1977)

};

// Write out a header for the output.

Console.WriteLine("Array - Unsorted\n");

foreach(car c in arrayOfCars)

Console.WriteLine(c.Make + "\t\t" + c.Year);

// Demo IComparable by sorting array with "default" sort order.

Array.Sort(arrayOfCars);

Console.WriteLine("\nArray - Sorted by Make (Ascending - IComparable)\n");

foreach(car c in arrayOfCars)

Console.WriteLine(c.Make + "\t\t" + c.Year);

// Demo ascending sort of numeric value with IComparer.

Array.Sort(arrayOfCars,car.sortYearAscending());

Console.WriteLine("\nArray - Sorted by Year (Ascending - IComparer)\n");

foreach(car c in arrayOfCars)

Console.WriteLine(c.Make + "\t\t" + c.Year);

// Demo descending sort of string value with IComparer.

Array.Sort(arrayOfCars,car.sortMakeDescending());

Console.WriteLine("\nArray - Sorted by Make (Descending - IComparer)\n");

foreach(car c in arrayOfCars)

Console.WriteLine(c.Make + "\t\t" + c.Year);

// Demo descending sort of numeric value using IComparer.

Array.Sort(arrayOfCars,car.sortYearDescending());

Console.WriteLine("\nArray - Sorted by Year (Descending - IComparer)\n");

foreach(car c in arrayOfCars)

Console.WriteLine(c.Make + "\t\t" + c.Year);

Console.ReadLine();

}

}

}

**How to sort a list of strings or integers?**

By just calling the List.Sort() method.

**Is there generic version of IComparable and IComparer interfaces?**

Yes. IComparable<T> and IComparer<T> interface helps to sort list of objects on custom classes easily.

**For example we have a simple Employee class with two fields, Name and Salary.**

**class Employee**

**{**

**public string Name { get; set; }**

**public int Salary { get; set; }**

**}**

**Now create a List of Employees and call Sort() method of a List.**

***// Use Collection Initializers( C# 3.0 ) to initialize the List***

**List<Employee> empList = new List<Employee>()**

**{ new Employee { Name = "a", Salary = 14000 },**

**new Employee { Name = "b", Salary = 13000 }**

**};**

**empList.Sort();**

**What happens when above code is executed?**

An Exception is thrown which says we need to implement IComparable<> interface. List.Sort() sorts any class that implements IComparable<> interface.

**ArrayList Concept in .Net**

Provides a collection similar to an array, but that grows dynamically as  
the number of elements change.  
  
Example  
  
static void Main()  
{  
 ArrayList list = new ArrayList();  
 list.Add(11);  
 list.Add(22);  
 list.Add(33);  
 foreach(int num in list)  
 {  
 Console.WriteLine(num);  
 }  
}  
  
Output

11  
22  
33

**Stack Concept in .Net**

A collection that works on the Last In First Out (LIFO) principle,  
i.e., the last item inserted is the first item removed from the collection.   
Push - To add element and   
Pop - To Remove element  
  
Example  
  
using System;  
using System.Collections;  
  
class Test  
{  
 static void Main()  
 {  
 Stack stack = new Stack();  
 stack.Push(2);  
 stack.Push(4);  
 stack.Push(6);  
   
 while(stack.Count != 0)  
 {  
 Console.WriteLine(stack.Pop());  
 }  
 }  
}  
  
**Output**

6  
4  
2

**Queue Concept in .Net**

A collection that works on the First In First Out (FIFO) principle, i.e.,  
the first item inserted is the first item removed from the collection.   
Enqueue - To add element and Dequeue – To Remove element

Example:  
  
static void Main()  
{  
 Queue queue = new Queue();  
 queue.Enqueue(2);  
 queue.Enqueue(4);  
 queue.Enqueue(6);  
   
 while(queue.Count != 0)  
 {  
 Console.WriteLine(queue.Dequeue());  
 }  
}  
  
Output

2  
4  
6

**Hashtable Concept in .Net**

Provides a collection of key-value pairs that are organized based on the hash code of the key.  
  
Example:  
static void Main()  
{  
 Hashtable ht = new Hashtable(20);  
 ht.Add("ht01", "DotNetGuts");  
 ht.Add("ht02", "EasyTutor.2ya.com");  
 ht.Add("ht03", "DailyFreeCode.com");  
   
 Console.WriteLine("Printing Keys...");  
 foreach(string key in ht.Keys)  
 {  
 Console.WriteLine(key);  
 }  
   
 Console.WriteLine("\nPrinting Values...");  
 foreach(string Value in ht.Values)  
 {  
 Console.WriteLine(Value);  
 }  
   
 Console.WriteLine("Size of Hashtable is {0}", ht.Count);  
   
 Console.WriteLine(ht.ContainsKey("ht01"));  
 Console.WriteLine(ht.ContainsValue("DailyFreeCode.com"));  
   
 Console.WriteLine("\nRemoving element with key = ht02");  
 ht.Remove("ht02");  
   
 Console.WriteLine("Size of Hashtable is {0}", ht.Count);  
}  
  
Output

Printing Keys...  
ht01  
ht02  
ht03  
  
Printing Values...  
DotNetGuts  
EasyTutor.2ya.com  
DailyFreeCode.com

Size of Hashtable is 3

True

True

Removing element with key = ht02

Size of Hashtable is 2

**Why are hashtable lookups so slow with struct keys?**

When struct objects are used as hashtable keys, the lookup operation for the hashtable performs horribly. The reason for the poor performance lies in the GetHashCode method which performs the lookup internally. When a struct contains only simple value types—int, short, *etc.*—, the GetHashCode algorithm which computes the hashes causes most of them to be *stored in the same bucket*. For example, assume a hashtable creates five buckets.

bucket-1 - value-a, value-b, value-c, …, value-n

bucket-2 - (empty)

bucket-3 - (empty)

bucket-4 - (empty)

bucket-5 - (empty)

Probably, all of the keys will be put into a single bucket. Therefore, when a lookup is performed, the .NET runtime will be forced to traverse the entire contents of that bucket to find the value. So, instead of the lookup operation being O(1), the average lookup case becomes O(n).

To alleviate this problem, *override the GetHashCode method*; so, the runtime can access the keys efficiently.

One way to address the problem is to create a key string by merging all the value types from a struct. Each type is seperated by a special delimiter character. Since the struct is a lookup criteria, it is certain that all struct values will be different; therefore, the generated string is guaranteed to be unique. And, since the string is derived from System.Object, the generated key string has a GetHashCode method which can be overridden.

Here is an example of overriding the GetHashCode method:

struct

{

int value-a;

short value-b;

public struct(int value-a, short value-b)

{

this.value-a = value-a;

this.value-b = value-b;

}

public override int GetHashCode()

{

string hash = value-a.ToString() + ":" + value-b.ToString();

return hash.GetHashCode();

}

}

Now, the generated hashcode is certain to be unique causing hash keys to be distributed across all buckets; so, lookup performance will be greatly improved.

**What’s the .NET collection class that allows an element to be accessed using a unique key?**HashTable.

**What is Difference between Array and Hash Table?**  
In an array we make collection of same type of data. But in hash table we can collect different type of objects.

In array we can access the element using index. we have to access element one by one. If we do'nt remember index then we can’t acess the element. On the other hand in hash table

we can acess the element by using key. Also in hash table we can take the key of the element a string also.

**What is the difference between Dictionary and Hashtable? How to decide which one to use?**

The Hashtable class and the Dictionary generic class implement the IDictionary interface. The **Dictionary** generic class also implements the IDictionary generic interface. Therefore, each element in these collections is a key-and-value pair.

A **Hashtable** object consists of buckets that contain the elements of the collection. A bucket is a virtual subgroup of elements within the **Hashtable**, which makes searching and retrieving easier and faster than in most collections. Each bucket is associated with a hash code, generated using a hash function and based on the key of the element.

*A hash function is an algorithm that returns a numeric hash code based on a key.* The key is the value of some property of the object being stored. *A hash function must always return the same hash code for the same key.* It is possible for a hash function to generate the same hash code for two different keys, but a hash function that generates a unique hash code for each unique key results in better performance when retrieving elements from the hash table.

*Each object that is used as an element in a* ***Hashtable*** *must be able to generate a hash code for itself using an implementation of the* [*GetHashCode*](http://msdn.microsoft.com/en-us/library/system.object.gethashcode(VS.80).aspx) *method.* However, you can also specify a hash function for all elements in a **Hashtable** by using a **Hashtable** constructor that accepts an IHashCodeProvider implementation as one of its parameters.

*When an object is added to a* ***Hashtable****, it is stored in the bucket that is associated with the hash code that matches the object's hash code. When a value is being searched for in the* ***Hashtable****, the hash code is generated for that value, and the bucket associated with that hash code is searched.*

For example, a hash function for a string might take the ASCII codes of each character in the string and add them together to generate a hash code. The string "picnic" would have a hash code that is different from the hash code for the string "basket"; therefore, the strings "picnic" and "basket" would be in different buckets. In contrast, "stressed" and "desserts" would have the same hash code and would be in the same bucket.

The **Dictionary** class has the same functionality as the **Hashtable** class. A **Dictionary** of a specific type (other than Object) has better performance than a **Hashtable** for value types because the elements of **Hashtable** are of type **Object** and, therefore, boxing and unboxing typically occur if storing or retrieving a value type.

**SortedList Concept in .Net**

Provides a collection of key-value pairs where the items are sorted according to the key. The items are accessible by both the keys and the index.

Example:

static void Main()  
{  
SortedList sl = new SortedList();  
sl.Add(18, "Java");  
sl.Add(5, "C#");  
sl.Add(11, "VB.Net");  
sl.Add(1, "C++.Net");  
  
Console.WriteLine("The items in the sorted order are...");  
Console.WriteLine("\t Key \t\t Value");  
Console.WriteLine("\t === \t\t =====");  
for(int i=0; i<sl.Count; i++)  
{  
Console.WriteLine("\t {0} \t\t {1}", sl.GetKey(i),  
sl.GetByIndex(i));  
}  
}  
  
Output

Key Value

==  
1 C++.Net  
5 C#  
11 VB.Net  
18 Java

**What class is underneath the SortedList class?**A sorted HashTable.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**CLR facilities**

**Exceptions**

**Automatic Memory Management**

**CLR Hosting and Appdomains**

**Thread synchronization**

**Exceptions**

**What is an Exception?**

*An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error.*

Exceptions are the standard mechanism for reporting errors. Applications and libraries should not use return codes to communicate errors. *The use of exceptions adds to a consistent framework design and allows error reporting from members, such as constructors, that cannot have a return type. Exceptions also allow programs to handle the error or terminate as appropriate.* The default behavior is to terminate an application if it does not handle a thrown exception.

**What is difference between Exception and Error?**

Exception is a run time error when an abnormal condition occurs.

Error is the compile time error which is only fixed by a programmer.

**Throw some light on error and exception handling. What are the differences?**

Error handling refers to the anticipation, detection, and resolution of programming, application, and communications errors. Specialized programs, called error handlers, are available for some applications. The best programs of this type forestall errors if possible, recover from them when they occur without terminating the application, or (if all else fails) gracefully terminate an affected application and save the error information to a log file.

In programming, a development error is one that can be prevented. Such an error can occur in [syntax](http://searchcio-midmarket.techtarget.com/sDefinition/0,,sid183_gci213081,00.html) or logic. Syntax errors, which are typographical mistakes or improper use of special characters, are handled by rigorous proofreading. Logic errors, also called [bug](http://searchsoftwarequality.techtarget.com/sDefinition/0,,sid92_gci211714,00.html)s, occur when [execute](http://searchcio-midmarket.techtarget.com/sDefinition/0,,sid183_gci212086,00.html)d [code](http://whatis.techtarget.com/definition/0,,sid9_gci213934,00.html) does not produce the expected or desired result. Logic errors are best handled by meticulous program [debugging](http://searchsoftwarequality.techtarget.com/sDefinition/0,,sid92_gci211915,00.html). This can be an ongoing process that involves, in addition to the traditional debugging routine, [beta](http://searchcio-midmarket.techtarget.com/sDefinition/0,,sid183_gci211654,00.html) testing prior to official release and customer feedback after official release.

A run-time error takes place during the execution of a program, and usually happens because of adverse system [parameter](http://whatis.techtarget.com/definition/0,,sid9_gci213939,00.html)s or invalid input [data](http://searchdatamanagement.techtarget.com/sDefinition/0,,sid91_gci211894,00.html). An example is the lack of sufficient [memory](http://searchmobilecomputing.techtarget.com/sDefinition/0,,sid40_gci212546,00.html) to run an application or a memory conflict with another program. On the Internet, run-time errors can result from electrical [noise](http://whatis.techtarget.com/definition/0,,sid9_gci212667,00.html), various forms of [malware](http://searchmidmarketsecurity.techtarget.com/sDefinition/0,,sid198_gci762187,00.html) or an exceptionally heavy demand on a server. Run-time errors can be resolved, or their impact minimized, by the use of error handler programs, by vigilance on the part of network and server administrators, and by reasonable security.

**What is the purpose of the finally block?**

The code in finally block is guaranteed to run, irrespective of whether an error occurs or not. Critical portions of code, for example release of file handles or database connections, should be placed in the finally block.

**Will the finally block get executed if an exception has not occurred?**Yes.

**Can I use exceptions in C#?**

Yes, in fact exceptions are the recommended error-handling mechanism in C# (and in .NET in general). Most of the .NET framework classes use exceptions to signal errors.

**Why is it a bad idea to throw your own exceptions?**

Well, if at that point you know that an error has occurred, then why not write the proper code to handle that error instead of passing a new Exception object to the catch block? Throwing your own exceptions signifies some design flaws in the project.

**What’s the C# syntax to catch any possible exception?**

A catch block that catches the exception of type System.Exception. You can also omit the parameter data type in this case and just write catch {}

**What’s the C# syntax to catch any possible exception?**A catch block that catches the exception of type System.Exception.  You can also omit the parameter data type in this case and just write catch {}. 

**Can multiple catch blocks be executed for a single try statement?**No.  Once the proper catch block processed, control is transferred to the finally block (if there are any). 

**Does the System.Exception class have any cool features?**

Yes - the feature which stands out is the StackTrace property. This provides a call stack which records where the exception was thrown from. For example, the following code:

using System;

class CApp

{

public static void Main()

{

try {

f();

}

catch( Exception e )

{

Console.WriteLine( "System.Exception stack trace = \n{0}", e.StackTrace );

}

}

static void f()

{

throw new Exception( "f went pear-shaped" );

}

}

produces this output:

System.Exception stack trace = at CApp.f() at CApp.Main()

Note, however, that this stack trace was produced from a debug build. A release build may optimise away some of the method calls which could mean that the call stack isn't quite what you expect.

**When should I throw an exception?**

This is the subject of some debate, and is partly a matter of taste. However, *it is accepted by many that exceptions should be thrown only when an 'unexpected' error occurs.* How do you decide if an error is expected or unexpected? This is a judgement call, but a straightforward example of an expected error is failing to read from a file because the seek pointer is at the end of the file, whereas an example of an unexpected error is failing to allocate memory from the heap.

**How to catch all exceptions in c# code?**

Answer:

Use try...catch block to catch exceptions.

To catch all exception use base Exception class "Exception"

try

{

}

catch(Exception)

{

}

**Can there be try block without catch block?**

A try block must be followed by either a catch or finally block.

**What should be the order of multiple catch blocks?**

A try block can throw multiple exceptions, which can be handled by using multiple catch blocks. Exceptions are implemented as classes, and if your catch blocks are looking for a base-class exception before the specific inherited exception, you may never catch the specific exception. The specific or inherited exception will be masked by the base class. *Remember that more specialized catch block should come before a generalized one. Otherwise the compiler will show a compilation error.*

**What is the difference between System exceptions and Application exceptions?**

All exception derives from Exception Base class. Exceptions can be generated programmatically or can be generated by system. Application Exception serves as the base class for all application-specific exception classes. It derives from Exception but does not provide any extended functionality. You should derive your custom application exceptions from Application Exception.

**List out some of the exception classes in C#?**  
The following exceptions are thrown by certain C# operations:  
*System.OutOfMemoryException* Thrown when an attempt to allocate memory (via new) fails.  
*System.StackOverflowException* Thrown when the execution stack is exhausted by having too many pending method calls; typically indicative of very deep or unbounded recursion.  
*System.NullReferenceException* Thrown when a null reference is used in a way that causes the referenced object to be required.  
*System.TypeInitializationException* Thrown when a static constructor throws an exception, and no catch clauses exists to catch in.  
*System.InvalidCastException* Thrown when an explicit conversion from a base type or interface to a derived types fails at run time.  
*System.ArrayTypeMismatchException* Thrown when a store into an array fails because the actual type of the stored element is incompatible with the actual type of the array.  
System.IndexOutOfRangeException Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array.  
*System.MulticastNotSupportedException* Thrown when an attempt to combine two non-null delegates fails, because the delegate type does not have a void return type.  
*System.ArithmeticException* A base class for exceptions that occur during arithmetic operations, such as DivideByZeroException and OverflowException.  
*System.DivideByZeroException* Thrown when an attempt to divide an integral value by zero occurs.  
*System.OverflowException* Thrown when an arithmetic operation in a checked context overflows.

**If I return out of a try/finally in C#, does the code in the finally-clause run?**

Yes. The code in the finally always runs. If you return out of the try block, or even if you do a “goto” out of the try, the finally block always runs:

using System;

class main

{

public static void Main()

{

try

{

Console.WriteLine("In Try block");

return;

}

finally

{

Console.WriteLine("In Finally block");

}

}

}

Both “In Try block” and “In Finally block” will be displayed. Whether the return is in the try block or after the try-finally block, performance is not affected either way. The compiler treats it as if the return were outside the try block anyway. If it’s a return without an expression (as it is above), the IL emitted is identical whether the return is inside or outside of the try. If the return has an expression, there’s an extra store/load of the value of the expression (since it has to be computed within the try block).

**If we write return statement in finally block will it works fine or throws any error?**

NO. We cannot have return in finally block. We get the following error.

“Control cannot leave the body of a finally clause”.

try

{

First.DoSomething1();

First.DoSomething2();

}

catch (Exception)

{

throw;

}

finally { return 0; }

**Which return will get invoked when exception occurs in try block and when exception does not occur?**

**try**

**{**

**First.DoSomething1();**

**First.DoSomething2();**

**return 1;**

**}**

**catch (Exception)**

**{**

**return 2;**

**}**

**finally { }**

When exception is thrown, return 2 will get invoked and when exception is not thrown, return 1 would get invoked.

**What is the difference between: catch(Exception e){throw e;} and catch(Exception e){throw;}? Or Why re-throw exception without specifying the exception?**

Once an exception is thrown, part of the information it carries is the stack trace. The stack trace is a list of the method call hierarchy that starts with the method that throws the exception and ends with the method that catches the exception. If an exception is re-thrown by specifying the exception in the **throw** statement, the stack trace is restarted at the current method and the list of method calls between the original method that threw the exception and the current method is lost. To keep the original stack trace information with the exception, use the **throw** statement without specifying the exception.

**Explain the guidelines for handling exceptions.**

The following guidelines help ensure that your library handles exceptions appropriately.

* Do not handle errors by catching non-specific exceptions, such as System.Exception, System.SystemException, and so on, in framework code.

You can catch exceptions when the purpose of catching the exception is to re-throw or transfer the exception to a different thread. The following code example demonstrates incorrect exception handling.

public class BadExceptionHandlingExample1

{

public void DoWork()

{

// Do some work that might throw exceptions.

}

public void MethodWithBadHandler()

{

try

{

DoWork();

}

catch (Exception e)

{

// Swallow the exception and continue

// executing.

}

}

}

* Avoid handling errors by catching non-specific exceptions, such as System.Exception, System.SystemException, and so on, in application code. There are cases when handling errors in applications is acceptable, but such cases are rare.
* Do not exclude any special exceptions when catching for the purpose of transferring exceptions.

Instead of creating lists of special exceptions in your catch clauses, you should catch only those exceptions that you can legitimately handle. Exceptions that you cannot handle should not be treated as special cases special-cased in non-specific exception handlers. The following code example demonstrates incorrectly testing for special exceptions for the purposes of re-throwing them.

public class BadExceptionHandlingExample2

{

public void DoWork()

{

// Do some work that might throw exceptions.

}

public void MethodWithBadHandler()

{

try

{

DoWork();

}

catch (Exception e)

{

if (e is StackOverflowException ||

e is OutOfMemoryException)

throw;

// Handle exception and continue

// executing.

}

}

}

1. Consider catching specific exceptions when you understand why it will be thrown in a given context.

You should catch only those exceptions that you can recover from. For example, a [FileNotFoundException](http://msdn.microsoft.com/en-us/library/system.io.filenotfoundexception(VS.80).aspx) that results from an attempt to open a non-existent file can be handled by an application because it can communicate the problem to the user and allow the user to specify a different file name or create the file. A request to open a file that generates an [ExecutionEngineException](http://msdn.microsoft.com/en-us/library/system.executionengineexception(VS.80).aspx) should not be handled because the underlying cause of the exception cannot be known with any degree of certainty, and the application cannot ensure that it is safe to continue executing.

* Do not overuse catch. Exceptions should often be allowed to propagate up the call stack.

Catching exceptions that you cannot legitimately handle hides critical debugging information.

* Do use try-finally and avoid using try-catch for cleanup code. In well-written exception code, try-finally is far more common than try-catch.
* Do prefer using an empty throw when catching and re-throwing an exception. This is the best way to preserve the exception call stack.

The following code example shows a method that can throw an exception. This method is referenced in later examples.

public void DoWork(Object anObject)

{

// Do some work that might throw exceptions.

if (anObject == null)

{

throw new ArgumentNullException("anObject",

"Specify a non-null argument.");

}

// Do work with o.

}

The following code example demonstrates catching an exception and incorrectly specifying it when re-throwing the exception. This causes the stack trace to point to the re-throw as the error location, instead of pointing to the DoWork method.

public void MethodWithBadCatch(Object anObject)

{

try

{

DoWork(anObject);

}

catch (ArgumentNullException e)

{

System.Diagnostics.Debug.Write(e.Message);

// This is wrong.

throw e;

// Should be this:

// throw;

}

}

* Do not handle non-CLS-compliant exceptions (exceptions that do not derive from System.Exception) using a parameterless catch block. Languages that support exceptions that are not derived from Exception are free to handle these non-CLS compliant exceptions.

The .NET Framework version 2.0 wraps non-CLS-compliant exceptions in a class derived from [Exception](http://msdn.microsoft.com/en-us/library/system.exception(VS.80).aspx).

**What are the rules for designing custom exceptions?**

The following guidelines help ensure that your custom exceptions are correctly designed.

* Avoid deep exception hierarchies.
* Do derive exceptions from System.Exception or one of the other common base exceptions.

Note that [Catching and Throwing Standard Exception Types](http://msdn.microsoft.com/en-us/library/ms229007(VS.80).aspx) has a guideline that states that you should not derive custom exceptions from [ApplicationException](http://msdn.microsoft.com/en-us/library/system.applicationexception(VS.80).aspx).

* Do end exception class names with the Exception suffix.
* Do make exceptions serializable. An exception must be serializable to work correctly across application domain and remoting boundaries.
* Do provide (at least) the following common constructors on all exceptions. Make sure the names and types of the parameters are the same those used in the following code example.

public class NewException : BaseException, ISerializable

{

public NewException()

{

// Add implementation.

}

public NewException(string message)

{

// Add implementation.

}

public NewException(string message, Exception inner)

{

// Add implementation.

}

// This constructor is needed for serialization.

protected NewException(SerializationInfo info, StreamingContext context)

{

// Add implementation.

}

}

* Do report security-sensitive information through an override of System.Object.ToString only after demanding an appropriate permission. If the permission demand fails, return a string that does not include the security-sensitive information.
* Do store useful security-sensitive information in private exception state. Ensure that only trusted code can get the information.
* Consider providing exception properties for programmatic access to extra information (besides the message string) relevant to the exception**.**

**What happens if you have unhandled constructor exceptions?**

An unhandled exception from a constructor can occur if the body of a constructor causes an exception to be raised. This is sometimes unavoidable, although it is inconvenient. For example, if you were unable to fully initialize the object, for example because of inconsistent or corrupt state, you would have no other choice but to throw an exception. The alternative would be to pretend everything was OK by returning successfully from the constructor, which could lead to failures later on in the program’s execution. It’s better just to have the program fail as soon as you notice a problem.

The reason that this topic is problematic enough to call out is that, in the face of an exception thrown by a constructor, the caller of your constructor will have no reference to your object. Consider what the IL sequence looks like for the C# code Foo f = new Foo():

newobj instance void Foo::.ctor()

stloc.0

The newobj instruction makes a call to the constructor, leaving the result on the execution stack. stloc.0 stores it in a local slot in the execution stack. But if an exception is raised as a result of the newobj instruction (because of insufficient memory to allocate the object, or because of an unhandled exception thrown from the constructor), the result will never be placed on the stack, and the stloc.0 will never execute because the CLR exception handling behavior will take over. The result is that the variable f will be null.

This can cause some unpredictable behavior. For example, consider a type that allocates resources in its constructor but then throws an exception. Normally, types like this will support the IDisposable interface—explained fully in Chapter 5—so that these resources can be reclaimed deterministically. Such code might look like this:

using (Foo f = new Foo())

{

// Do something interesting...

}

The semantics of the using statement are such that the variable inside the parenthesis will have its Dispose method called at the end of the block, regardless of whether exit is normal or due to an exception from the block’s body. But if an exception occurs during the call to Foo’s constructor, f will never get assigned a value! The result is that Dispose will never get called, and hopefully f’s finalizer will clean up the resources sometime later on.

**Constructors do not have return type and so they cannot return error codes (of course possible with *out* or *ref* parameter, but it is not recommended to use out or ref parameter). How are errors or exceptions handled in constructors? What if the calls that you make in the constructor can actually throw exceptions? How do you let the caller know something bad happened in a constructor? Or, How to let the invoker know that the instance of the object has not been created successfully, so that the invoker can perform appropriate actions?**

There are a few ways to do robust error/exception handling in constructors

1. Do as little in the constructor has you can. Then provide an Init() function in the constructor, which does the normal initialization stuff. The user can then call this function after creating an object. The problem here is, its up to the user to actually call the Init() function. The user could potentially miss this step, making this method error prone. However, there are a lot of places where this methodology is used. You are trying to eliminate error handling in the constructor by using this method
2. Another way to do this is by putting the object in a Zombie state. This is one approach you can take especially if you do not have the option of using exceptions. When you go with this option, you will also to do provide a function that will check the state of the object after construction. The downsides to this option is that, its up to the user to do these checks and the users will need to do this every time one attempts to create an object. It’s usually always better and cleaner to throw an exception instead. Use the Zombie option as a last resort.
3. The downsides to the above methods can be reduced by making the constructor private or protected, expose a CreateInstance() public method, and do all the error handling here rather than leave it to the user. But sometimes, it’s not possible to handle all the error conditions in a generic manner and you will need to throw an exception.
4. If an exception is thrown in the constructor, the destructor will not get called. So you need to handle and clean up as much as you can before you leave the constructor. The best way to do this is using the “resource allocation is initialization” technique in C++. But the basic idea is to assign resource allocation and cleanup to other objects. Basically, you are trying to get allocation out of the way (indirect) so that you don’t have to do it explicitly. When you don’t allocate something directly, you don’t have to release it either because it will be done by the component or class who deals with it. E.g. If you need to allocate some memory or open up a file, You can use smart objects (smart pointer, auto\_ptr, smart file handlers etc..) instead of calling new or fopen directly. When you do this, and if an exception is thrown in your constructor, the smart objects will automatically release the resources it acquired, as the stack unwinds. If you do not use the “resource allocation is initialization” technique (in java and C#), the user will need to wrap the statements in try/catch block and re-throw after cleaning up the mess, something like what the finally block does in Java or C#. Although this works in theory, it’s up to the user to make this work and it also always a source of errors and bugs (esp. memory and handle leaks) and is messy.

You have to be careful when throwing exceptions **in** a **constructor**, that is, if you want clean running code. Constructors should also return valid objects but sometimes can't due to errors. The best way to deal with something of this scenario is to present clear error messages that let the user know what they failed to give or present, and second is to use the finally clause. The finally clause will give about the only way to "clean up" your code after failure, although you must be careful here also. Using the finally clause means that it will execute every time the code is run(you know this already), which means some kind of flag must be created **in** order for proper cleanup.

As you have seen, there is no “one size fits all” rule to do error/exception handling in constructors. I have listed the most commonly used methods and one of these should work most of the time.

**How can one report errors to invoker of constructor using error code?**

There is a more clean solution to the problem. Constructors, as any other methods, can return many values to the calling program via “ref” parameters! It would be natural for constructors to validate their arguments and report errors. So here is solution:  
class Levels  
{  
private double[] \_buffer;  
private int \_size;

public Levels(int xsize, ref int xerror) //here goes my return value  
{  
if (xsize < 2)  
{  
xerror = -1; //"Need at least 2 Levels"  
return;  
}  
\_size = xsize;  
\_buffer = new double[xsize];  
}  
…}  
Now, in the main program:  
public static void Main()  
{  
int error = -2; //should be set to 0 if all goes well  
Levels support = new Levels(1, ref error); //error will tell me if all is OK  
if (error != 0)  
{  
Console.WriteLine("ABEND {0}", error); //ABnormal END  
Console.ReadKey();  
Environment.ExitCode = error;  
return; //min 2 levels  
}

The downsides to this option is that, it’s up to the user to do these checks and the users will need to do this every time one attempts to create an object. It’s usually always better and cleaner to throw an exception instead. Of course this provides one solution without the need to re-throw exception. Throwing exception is always costly.

**Write code that helps in understanding program for handling Constructor exception.**

A Constructor means a method which has the same name as of the class. Constructor initializes a new object belonging to the class automatically. The program given below consists of a parameterized constructor which takes String and int as a parameters. The main reason of exception occurring in the program is   ConstructorException c = new ConstructorException("gh", -1);For removing this exception we have to passs a value which is greater than 0 instead of -1..

**ConstructorException.java**

| **public class**ConstructorException {      **private static**String name;     **private static int**age;     **public**ConstructorException(String Name, **int**Age) {         name = Name;         age = Age;         **if**(age < 0) {             **throw new**IllegalArgumentException("age out of range: " + age );         }         **if**(name == **null**) {             **throw new**IllegalArgumentException(                     "name is null");         }     }      **public static void**main(String[] args) {         ConstructorException c = **new**ConstructorException("gh", -1);     } } |
| --- |

output of the program

| Exception in thread "main" java.lang.IllegalArgumentException: age out of range: -1 at ConstructorException.<init>(ConstructorException.java:10) at ConstructorException.main(ConstructorException.java:19) Java Result: 1 |
| --- |

**Attributes**

**How do you specify a custom attribute for the entire assembly (rather than for a class)?** - Global attributes must appear after any top-level using clauses and before the first type or namespace declarations. An example of this is as follows:

using System;

[assembly : MyAttributeClass] class X {}

Note that in an IDE-created project, by convention, these attributes are placed in AssemblyInfo.cs.

**How do you mark a method obsolete?** -

[Obsolete] public int Foo() {...}

or

[Obsolete("This is a message describing why this method is obsolete")] public int Foo() {...}

Note: The O in Obsolete is always capitalized.

**CLR hosting and AppDomain**

#### 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).

**What is an application domain?**

*Application domain is the boundary within which an application runs. A process can contain multiple application domains.* Application domains provide an isolated environment to applications that is similar to the isolation provided by processes. An application running inside one application domain cannot directly access the code running inside another application domain. To access the code running in another application domain, an application needs to use a proxy.

### 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.

Microsoft have an [AppDomain FAQ](http://www.gotdotnet.com/team/clr/AppdomainFAQ.aspx).

### 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;

using System.Runtime.Remoting;

using System.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" );

CAppDomainInfo adInfo = (CAppDomainInfo)ad.CreateInstanceAndUnwrap(

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

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

return 0;

}

}

#### Serialization and Deserialization

### What is serialization and deserialization? What are their uses?

*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).

### 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.

### How Do You Use Serialization?

Serialization is handled primarily by classes and interfaces in the **System.Runtime.Serialization** namespace. To serialize an object, you need to create two things:

* A stream to contain the serialized objects.
* A formatter to serialize the objects into the stream.

### 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.

### 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 the ISerializable interface on the class whose instances are to be serialized.

### 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.

### 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.

### 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.

### 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 XmlArrayItem attibute like this:

public class Person

{

public string Name;

public int Age;

}

public class Population

{

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

}

| **1. Which class is responsible for binary serialization?** |
| --- |
| The **BinaryFormater** is the class responsible for the binary serialization and It's commonly used for the .Net Remoting. |
| **2. What does it take to make my object serializable?** |
| Your class must have the attribute **SerializableAttribute** and all its members must also be serializable, except if they are ignored with the attribute **NonSerializedAttribute**. Private and public fields are serialized by default.  [Serializable] public class Invoice {     private string clientName;     private DateTime date;     private double total;      [NonSerialized]     private string internalCode;        public string ClientName {         get {             return clientName;         }         set {             clientName = value;         }     }      public DateTime Date {         get {             return date;         }         set {             date = value;         }     }      public String Tag {         get {             return tag;         }         set {             tag = value;         }     }      public double Total {         get {             return total;         }         set {             total = value;         }     } } |
| **3. What are the main advantages of binary serialization?** |
| * Smaller * Faster * More powerful (support complex objects and read only properties) |
| **4. How do you encapsulate the binary serialization?** |
| public static void SerializeBinary( object aObject, string aFileName) {    using (FileStream \_FileStream = new FileStream(aFileName, FileMode.Create)) {      BinaryFormatter \_Formatter = new BinaryFormatter ();      \_Formatter.Serialize(\_FileStream, aObject);    } } |
| **5. How do you encapsulate the binary deserialization method?** |
| public static object DeserializeBinary(string aFileName) {    using (FileStream \_FileStream = new FileStream(aFileName, FileMode.Open)) {       BinaryFormatter \_Formatter = new BinaryFormatter ();       return \_Formatter.Deserialize(\_FileStream);    } } |
| **6. Will my read only properties be serialized?** |
| Yes if the properties encapsulate a field. By default all the private and public fields are serialized. Binary serialization is not related to properties. |
| **7. Is it possible to have circular reference?** |
| Yes it is, you can have circular reference and the binary serialization process will work fine. .Net generate the object graph before the executing the serialization and finally generate the stream. Unlike Xml serialization process, the **BinaryFormater** has no problem with the circular reference. |
| **8. Why my Dataset is so big when it's serialized in binary?** |
| By default the **DataSet** is serialized in Xml and the binary stream only wraps the Xml Data inside it. That's mean that the size is similar to the Xml size. .Net 2.0 add a new property named **RemotingFormat** used to change the binary serialization format of the **DataSet**. **SerializationFormat.Binary** will generate a better result.  For previous version, it's also possible to download the [DataSetSurrogate](http://support.microsoft.com/Default.aspx?scid=kb;en-us;829740) to reduce the size and increase the performance. |
| **9. How do I serialize a Dataset in binary?** |
| See answer [above](http://www.devolutions.net/articles/serialization.aspx#S18). |
| **10. How can I implement a custom serialization?** |
| If you need to control the serialization process of your class, you can implement the **ISerializable** interface which contains a method **GetObjectData** and a special constructor . Why use custom serialization ? By using it you will be able to handle version change in your class or get a better performance result. An exception of type **SerializationException** is raised if the fields does not exists.  #region ISerializable Members  //Special constructor protected CustomInvoice(SerializationInfo info, StreamingContext context) {     clientName = info.GetString("clientName");     date = info.GetDateTime("date");     total = info.GetDouble("total"); }  [SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter = true)] public void GetObjectData(SerializationInfo info, StreamingContext context) {     info.AddValue("clientName", clientName);     info.AddValue("date", date);     info.AddValue("total", total); }  #endregion |
| **11. When does a change in my object break the deserialization?** |
| Binary serialization is not tolerant to version change. There is no problem when a new field is added but if a field is missing, an exception is throw. New field will get the default value. .Net 2.0 include a new attribute named **OptionalFieldAttribute**. For previous version, you must implement your own custom serialization and handle manually the changes. |
| **12. When does a change in my object NOT break the deserialization?** |
| Version fault exception are only checked if the assembly is signed, all version change are ignored otherwise. |
| **13. How can I make my object version tolerant?** |
| Use the **OptionalFieldAttribute** or implement your own custom serialization with **ISerializable**. |
| **14. Why set VersionAdded of OptionalFieldAttribute since it's a free text?** |
| The parameter is only for informative purpose. .Net never checked it by default but it could be used in a custom serialization. Use the reflection to get the attribute of field and check the added version value. |
| **15. Does BinaryFormatter from .Net 1.1 is compatible with the BinaryFormatter 2.0?** |
| Absolutely, the **BinaryFormatter** 2.0 is 100% with other version, but it's not the case with the **SoapFormatter**. |
| **16. How can I modify a value just before the serialization or just after the deserialization?** |
| You can add custom attribute to some method. Your marked method will get called a the right time. This is usefull to initialize a property after the deserialization or to clean up your instance before the serialization.   * + **OnDeserializingAttribute** :This event happens before deserialization   + **OnDeserializedAttribute** :This event happens after deserialization   + **OnSerializingAttribute** :This event happens before serialization   + **OnSerializedAttribute** :This even happens after serialization   [Serializable] public class SecurityToken {   private string password;   private string userName;    private string Decrypt(string aPassword) {     // Decrypt the password here !!!     return password;   }    private string Encrypt(string aPassword) {     // Encrypt the password here !!!     return password;   }    [OnSerializing()]   internal void OnSerializingMethod(StreamingContext context) {     password = Encrypt(password);    }    [OnDeserialized()]   internal void OnDeserializedMethod(StreamingContext context) {     password = Decrypt(password);      // Set the default     if (userName == null) {       userName = Environment.UserName;     }   }    public string Password {     get {       return password;     }     set {       password = value;     }   }    public string UserName {     get {       return userName;     }     set {       userName = value;     }   } } |
| **17. How can I create a generic Binary deserialization method?** |
| // Binary deserialization (generic version with the return value casted) public static T DeserializeBinary<T>(string aFileName) {   using (FileStream \_FileStream = new FileStream(aFileName, FileMode.Open)) {     BinaryFormatter \_Formatter = new BinaryFormatter();     return (T)\_Formatter.Deserialize(\_FileStream);   } } |

#### *XML serialization questions*

| **1. Which class is responsible for Xml serialization?** |
| --- |
| **XmlSerializer** is responsible of the Xml serialization. |
| **2. What is the difference between the SoapFormatter and the XmlSerializer?** |
| **SoapFormatter** is used to create a Soap envelop and use an object graph to generate the result. The **XmlSerializer** process use only the public data and the result is a more common xml file. The Web Service in .Net use an **XmlSerializer** to generate the output contained in the Soap message. The **SoapFormatter** and the BinaryFormatter are used in the .Net Remoting serialization process. |
| **3. What does it take to make my object serializable?** |
| Nothing, but there is some constraint :   * + Your object must have a public empty constructor.   + Field or property must be public   + Their return type must also respect serialization rules.   + Property must be read write. |
| **4. What are the main advantages of Xml serialization?** |
| * + Based on international standard (XML).   + Cross platforms.   + Readable and can be edited easily. |
| **5. How do I encapsulate the Xml serialization method?** |
| public static void SerializeXml( object aObject, string aFileName) {    using (FileStream \_FileStream = new FileStream(aFileName, FileMode.Create)) {      XmlSerializer \_Serializer = new XmlSerializer ( aObject.GetType());      \_Serializer.Serialize(\_FileStream, aObject);    } } |
| **6. How do I encapsulate the Xml deserialization method?** |
| public static object DeserializeXml( string aFileName, Type aType) {    using (FileStream \_FileStream = new FileStream(aFileName, FileMode.Create)) {      XmlSerializer \_Serializer = new XmlSerializer (aType);      return \_Serializer.Deserialize(\_FileStream);    } } |
| **7. How can I create a generic Xml deserialization method?** |
| public static T DeserializeXml<T>(string aFileName) {    using (FileStream \_FileStream = new FileStream(aFileName, FileMode.Open)) {      XmlSerializer \_Serializer = new XmlSerializer (typeof(T));      return (T)\_Serializer.Deserialize(\_FileStream);    } } |
| **8. How can I ignore a property in serialization?** |
| If you use a **XmlSerializer**, mark your property with the custom attribute **XmlIgnoreAttribute** and if you use a **SoapFormatter**, use a **SoapIgnoreAttribute** instead. |
| **9. How can I rename a field or a property in the Xml output?** |
| Use the attribute **XmlElementAttribute** or **XmlAttributeAttribute** with the new name as parameter. To rename a class, use **the XmlTypeAttribute**.  [XmlType("city")] public class Town {     private string name;     private string state;      [XmlElement("townname")]     public string Name {         get {             return name;         }         set {             name = value;         }     }      [XmlAttribute("state")]     public string State {         get {             return state;         }         set {             state = value;         }     } }  Result:  <?xml version="1.0"?> <city state="CA">   <townname>Los Angeles</townname> </city> |
| **10. How can I read a field from an Xml stream without deserializing it?** |
| XPath can do the job.. This is an example where we read the town name (Xml element) )and the state attribute from the Xml file:  XmlDocument document = new XmlDocument(); document.Load("town.xml");  //Select an element string \_TownName = document.SelectSingleNode("//city/townname").InnerText;  //Select an attribute string \_State =    document.SelectSingleNode("//city").Attributes["state"].InnerText;  MessageBox.Show(\_TownName + " is in " + \_State);  This is an example where we read an application setting from a web config file.  XmlDocument configDocument = new XmlDocument(); configDocument.Load("web.config");  // Add the namespace with .Net web.config XmlNamespaceManager nsmgr = new XmlNamespaceManager(configDocument.NameTable); nsmgr.AddNamespace("ms", "http://schemas.microsoft.com/.NetConfiguration/v2.0");  string appSetting = configDocument.SelectSingleNode(     "/ms:configuration/ms:appSettings/ms:add[@key='DatabaseName']",      nsmgr).Attributes["value"].InnerText;  The orignal config file:  <configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">   <appSettings>     <add key="DatabaseName" value="Pubs"/>     <add key="DatabaseServer" value="(local)"/>   </appSettings> </configuration>  Note, you don't need the **XmlNamespaceManager** if you don't have a default namespace. Web.config in the ealier version does not contain a default namespace. |
| **11. How can I serialize a property as an Xml attribute?** |
| By default properties are serialized as Xml elements, but if you add an **XmlAttributeAttribute** to a property, .Net will generate an attribute instead. It's must be type compatible with an Xml attribute. See example [here](http://www.devolutions.net/articles/serialization.aspx#S29)  ...     [XmlAttribute("state")]     public string State {         get {             return state;         }         set {             state = value;         }     } |
| **12. How can I implement custom serialization?** |
| You need to Implement the interface **IXmlSerializable**. This class is available in the .Net 1.X but it's was not documented. It's now official available with .Net 2.0. With custom serialization, it's possible to optimize the output and generate only what is needed. In this example, we generate only the non empty properties  public class SessionInfo : IXmlSerializable {  #region IXmlSerializable Members    public System.Xml.Schema.XmlSchema GetSchema() {     return null;   }    public void ReadXml(XmlReader reader) {     UserName = reader.GetAttribute("UserName");      while (reader.Read()) {       if (reader.IsStartElement()) {         if (!reader.IsEmptyElement) {           string \_ElementName = reader.Name;              reader.Read(); // Read the start tag.              if(\_ElementName == "MachineName") {               MachineName = reader.ReadString();             } else {               reader.Read();             }           }         }       }      }    public void WriteXml(XmlWriter writer) {     if (!String.IsNullOrEmpty(UserName))       writer.WriteAttributeString("UserName", UserName);      if (!String.IsNullOrEmpty(MachineName))       writer.WriteElementString("MachineName", MachineName);   }    #endregion    private string machineName;   private string userName;    public string MachineName {     get {       return machineName;     }     set {       machineName = value;     }   }    public string UserName {     get {       return userName;     }     set {       userName = value;     }   } }  The output could be something like that if UserName and Machine are not empty :  <?xml version="1.0"?> <SessionInfo UserName="David">   <MachineName>MyMachine</MachineName> </SessionInfo>  or it could be like that if MachineName is empty :  <?xml version="1.0"?> <SessionInfo UserName="David"> |
| **13. How can I serialize a property array?** |
| Array are compatible with the serialization, but all elements must be of the same type. If not all types must be specified, see [below.](http://www.devolutions.net/articles/serialization.aspx#S214) |
| **14. How can I serialize an array with different types?** |
| It's possible to tag the array with all the possible type. Use **XmlInclude** on the class containing the array or **XmlArrayItem**. All the possible types must be specified with the attributes. Array types must be known because of the Xml schema. **XmlInclude** could be used for property that returned differente types. It's complicate object inheritance since all types must be known.  This example will fail with a message "There was an error generating the XML document." because the Cars could contain undefined types.  public class Car {}  public class Ford : Car{}   public class Honda: Car {}  public class Toyota: Car {}  public class CarSeller {   private List<Car> cars = new List<Car>();    public List<Car> Cars {     get {       return cars;     }     set {       cars = value;     }   } }  ... Ford \_Ford = new Ford(); Honda \_Honda = new Honda(); Toyota \_Toyota = new Toyota();  CarSeller \_Seller = new CarSeller(); \_Seller.Cars.Add(\_Ford); \_Seller.Cars.Add(\_Honda); \_Seller.Cars.Add(\_Toyota);  SerializationHelper.SerializeXml(\_Seller, @"seller.xml");  Three possible solutions:  **#1 Add XmlIncludeAttribute to the Seller class:**  [XmlInclude(typeof(Ford))] [XmlInclude(typeof(Honda))] [XmlInclude(typeof(Toyota))] public class CarSeller {   private List<Car> cars = new List<Car>();    public List<Car> Cars {     get {       return cars;     }     set {       cars = value;     }   } }  with this result  <?xml version="1.0"?> <CarSeller>   <Cars>     <Car xsi:type="Ford" />     <Car xsi:type="Honda" />     <Car xsi:type="Toyota" />   </Cars> </CarSeller>  **#2 Add XmlArrayItem to the property named Cars:**  public class CarSeller {   private List<Car> cars = new List<Car>();    [XmlArrayItem(typeof(Ford))]   [XmlArrayItem(typeof(Honda))]   [XmlArrayItem(typeof(Toyota))]   public List<Car> Cars {       get {           return cars;       }       set {           cars = value;       }   } }  with this result  <?xml version="1.0"?> <CarSeller>   <Cars>     <Ford />     <Honda />     <Toyota />   </Cars> </CarSeller>  **#3 Implement our own serialization with IXmlSerializable :**  see items [above](http://www.devolutions.net/articles/serialization.aspx#S212) |
| **15. How can I serialize a collection?** |
| Collection are serialized correctly, but they must contains only object of same types. Read only properties of type **ArrayList**, **List<type>** and other collections will be serialized and deserialized correctly.  public class Role {     private string name;      public string Name     {       get { return name; }       set { name = value; }     } }  public class UserAccount {     private string userName;      public string UserName {         get {             return userName;         }         set {             userName = value;         }     }      // Generic version     private List<Role> roles = new List<Role>();      public List<Role> Roles {         get {             return roles;         }     }      // ArrayList version     private ArrayList roleList = new ArrayList();      public ArrayList RoleList {         get {             return roleList;         }     }      // String collection version     private StringCollection roleNames = new StringCollection();      public StringCollection RoleNames {         get {             return roleNames;         }     }  UserAccount \_UserAccount = new UserAccount(); \_UserAccount.UserName = "dhervieux";  Role \_RoleAdmin = new Role(); \_RoleAdmin.Name = "Admin";  Role \_RoleSales = new Role(); \_RoleSales.Name = "Sales";  \_UserAccount.RoleList.Add(\_RoleAdmin); \_UserAccount.RoleList.Add(\_RoleSales); \_UserAccount.Roles.Add(\_RoleAdmin); \_UserAccount.Roles.Add(\_RoleSales); \_UserAccount.RoleNames.Add("Admin"); \_UserAccount.RoleNames.Add("Sales");  SerializationHelper.SerializeXml(\_UserAccount, @"useraccount.xml"); UserAccount \_Result =     SerializationHelper.DeserializeXml<UserAccount>(@"useraccount.xml");  will produce:  <?xml version="1.0"?> <UserAccount>   <UserName>dhervieux</UserName>   <Roles>     <Role>       <Name>Admin</Name>     </Role>     <Role>       <Name>Sales</Name>     </Role>   </Roles>   <RoleList>     <anyType xsi:type="Role">       <Name>Admin</Name>     </anyType>     <anyType xsi:type="Role">       <Name>Sales</Name>     </anyType>   </RoleList>   <RoleNames>     <string>Admin</string>     <string>Sales</string>   </RoleNames> </UserAccount> |
| **16. Why is the first serialization of each type of object is so long?** |
| **XmlSerializer** generate an assembly in memory optimized for each type. That's explain why the first call to a Web Service is so long. In .Net 2.0, there is an option in the project properties of Visual Studio to generate the Xml serialization assembly. Use it directly in the IDE or use **sgen.exe**, this tools come with the .Net Framework SDK. |
| **17. How can I optimize the serialization process?** |
| Pregenerate your serialization assembly with Visual Studio or **sgen.exe**. See details in answer above. Implementing your own serialization could also increase the performance. |
| **18. How can I serialize an array directly to stream?** |
| Yes it possible. .Net will name the **Array** and save the content. All the data must be of the same type.  bool[] \_BoolArray = new bool[] { true, false, false, true };  // Serialization SerializationHelper.SerializeXml(\_BoolArray, @"boolarray.xml");  //Deserialization \_Result = SerializationHelper.DeserializeXml<bool[]>(@"directboolarray.xml");  will produce:  <?xml version="1.0"?> <ArrayOfBoolean>   <boolean>true</boolean>   <boolean>false</boolean>   <boolean>false</boolean>   <boolean>true</boolean> </ArrayOfBoolean> |
| **19. Which serializer is used by a Web Services?** |
| Web Services are using SOAP to communicate, but returned objets or parameters are serialized with the **XmlSerializer**. Write unit test to be sure that your objects are serializable. |
| **20. Does read only properties are serialized?** |
| No, they are not, except for collections. |
| **21. How can I serialize a multidimensional array** |
| You need to encapsulate your array in a structure or a class an serialize it. Multidimensional array are not serializable by default. |
| **22. How can I avoid serialization for an empty list or property with a default value?** |
| There is an undocumented way of doing that, you need to create a method named ShouldSerialize<propertyname> where <propertyname> is replaced by the property name. This method should return a boolean that indicate if the property must be serialized or not. For exemple, if you have list with no item, there is no need to serialize an empty list.  public class Registration {   private string[] users = new string[0];    public bool ShouldSerializeUsers() {     return users.Length > 0;   }    public string[] Users {     get {       return users;     }     set {       users = value;     }   } }  Result :  <?xml version="1.0"?> <Registration />  Without the ShouldSerializeUsers :  <?xml version="1.0"?> <Registration >   <Users /> </Registration> |
| **23. Why my object is marked as Serializable (like SortedList) and it's does not work?** |
| The **SerializationAttribute** is only used for the binary serialization. That does not mean that it will work with an **XmlSerializer**. That's the case of the **SortedList**. |
| **24. How to remove the default namespace in the serialization?** |
| It's possible to remove the **xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"** and **xmlns:xsd="http://www.w3.org/2001/XMLSchema"** from the serialization result, it's possible to add an **XmlSerializerNamespaces** with an empty namespace mapping.  User \_User = new User(new string[] { "Admin", "Manager" });  using (FileStream \_FileStream = new FileStream("user.xml", FileMode.Create)) {   XmlSerializer \_Serializer = new XmlSerializer(\_User.GetType());    XmlSerializerNamespaces ns = new XmlSerializerNamespaces();   ns.Add("", "");    \_Serializer.Serialize(\_FileStream, \_User, ns); } |

### *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.

**Describe the advantages of writing a managed code application instead of unmanaged one. What’s involved in certain piece of code being managed?**

The advantages include automatic garbage collection, memory management, support for versioning and security. These advantages are provided through .NET FCL and CLR, while with the unmanaged code similar capabilities had to be implemented through third-party libraries or as a part of the application itself.

**Are COM objects managed or unmanaged?**

Since COM objects were written before .NET, apparently they are unmanaged.

**So can a COM object talk to a .NET object?**

Yes, through Runtime Callable Wrapper (RCW) or PInvoke.

**Interop Services?**The common language runtime provides two mechanisms for interoperating with unmanaged code:

* Platform invoke, which enables managed code to call functions exported from an unmanaged library.
* COM interop, which enables managed code to interact with COM objects through interfaces.

Both platform invoke and COM interop use interop marshaling to accurately move method arguments between caller and callee and back, if required.

**How does u handle this COM components developed in other programming languages in .NET?**

**What is RCW (Runtime Callable Wrappers)?**The common language runtime exposes COM objects through a proxy called the runtime callable wrapper (RCW). Although the RCW appears to be an ordinary object to .NET clients, its primary function is to marshal calls between a .NET client and a COM object.

**What is CCW (COM Callable Wrapper)**

A proxy object generated by the common language runtime so that existing COM applications can use managed classes, including .NET Framework classes, transparently.

**How CCW and RCW is working?**  
\*\*

**How will you register com+ services?**The .NET Framework SDK provides the .NET Framework Services Installation Tool (Regsvcs.exe - a command-line tool) to manually register an assembly containing serviced components. You can also access these registration features programmatically with the System.EnterpriseServicesRegistrationHelper class by creating an instance of class RegistrationHelper and using the method InstallAssembly

**What is use of ContextUtil class?**ContextUtil is the preferred class to use for obtaining COM+ context information.

**What is the new three features of COM+ services, which are not there in COM (MTS)?**\*\*

**Is the COM architecture same as .Net architecture?  What is the difference between them?**\*\*

**Can we copy a COM dll to GAC folder?**\*\*

**What is Pinvoke?**Platform invoke is a service that enables managed code to call unmanaged functions implemented in dynamic-link libraries (DLLs), such as those in the Win32 API. It locates and invokes an exported function and marshals its arguments (integers, strings, arrays, structures, and so on) across the interoperation boundary as needed.

**Is it true that COM objects no longer need to be registered on the server?**Yes and No. Legacy COM objects still need to be registered on the server before they can be used. COM developed using the new .NET Framework will not need to be registered. Developers will be able to auto-register these objects just by placing them in the 'bin' folder of the application.

**Can .NET Framework components use the features of Component Services?**Answer: Yes, you can use the features and functions of Component Services from a .NET Framework component. <http://msdn.microsoft.com/library/techart/Pahlcompserv.htm>

**How do you generate an RCW from a COM object?**

Use the Type Library Import utility shipped with SDK. tlbimp COMobject.dll /out:.NETobject.dll or reference the COM library from Visual Studio in your project.

**I can’t import the COM object that I have on my machine.**

Did you write that object? You can only import your own objects. If you need to use a COM component from another developer, you should obtain a Primary Interop Assembly (PIA) from whoever authored the original object.

**How do you call unmanaged methods from your .NET code through PInvoke?** Supply a DllImport attribute. Declare the methods in your .NET code as *static extern*. Do not implement the methods as they are implemented in your unmanaged code, you’re just providing declarations for method signatures.

#### 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 );

}

}

**Can you retrieve complex data types like structs from the PInvoke calls?**

Yes, just make sure you re-declare that struct, so that managed code knows what to do with it.

**I want to expose my .NET objects to COM objects. Is that possible?**

Yes, but few things should be considered first. Classes should implement interfaces explicitly. Managed types must be public. Methods, properties, fields, and events that are exposed to COM must be public. Types must have a public default constructor with no arguments to be activated from COM. Types cannot be abstract.

**Can you inherit a COM class in a .NET application?**

The .NET Framework extends the COM model for reusability by adding implementation inheritance. Managed types can derive directly or indirectly from a COM coclass; more specifically, they can derive from the runtime callable wrapper generated by the runtime. The derived type can expose all the method and properties of the COM object as well as methods and properties implemented in managed code. The resulting object is partly implemented in managed code and partly implemented in unmanaged code.

**Suppose I call a COM object from a .NET applicaiton, but COM object throws an error. What happens on the .NET end?**

COM methods report errors by returning HRESULTs; .NET methods report them by throwing exceptions. The runtime handles the transition between the two. Each exception class in the .NET Framework maps to an HRESULT.

### Does .NET replace COM?

This subject causes a lot of controversy, as you'll see if you read the mailing list archives. Take a look at the following two threads:

<http://discuss.develop.com/archives/wa.exe?A2=ind0007&L=DOTNET&D=0&P=68241>  
<http://discuss.develop.com/archives/wa.exe?A2=ind0007&L=DOTNET&P=R60761>

The bottom line is that .NET has its own mechanisms for type interaction, and they don't use COM. No IUnknown, no IDL, no typelibs, no registry-based activation. This is mostly good, as a lot of COM was ugly. Generally speaking, .NET allows you to package and use components in a similar way to COM, but makes the whole thing a bit easier.

### Is DCOM dead?

Pretty much, for .NET developers. The .NET Framework has a new remoting model which is not based on DCOM. DCOM was pretty much dead anyway, once firewalls became widespread and Microsoft got [SOAP](http://www.w3.org/TR/soap/) fever. Of course DCOM will still be used in interop scenarios.

### Is COM+ dead?

Not immediately. The approach for .NET 1.0 was to provide access to the existing COM+ services (through an interop layer) rather than replace the services with native .NET ones. Various tools and attributes were provided to make this as painless as possible. Over time it is expected that interop will become more seamless - this may mean that some services become a core part of the CLR, and/or it may mean that some services will be rewritten as managed code which runs on top of the CLR.

For more on this topic, search for postings by Joe Long in the archives - Joe is the MS group manager for COM+. Start with this message:

<http://discuss.develop.com/archives/wa.exe?A2=ind0007&L=DOTNET&P=R68370>

### Can I use COM components from .NET programs?

Yes. COM components are accessed from the .NET runtime via a Runtime Callable Wrapper (RCW). This wrapper turns the COM interfaces exposed by the COM component into .NET-compatible interfaces. For oleautomation interfaces, the RCW can be generated automatically from a type library. For non-oleautomation interfaces, it may be necessary to develop a custom RCW which manually maps the types exposed by the COM interface to .NET-compatible types.

Here's a simple example for those familiar with ATL. First, create an ATL component which implements the following IDL:

import "oaidl.idl";

import "ocidl.idl";

[

object,

uuid(EA013F93-487A-4403-86EC-FD9FEE5E6206),

helpstring("ICppName Interface"),

pointer\_default(unique),

oleautomation

]

interface ICppName : IUnknown

{

[helpstring("method SetName")] HRESULT SetName([in] BSTR name);

[helpstring("method GetName")] HRESULT GetName([out,retval] BSTR \*pName );

};

[

uuid(F5E4C61D-D93A-4295-A4B4-2453D4A4484D),

version(1.0),

helpstring("cppcomserver 1.0 Type Library")

]

library CPPCOMSERVERLib

{

importlib("stdole32.tlb");

importlib("stdole2.tlb");

[

uuid(600CE6D9-5ED7-4B4D-BB49-E8D5D5096F70),

helpstring("CppName Class")

]

coclass CppName

{

[default] interface ICppName;

};

};

When you've built the component, you should get a typelibrary. Run the TLBIMP utility on the typelibary, like this:

tlbimp cppcomserver.tlb

If successful, you will get a message like this:

Typelib imported successfully to CPPCOMSERVERLib.dll

You now need a .NET client - let's use C#. Create a .cs file containing the following code:

using System;

using CPPCOMSERVERLib;

public class MainApp

{

static public void Main()

{

CppName cppname = new CppName();

cppname.SetName( "bob" );

Console.WriteLine( "Name is " + cppname.GetName() );

}

}

Compile the C# code like this:

csc /r:cppcomserverlib.dll csharpcomclient.cs

Note that the compiler is being told to reference the DLL we previously generated from the typelibrary using TLBIMP. You should now be able to run csharpcomclient.exe, and get the following output on the console:

Name is bob

### Can I use .NET components from COM programs?

Yes. .NET components are accessed from COM via a COM Callable Wrapper (CCW). This is similar to a RCW (see previous question), but works in the opposite direction. Again, if the wrapper cannot be automatically generated by the .NET development tools, or if the automatic behaviour is not desirable, a custom CCW can be developed. Also, for COM to 'see' the .NET component, the .NET component must be registered in the registry.

Here's a simple example. Create a C# file called testcomserver.cs and put the following in it:

using System;

using System.Runtime.InteropServices;

namespace AndyMc

{

[ClassInterface(ClassInterfaceType.AutoDual)]

public class CSharpCOMServer

{

public CSharpCOMServer() {}

public void SetName( string name ) { m\_name = name; }

public string GetName() { return m\_name; }

private string m\_name;

}

}

Then compile the .cs file as follows:

csc /target:library testcomserver.cs

You should get a dll, which you register like this:

regasm testcomserver.dll /tlb:testcomserver.tlb /codebase

Now you need to create a client to test your .NET COM component. VBScript will do - put the following in a file called comclient.vbs:

Dim dotNetObj

Set dotNetObj = CreateObject("AndyMc.CSharpCOMServer")

dotNetObj.SetName ("bob")

MsgBox "Name is " & dotNetObj.GetName()

and run the script like this:

wscript comclient.vbs

And hey presto you should get a message box displayed with the text "Name is bob".

An alternative to the approach above it to use the [dm.net moniker](http://staff.develop.com/jasonw/clr/readme.htm) developed by Jason Whittington and Don Box.

### Is ATL redundant in the .NET world?

Yes. ATL will continue to be valuable for writing COM components for some time, but it has no place in the .NET world.

**How do you directly call a native function exported from a DLL?**

Here’s a quick example of the DllImport attribute in action:

using System.Runtime.InteropServices;

class C

{

[DllImport("user32.dll")]

public static extern int MessageBoxA(int h, string m, string c, int type);

public static int Main()

{

return MessageBoxA(0, "Hello World!", "Caption", 0);

}

}

This example shows the minimum requirements for declaring a C# method that is implemented in a native DLL. The method C.MessageBoxA() is declared with the static and external modifiers, and has the DllImport attribute, which tells the compiler that the implementation comes from the user32.dll, using the default name of MessageBoxA. For more information, look at the Platform Invoke tutorial in the documentation.

**How do I simulate optional parameters to COM calls?**

You must use the Missing class and pass Missing.Value (in System.Reflection) for any values that have optional parameters.

### 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.

**Automatic Memory Management - 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.

**What is a garbage collector?**

A garbage collector performs periodic checks on the managed heap to identify objects that are no longer required by the program and removes them from memory.

**How garbage collection happens?**

The methods in GC class influence when garbage collection is performed on an object, and when resources allocated by an object are released. Properties in this class provide information about the total amount of memory available in the system and the age category, or generation, of memory allocated to an object.

The garbage collector tracks and reclaims objects allocated in managed memory. Periodically, the garbage collector performs garbage collection to reclaim memory allocated to objects for which there are no valid references. *Garbage collection happens automatically when a request for memory cannot be satisfied using available free memory.* Alternatively, an application can force garbage collection using the [Collect](http://msdn.microsoft.com/en-us/library/system.gc.collect(VS.80).aspx) method.

Garbage collection consists of the following steps:

1. The garbage collector searches for managed objects that are referenced in managed code.
2. The garbage collector attempts to finalize objects that are not referenced.
3. The garbage collector frees objects that are not referenced and reclaims their memory

#### What are Generations in Garbage Collector? What is the importance & use of the

#### generations process?

*Generations in the Garbage Collector is a way of enhancing the garbage collection*

*performance.* In .NET, all resources are allocated space (memory) from the heap. Objects

are automatically freed from the managed heap when they are no longer required by the

application.There are 3 Generations...0,1,2.

* **Generation 0** - When an object is initialized, its in generation 0. These are new objects that

have never been played around with by the GC. As and when more objects get created, the

process of Garbage Collection is invoked by the **CLR**.

* **Generation 1** - The objects that survive the garbage collection process are considered to

be in generation 1. These are the old objects.

* **Generation 2** - As more new objects get created and added to the memory, the new

objects are added to generation 0, the generation 1 old objects become older, and so are

considered to be in generation 2. Generation 2 is the highest level generation in the

garbage collection process. Any further garbage collection process occuring causes the

level 1 objects promoted to level 2, and the level 2 objects stay in level 2 itself, as this

generation level is the highest level.

*Importance & use*

It’s actually the priority the GC gives to objects while freeing objects from the heap. During every GC cycle, the objects in the Generation 0 are scanned first -> Followed by Generation 1 and then 2. This is because the generation 0 objects are usually short term objects that need to be freed.

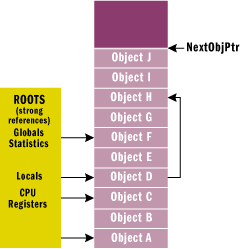
*The newer an object, the shorter its life is. The older an object, longer its life is.*

This process also helps in categorizing the memory heap as to where the de-allocation

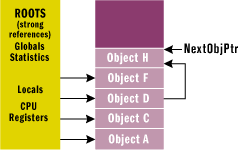
needs to be done first and where next.

#### Explain how GC algorithm works.

The garbage collector checks to see if there are any objects in the heap that are no longer being used by the application. If such objects exist, then the memory used by these objects can be reclaimed. (If no more memory is available for the heap, then the new operator throws an OutOfMemoryException.) How does the garbage collector know if the application is using an object or not? As you might imagine, this isn't a simple question to answer.  
Every application has a set of roots. Roots identify storage locations, which refer to objects on the managed heap or to objects that are set to null. For example, all the global and static object pointers in an application are considered part of the application's roots. In addition, any local variable/parameter object pointers on a thread's stack are considered part of the application's roots. Finally, any CPU registers containing pointers to objects in the managed heap are also considered part of the application's roots. The list of active roots is maintained by the just-in-time (JIT) compiler and common language runtime, and is made accessible to the garbage collector's algorithm.  
When the garbage collector starts running, it makes the assumption that all objects in the heap are garbage. In other words, it assumes that none of the application's roots refer to any objects in the heap. Now, the garbage collector starts walking the roots and building a graph of all objects reachable from the roots. For example, the garbage collector may locate a global variable that points to an object in the heap.  
Figure 2 shows a heap with several allocated objects where the application's roots refer directly to objects A, C, D, and F. All of these objects become part of the graph. When adding object D, the collector notices that this object refers to object H, and object H is also added to the graph. The collector continues to walk through all reachable objects recursively.

  
**Figure 2** **Allocated Objects in Heap**

Once this part of the graph is complete, the garbage collector checks the next root and walks the objects again. As the garbage collector walks from object to object, if it attempts to add an object to the graph that it previously added, then the garbage collector can stop walking down that path. This serves two purposes. First, it helps performance significantly since it doesn't walk through a set of objects more than once. Second, it prevents infinite loops should you have any circular linked lists of objects.  
Once all the roots have been checked, the garbage collector's graph contains the set of all objects that are somehow reachable from the application's roots; any objects that are not in the graph are not accessible by the application, and are therefore considered garbage. The garbage collector now walks through the heap linearly, looking for contiguous blocks of garbage objects (now considered free space). The garbage collector then shifts the non-garbage objects down in memory (using the standard memcpy function that you've known for years), removing all of the gaps in the heap. Of course, moving the objects in memory invalidates all pointers to the objects. So the garbage collector must modify the application's roots so that the pointers point to the objects' new locations. In addition, if any object contains a pointer to another object, the garbage collector is responsible for correcting these pointers as well. Figure 3 shows the managed heap after a collection.

  
**Figure 3** **Managed Heap after Collection**

After all the garbage has been identified, all the non-garbage has been compacted, and all the non-garbage pointers have been fixed-up, the NextObjPtr is positioned just after the last non-garbage object. At this point, the new operation is tried again and the resource requested by the application is successfully created.

#### Define finalization. When do we need to implement Finalize method in a type?

*Finalization* is a mechanism offered by the CLR that allows an object to perform some graceful

cleanup prior to the garbage collector reclaiming the object's memory. Finalization is another feature of garbage collector.

Any type that wraps a native resource, such as a file, network connection, socket, mutex, or other type, must support finalization. Basically, the type implements a method named **Finalize**. *When the garbage collector determines that an object is garbage, it calls the object's* ***Finalize*** *method (if it*

*exists).* I think of it this way: any type that implements the **Finalize** method is in effect stating

that all of its objects want a last meal before they are killed.

Here is an oversimplification of what happens: when the garbage collector detects that an object is garbage, the garbage collector calls the object's Finalize method (if it exists) and then the object's memory is reclaimed. For example, let's say you have the following type (in C#):

public class BaseObj {

public BaseObj() {

}

protected override void Finalize() {

// Perform resource cleanup code here...

// Example: Close file/Close network connection

Console.WriteLine("In Finalize.");

}

}

Now you can create an instance of this object by calling:

BaseObj bo = new BaseObj();

Some time in the future, the garbage collector will determine that this object is garbage. When that happens, the garbage collector will see that the type has a Finalize method and will call the method, causing "In Finalize" to appear in the console window and reclaiming the memory block used by this object.

#### Why should you avoid using a Finalize method when designing a type?

When designing a type, it's best if you avoid using a **Finalize** method for several reasons all

related to performance:

#### Finalizable objects get promoted to older generations, which increases memory pressure and prevents the object's memory from being collected when the garbage collector determines the object is garbage. In addition, all objects referred to directly or indirectly by this object get promoted as well. Generations and promotions will be discussed in Part 2 of this article.

#### Finalizable objects take longer to allocate.

#### Forcing the garbage collector to execute a Finalize method can significantly hurt performance. Remember, each object is finalized. So if I have an array of 10,000 objects, each object must have its Finalize method called.

#### Finalizable objects may refer to other (non-finalizable) objects, prolonging their lifetime unnecessarily. In fact, you might want to consider breaking a type into two different types: a lightweight type with a Finalize method that doesn't refer to any other objects, and a separate type without a Finalize method that does refer to other objects.

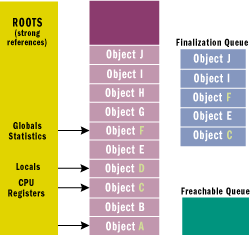
#### You have no control over when the Finalize method will execute. The object may hold on to resources until the next time the garbage collector runs.

#### When an application terminates, some objects are still reachable and will not have their Finalize method called. This can happen if background threads are using the objects or if objects are created during application shutdown or AppDomain unloading. In addition, by default, Finalize methods are not called for unreachable objects when an application exits so that the application may terminate quickly. Of course, all operating system resources will be reclaimed, but any objects in the managed heap are not able to clean up gracefully. You can change this default behavior by calling the System.GC type's RequestFinalizeOnShutdown method. However, you should use this method with care since calling it means that your type is controlling a policy for the entire application.

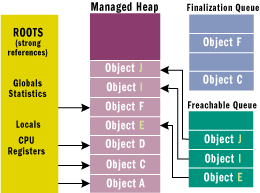
#### The runtime doesn't make any guarantees as to the order in which Finalize methods are called. For example, let's say there is an object that contains a pointer to an inner object. The garbage collector has detected that both objects are garbage. Furthermore, say that the inner object's Finalize method gets called first. Now, the outer object's Finalize method is allowed to access the inner object and call methods on it, but the inner object has been finalized and the results may be unpredictable. For this reason, it is strongly recommended that Finalize methods not access any inner, member objects.

**Explain how two GCs are required to reclaim memory used by objects that require finalization.**

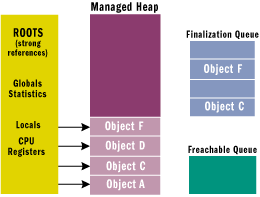
On the surface, finalization seems pretty straightforward: you create an object and when the object is collected, the object's Finalize method is called. But there is more to finalization than this.  
When an application creates a new object, the new operator allocates the memory from the heap. If the object's type contains a Finalize method, then a pointer to the object is placed on the finalization queue. The finalization queue is an internal data structure controlled by the garbage collector. Each entry in the queue points to an object that should have its Finalize method called before the object's memory can be reclaimed.   
**Figure 5** shows a heap containing several objects. Some of these objects are reachable from the application's roots, and some are not. When objects C, E, F, I, and J were created, the system detected that these objects had Finalize methods and pointers to these objects were added to the finalization queue.

  
**Figure 5** **A Heap with Many Objects**

When a GC occurs, objects B, E, G, H, I, and J are determined to be garbage. The garbage collector scans the finalization queue looking for pointers to these objects. When a pointer is found, the pointer is removed from the finalization queue and appended to the freachable queue (pronounced "F-reachable"). The freachable queue is another internal data structure controlled by the garbage collector. Each pointer in the freachable queue identifies an object that is ready to have its Finalize method called.  
After the collection, the managed heap looks like **Figure 6**. Here, you see that the memory occupied by objects B, G, and H has been reclaimed because these objects did not have a Finalize method that needed to be called. However, the memory occupied by objects E, I, and J could not be reclaimed because their Finalize method has not been called yet.

  
**Figure 6** **Managed Heap after Garbage Collection**

There is a special runtime thread dedicated to calling Finalize methods. When the freachable queue is empty (which is usually the case), this thread sleeps. But when entries appear, this thread wakes, removes each entry from the queue, and calls each object's Finalize method. Because of this, you should not execute any code in a Finalize method that makes any assumption about the thread that's executing the code. For example, avoid accessing thread local storage in the Finalize method.  
The interaction of the finalization queue and the freachable queue is quite fascinating. First, let me tell you how the freachable queue got its name. The f is obvious and stands for finalization; every entry in the freachable queue should have its Finalize method called. The "reachable" part of the name means that the objects are reachable. To put it another way, the freachable queue is considered to be a root just like global and static variables are roots. Therefore, if an object is on the freachable queue, then the object is reachable and is not garbage.   
In short, when an object is not reachable, the garbage collector considers the object garbage. Then, when the garbage collector moves an object's entry from the finalization queue to the freachable queue, the object is no longer considered garbage and its memory is not reclaimed. At this point, the garbage collector has finished identifying garbage. Some of the objects identified as garbage have been reclassified as not garbage. The garbage collector compacts the reclaimable memory and the special runtime thread empties the freachable queue, executing each object's Finalize method.

  
**Figure 7** **Managed Heap after Second Garbage Collection**

The next time the garbage collector is invoked, it sees that the finalized objects are truly garbage, since the application's roots don't point to it and the freachable queue no longer points to it. Now the memory for the object is simply reclaimed. The important thing to understand here is that two GCs are required to reclaim memory used by objects that require finalization. In reality, more than two collections may be necessary since the objects could get promoted to an older generation. **Figure 7** shows what the managed heap looks like after the second GC.

#### 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.

#### Why do we need the Dispose Pattern when we have finalization technique offered by

#### CLR to clean up un-managed resources in a type?

The *Finalize*method is incredibly useful because it ensures that native resources aren't

leaked when managed objects have their memory reclaimed. Finalizer has to be implemented in a type when it uses unmanged resources.

Finalization technique to clean up unmanaged native resources has below limitations:

* *Finalize*method does not guarantee when it will be called as it is called by GC. Implemeting Finalize method in type causes non deterministic finalization.
* *Finalize*method isn't a public method; a user of the class can't call it explicitly.
* Use of finalization technique creates a lot of performance issues.

Implementing Dispose pattern can overcome all of the above pitfalls of finalization technique offered by GC. Dispose pattern allows deterministic finalization by calling Dispose method explicitly.

Note that a type implementing Dispose pattern should also implement finalizer so as to ensure clean up of unmanaged resources in case user of the type forgets to call Dispose method.

### How does the generational garbage collector in the .NET CLR manage object lifetime?

### What is non-deterministic finalization?

### The hugely simplistic version is that every time it garbage-collects, it starts by assuming everything

### to be garbage, then goes through and builds a list of everything reachable. Those become not-

### garbage, everything else doesn't, and gets thrown away. What makes it generational is that every

### time an object goes through this process and survives, it is noted as being a member of an older

### generation (up to 2, right now). When the garbage-collector is trying to free memory, it starts with

### the lowest generation (0) and only works up to higher ones if it can't free up enough space, on the

### grounds that shorter-lived objects are more likely to have been freed than longer-lived ones.

### *Non-deterministic finalization* implies that the destructor (if any) of an object will not necessarily be

### run (nor its memory cleaned up, but that's a relatively minor issue) immediately upon its going out

### of scope. Instead, it will wait until first the garbage collector gets around to finding it, and then the

### finalisation queue empties down to it; and if the process ends before this happens, it may not be

### finalised at all. (Although the operating system will usually clean up any process-external resources

### left open - note the usually there, especially as the exceptions tend to hurt a lot.)

### Explain the Dispose pattern and how and when it should be used?

### When creating .NET Disposable types, you usually want to create a Finalize (C# destructor

### syntax), Dispose(), Dispose(bool), and inherit from System.IDisposable.  *Almost all*

### *of your actual object cleanup will occur inside Dispose(bool).* That's the pattern they

### came up with, and it's a good way to keep all your cleanup in one place.  We'll get to cases

### in a second, but first, here's a hunk of C# code plaigarized from the whitepaper:

public class BothType: IDisposable{  
   ~BothType(){  //finalizer  
    Dispose(false);  
  }

public void Dispose()

{  
    Dispose(true);  
    GC.SupressFinalize(this);  
  }

  protected virtual void Dispose(bool disposing){  
    if(disposing){  
      //'Disposable' managed resource cleanup code

// You should only clean up your Disposable member objects.  Don't bother with objects that

//don't have a Disposer, those objects will be cleaned up just fine by the GC.  
    }  
    //Unmanaged resource cleanup code  
    base.Dispose(disposing);  
  }

……  
}

When Disposing, the first call is always to Dispose().  Biggest take-away from one of my talks with Brandon.  All disposing starts with that call.  The CLR way of doing "chaining" destructors is the manual Dispose(bool) pattern.  Basically, the user calls Dispose(), and inside that function, Dispose(true) is called.  This does a callvirt on Dispose(bool), which will thunk down to the *lowest child instance*.  Then, you have the manual base.Dispose(disposing) calls that will walk up the chain.  I had to step up to my whiteboard for a while to prove to myself that this pattern works - but it does.

*This pattern should be used for freeing unmanaged resources.  In a nutshell, the Garbage Collector really only gets invoked when there's memory pressure.  But the GC can't "feel" pressure from unmanaged resources.  If you have a bunch of objects hanging around that have gobs of natively allocated memory (C++ new, or Marshal.AllocHGlobal), the GC won't know about this memory pressure, and won't begin freeing up these objects.*

### Why GC.SuppressFinalize() be called inside the Dispose method?

To ensure that the Garbage Collector won't Finalize an object that's already been Disposed.  In general, you're either going to Dispose an object, or the GC will Finalize it, not both.

### What is the role of the finalizer in the Dispose pattern?

The Finalizer is a backup system in this paradigm. It should be there only to ensure that all unmanaged resources get cleaned up in case the user forgot to call Dispose() on the object. That is, if a Disposable object is being Finalized, it's probably a mistake. So a call to Finalize() yields a call to Dispose(false), which takes care of only those unmanaged resources it holds, those that the GC is unable to clean up.

Note: a call to Dispose(false) should only clean up the unmanaged resources.

### What is the difference between Finalize() and Dispose()?

Both Finalize and Dispose are used to free unmanaged resources. Below are the differences.

| **Finalize** | **Dispose** |
| --- | --- |
| 1. Is a destructor, called by Garbage Collector when the object goes out of scope | 1. Is a method, called explicitly by the user of an object that implements System.IDisposable interface |
| 2. Finalize() is implicitly called by the runtime – CLR when ever it assumes to be appropriate. Finalize/Destructor cannot be called by User code | 2. Dispose() is explicitly called by the user of the object |
| 3. Implement it when you have unmanaged resources in your code, and want to make sure that these resources are freed when the Garbage collection happens. | 3. Same purpose as finalize, to free unmanaged resources. However, implement this when you are writing a custom class that will be used by other users. |
| 4. Finalization is not deterministic. | 4. Invoking Dispose() method can deterministically finalize any object that implements the IDisposable interface. |

**What is deterministic finalization?**

One of the strengths of the Common Language Runtime (CLR) is its memory management abilities. It spawns garbage collection threads that clean up all of the objects that are no longer in use and frees up space in the memory heap that has been allocated to them. Even though the CLR guarantees that your unused objects will be cleaned up eventually, there’s no way of knowing when or how it’ll be removed from the memory heap. Theoretically, this could be years from the time that the object is instantiated.

There’s no good reason not to deterministically clean up your objects. Not only is it excellent programming practice, but you’re able to ensure that the object and all of the components that it encapsulates are being cleaned up immediately.

*You can deterministically finalize any object that implements the IDisposable interface by invoking its Dispose() method.* If you use C# you have the advantage of the using statement which allows you to define a scope at the end of which an object will be disposed. One of the most common examples given uses a SqlConnection object:

using ( SqlConnection test = new SqlConnection( ... ) ) {

// code implementation

} // SqlConnection.Dispose() is automatically invoked

At the end of this scope, the SqlConnection’s Dispose() method is automatically invoked and all resources held by this object are released.

By adopting the practice of deterministic finalization, you’ll be able to efficaciously manage your resources. Additionally, this enables you to ensure that there are no objects that contain large chunks of unmanaged resources sitting around waiting for the garbage collector to come around and clean them up.

**Question on output through console method in finalize method?**

**What Causes Finalize Methods to Be Called?**

Finalize methods are called at the completion of a garbage collection, which is started by one

of the following five events:

1. **Generation 0 is full** When generation 0 is full, a garbage collection starts. This event

is by far the most common way for **Finalize** methods to be called because it occurs

naturally as the application code runs, allocating new objects. I'll discuss generations

later in this chapter.

1. **Code explicitly calls System.GC**'**s static Collect method** Code can explicitly request

that the CLR perform a collection. Although Microsoft strongly discourages such

requests, at times it might make sense for an application to force a collection.

1. **Windows is reporting low memory conditions** The CLR internally uses the Win32

*CreateMemoryResourceNotification*and *QueryMemoryResourceNotification*functions

to monitor system memory overall. If Windows reports low memory, the CLR will force

a garbage collection in an effort to free up dead objects to reduce the size of a process'

working set.

1. **The CLR is unloading an AppDomain** When an AppDomain unloads, the CLR considers

nothing in the AppDomain to be a root, and a garbage collection consisting of all

generations is performed.

1. **The CLR is shutting down** The CLR shuts down when a process terminates normally

(as opposed to an external shutdown via Task Manager, for example). During this shutdown,

the CLR considers nothing in the process to be a root and calls the **Finalize**

method for all objects in the managed heap. Note that the CLR does not attempt to compact

or free memory here because the whole process is terminating, and Windows will

reclaim all of the processes' memory.

The CLR uses a special, dedicated thread to call **Finalize** methods.

### How is the using () pattern useful? What is IDisposable? How does it support

### deterministic finalization?

The advantage of the using statement is that it allows to define a scope at the end of which an object will be disposed. One of the most common examples given uses a SqlConnection object:

using ( SqlConnection test = new SqlConnection( ... ) ) {

// code implementation

} // SqlConnection.Dispose() is automatically invoked

At the end of this scope, the SqlConnection’s Dispose() method is automatically invoked and all resources held by this object are released.

You can deterministically finalize any object that implements the IDisposable interface by invoking its Dispose() method. The C# using statement calls the Dispose() method automaticaaly. So, the using statement is able to support deterministic finalization. Deterministically cleaning up objects is not only an excellent programming practice, but also it ensures that the object and all of the components that it encapsulates are being cleaned up immediately.

### Memory leaks in .NET (Observer pattern) and the Dispose pattern. How do you find and

### solve performance problems?

### Can we call or override the Finalize method?

*You cannot call or override the Finalize method. It is generated implicitly if you have a destructor for the class.* This is shown in the following piece of C# code:—

class Test

{

// Some Code

~Test

{

//Necessary cleanup code

}

}

In the preceding code, the ~Test syntax declares an explicit destructor in C#, letting you write explicit cleanup code that will run during the finalize operation.  
  
The framework implicitly translates the explicit destructor to create a call to Finalize:

protected override void Finalize()

{

try

{

//Necessary cleanup code

}

finally

{

base.Finalize();

}

}

Note that the generated code above calls the base.Finalize method.

#### 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.

### 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.

There was an [interesting thread](http://discuss.develop.com/archives/wa.exe?A2=ind0007&L=DOTNET&P=R24819) on the DOTNET list, started by Chris Sells, about the implications of non-deterministic destruction of objects in C#. In October 2000, Microsoft's Brian Harry posted a [lengthy analysis](http://discuss.develop.com/archives/wa.exe?A2=ind0010A&L=DOTNET&P=R28572) of the problem. Chris Sells' [response](http://discuss.develop.com/archives/wa.exe?A2=ind0010C&L=DOTNET&P=R983) to Brian's posting is here.

### 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.

### 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.

### 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.)

For classes that need to implement IDisposable *and* override Finalize, see Microsoft's [documented pattern](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconFinalizeDispose.asp).

Note that some developers argue that implementing a Finalizer is always a bad idea, as it hides a bug in your code (i.e. the lack of a Dispose call). A less radical approach is to implement Finalize but include a Debug.Assert at the start, thus signalling the problem in developer builds but allowing the cleanup to occur in release builds.

### 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.

### 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.

### 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.

### When do I need to use GC.KeepAlive?

It's very unintuitive, but the runtime can decide that an object is garbage much sooner than you expect. More specifically, an object can become garbage while a method is executing on the object, which is contrary to most developers' expectations. Chris Brumme [explains](http://blogs.msdn.com/cbrumme/archive/2003/04/19/51365.aspx) the issue on his blog. I've taken Chris's code and expanded it into a full app that you can play with if you want to prove to yourself that this is a real problem:

using System;

using System.Runtime.InteropServices;

class Win32

{

[DllImport("kernel32.dll")]

public static extern IntPtr CreateEvent( IntPtr lpEventAttributes,

bool bManualReset,bool bInitialState, string lpName);

[DllImport("kernel32.dll", SetLastError=true)]

public static extern bool CloseHandle(IntPtr hObject);

[DllImport("kernel32.dll")]

public static extern bool SetEvent(IntPtr hEvent);

}

class EventUser

{

public EventUser()

{

hEvent = Win32.CreateEvent( IntPtr.Zero, false, false, null );

}

~EventUser()

{

Win32.CloseHandle( hEvent );

Console.WriteLine("EventUser finalized");

}

public void UseEvent()

{

UseEventInStatic( this.hEvent );

}

static void UseEventInStatic( IntPtr hEvent )

{

//GC.Collect();

bool bSuccess = Win32.SetEvent( hEvent );

Console.WriteLine( "SetEvent " + (bSuccess ? "succeeded" : "FAILED!") );

}

IntPtr hEvent;

}

class App

{

static void Main(string[] args)

{

EventUser eventUser = new EventUser();

eventUser.UseEvent();

}

}

If you run this code, it'll probably work fine, and you'll get the following output:

SetEvent succeeded

EventDemo finalized

However, if you uncomment the GC.Collect() call in the UseEventInStatic() method, you'll get this output:

EventDemo finalized

SetEvent FAILED!

(Note that you need to use a release build to reproduce this problem.)

So what's happening here? Well, at the point where UseEvent() calls UseEventInStatic(), a copy is taken of the hEvent field, and there are no further references to the EventUser object anywhere in the code. So as far as the runtime is concerned, the EventUser object is garbage and can be collected. Normally of course the collection won't happen immediately, so you'll get away with it, but sooner or later a collection will occur at the wrong time, and your app will fail.

A solution to this problem is to add a call to GC.KeepAlive(this) to the end of the UseEvent method, as [Chris explains](http://blogs.msdn.com/cbrumme/archive/2003/04/19/51365.aspx).

**Why to finalize objects?**

During a collection, the garbage collector will not free an object if it finds one or more references to the object in managed code. However, the garbage collector does not recognize references to an object from unmanaged code, and might free objects that are being used exclusively in unmanaged code unless explicitly prevented from doing so. The [KeepAlive](http://msdn.microsoft.com/en-us/library/system.gc.keepalive(VS.80).aspx) method provides a mechanism that prevents the garbage collector from collecting objects that are still in use in unmanaged code.

Aside from managed memory allocations, implementations of the garbage collector do not maintain information about resources held by an object, such as file handles or database connections. When a type uses unmanaged resources that must be released before instances of the type are reclaimed, the type can implement a finalizer.

In most cases, finalizers are implemented by overriding the [Object.Finalize](http://msdn.microsoft.com/en-us/library/system.object.finalize(VS.80).aspx) method; however, types written in C# or C++ implement destructors, which compilers turn into an override of **Object.Finalize**. In most cases, if an object has a finalizer, the garbage collector calls it prior to freeing the object. However, the garbage collector is not required to call finalizers in all situations; for example, the [SuppressFinalize](http://msdn.microsoft.com/en-us/library/system.gc.suppressfinalize(VS.80).aspx) method explicitly prevents a finalizer from being called. Also, the garbage collector is not required to use a specific thread to finalize objects, or guarantee the order in which finalizers are called for objects that reference each other but are otherwise available for garbage collection.

In scenarios where resources must be released at a specific time, classes can implement the [IDisposable](http://msdn.microsoft.com/en-us/library/system.idisposable(VS.80).aspx) interface, which contains the [IDisposable.Dispose](http://msdn.microsoft.com/en-us/library/system.idisposable.dispose(VS.80).aspx) method that performs resource management and cleanup tasks. Classes that implement **Dispose** must specify, as part of their class contract, if and when class consumers call the method to clean up the object. The garbage collector does not, by default, call the **Dispose** method; however, implementations of the **Dispose** method can call methods in the **GC** class to customize the finalization behavior of the garbage collector.

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

*Generations are the division of objects on the managed heap used by the garbage collector*. This mechanism allows the garbage collector to perform highly optimized garbage collection. The unreachable objects are placed in generation 0, the reachable objects are placed in generation 1, and the objects that survive the collection process are promoted to higher generations.

**What is object aging?**

It is recommended, but not required, that garbage collectors support object aging using generations. *A generation is a unit of measure of the relative age of objects in memory. The generation number, or age, of an object indicates the generation to which an object belongs.* Objects created more recently are part of newer generations, and have lower generation numbers than objects created earlier in the application life cycle. Objects in the most recent generation are in generation zero.

*Notes to Implementers*This implementation of the garbage collector supports three generations of objects. [MaxGeneration](http://msdn.microsoft.com/en-us/library/system.gc.maxgeneration(VS.80).aspx) is used to determine the maximum generation number supported by the system. Object aging allows applications to target garbage collection at a specific set of generations rather than requiring the garbage collector to evaluate all generations.

For example:

using System;

namespace GCCollectIntExample

{

class MyGCCollectClass

{

private const long maxGarbage = 1000;

static void Main()

{

MyGCCollectClass myGCCol = new MyGCCollectClass();

// Determine the maximum number of generations the system

// garbage collector currently supports.

Console.WriteLine("The highest generation is {0}", GC.MaxGeneration);

myGCCol.MakeSomeGarbage();

// Determine which generation myGCCol object is stored in.

Console.WriteLine("Generation: {0}", GC.GetGeneration(myGCCol));

// Determine the best available approximation of the number

// of bytes currently allocated in managed memory.

Console.WriteLine("Total Memory: {0}", GC.GetTotalMemory(false));

// Perform a collection of generation 0 only.

GC.Collect(0);

// Determine which generation myGCCol object is stored in.

Console.WriteLine("Generation: {0}", GC.GetGeneration(myGCCol));

Console.WriteLine("Total Memory: {0}", GC.GetTotalMemory(false));

// Perform a collection of all generations up to and including 2.

GC.Collect(2);

// Determine which generation myGCCol object is stored in.

Console.WriteLine("Generation: {0}", GC.GetGeneration(myGCCol));

Console.WriteLine("Total Memory: {0}", GC.GetTotalMemory(false));

Console.Read();

}

void MakeSomeGarbage()

{

Version vt;

for(int i = 0; i < maxGarbage; i++)

{

// Create objects and release them to fill up memory

// with unused objects.

vt = new Version();

}

}

}

}

**When should you call the garbage collector in .NET?**  
As a good rule, you should not call the garbage collector.  However, you could call the garbage collector when you are done using a large object (or set of objects) to force the garbage collector to dispose of those very large objects from memory.  However, this is usually not a good practice.

**How do I use the 'using' keyword with multiple objects?**

You can nest using statements, like this:

using( obj1 )

{ using( obj2 )

{

...

}

}

However consider using this more aesthetically pleasing (but functionally identical) formatting:

using( obj1 )

using( obj2 )

{

...

}

**What do you mean memory leaks?**

*A memory leak is a particular type of unintentional memory consumption by a computer program where the program fails to release memory when no longer needed. This condition is normally the result of a bug in a program that prevents it from freeing up memory that it no longer needs.*

**Why languages that provide automatic memory management, like Java, C# are not immune to memory leaks?**

The garbage collector recovers only memory that has become unreachable. It does not free memory that is still reachable. In .NET, this means that objects reachable by at least one reference won't be released by the garbage collector.

**Give examples of resources that can cause memory leak.**

* The system uses **User objects** to support window management. They include: Accelerator tables, Carets, Cursors, Hooks, Icons, Menus and Windows.
* **GDI objects** support graphics: Bitmaps, Brushes, Device Contexts (DC), Fonts, Memory DCs, Metafiles, Palettes, Pens, Regions, etc.
* **Kernel objects** support memory management, process execution, and inter-process communications (IPC): Files, Processes, Threads, Semaphores, Timers, Access tokens, Sockets, etc.

**How to eradicate leaks? Or how would you troubleshoot a memory leak in a .NET application?**

There are usually three steps in leak eradication:

1. *Detect a leak*
2. *Find the leaking resource*
3. *Decide where and when the resource should be released in the source code*

**How to detect leaks and find the leaking resources?**

The most direct way to "detect" leaks is to suffer from them.

You won't likely see your computer run out of memory. "Out of memory" messages are quite rare. This is because when operating systems run out of RAM, they use hard disk space to extend the memory workspace (this is called virtual memory).

What you're more likely to see happen are "out of handles" exceptions in your Windows graphical applications. The exact exception is either a *System.ComponentModel.Win32Exception* or a *System.OutOfMemoryException* with the following message: ["Error creating window handle"](http://weblogs.asp.net/fmarguerie/archive/2009/08/07/cannot-create-window-handle-desktop-heap.aspx). This happens when two many resources are consumed at the same time, most likely because of objects not being released while they should.

Another thing you may see even more often is your application or the whole computer getting slower and slower. This can happen because your machine is simply getting out of resources.

Let me make a blunt assertion: most applications leaks. Most of the time it's not a problem because the issues resulting from leaks show up only if you use applications intensively and for a long period of time.

If you suspect that objects are lingering in memory while they should have been released, the first thing you need to do is to find what these objects are.

*To find leaking objects/resources:*

Look for unexpected and lingering high level objects or root containers with your favorite memory profiler. In project X, this can be objects such as LayoutView instances (we use [the MVP pattern](http://msdn.microsoft.com/en-us/magazine/cc188690.aspx) with CAB/SCSF). In your case, it all depends on what the root objects are.

The next step is to find why these objects are being kept in memory while they shouldn't be. This is where debuggers and profilers really help. They can show you how objects are linked together.

Your goal should be to find the root reference. Don't stop at the first object you'll find, but ask yourself why this object is kept in memory.

**What are the common memory leak causes?**

There is only a small set of causes. That means that you won't have to look for a lot of cases when you'll try to solve a leak.

These are:

* Static references
* Event with missing unsubscription
* Static event with missing unsubscription
* Dispose method not invoked
* Incomplete Dispose method

The culprits that are just listed concern your applications, but you should understand that leaks can happen in other pieces of .NET code that your applications rely on. There can actually be bugs in libraries you use.

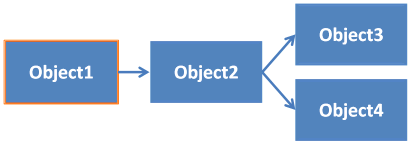
Let's take an example. In project X, a third-party visual controls suite is used to build the GUI. One of these controls is used to display toolbars. The way it is used is via a component that manages a list of toolbars. This works fine, except that even though the toolbar class implements IDisposable, the manager class never calls the Dispose method on the toolbars it manages. This is a bug. Fortunately, a workaround is easy to find: just call Dispose by ourselves on each toolbar. Unfortunately, this is not enough because the toolbar class itself is buggy: it does not dispose the controls (buttons, labels, etc.) it contains. Again, the solution is to dispose each control the toolbar contain, but that's not so easy this time because each sub-control is different.  
Anyway, this is just a specific example. Any libraries and components you use may cause leaks in your applications.

**Explain the common causes of memory leak in an application.**

### Static references

If an object is referenced by a static field, then it will never be released. This is also true with such things as singletons. Singletons are often static objects, and if it's not the case, they are usually long-lived objects anyway.

This may be obvious, but keep in mind that not only direct references are dangerous. The real danger comes from indirect references. In fact, you must pay attention to the chains of references. What counts is the root of each chain. If the root is static, then all the objects down the chain will stay alive forever.



If Object1 on the above diagram is static, and most likely long-lived, then all the other objects down the reference chain will be kept in memory for a long time. The danger is that the chain can be too long to realize that the root of the chain is static. If you care about only one level of depth, you will consider that Object3 and Object4 will go away when Object2 goes away. That's correct, for sure, but you need to take into account the fact that they may never go away because Object1 keeps the whole object graph alive.

Be very careful with all kinds of statics. Avoid them if possible. If not, pay careful attention to the objects your static objects and singletons keep in memory.

A specific kind of risky statics are static events. I'll cover them just after I cover events in general.

### Events, or the "lapsed listener" issue

A child form is subscribing to an event of the main form to get notified when the opacity changes.

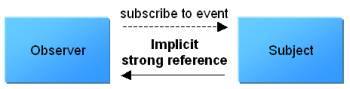
mainForm.OpacityChanged += mainForm\_OpacityChanged;

The problem is that the subscription to the OpacityChanged event creates a reference from the main form to the child form.

Here is how objects are connected after the subscription:



See [this post of mine](http://weblogs.asp.net/fmarguerie/archive/2004/07/27/198489.aspx) to learn more about events and references. Here is a figure from this post that shows the "back" reference from a subject to its observers:



MainForm keeps a reference to EventForm. This means that all the child forms that you'll open will stay in memory while the application is alive, even if you don't use them anymore.

The simplest solution is to remove the reference by having the child forms unsubscribe from the main form's event when they get disposed:

mainForm.OpacityChanged -= mainForm\_OpacityChanged; };

Nota Bene: We have a problem here because the MainForm object remains alive until the application is shut down. Interconnected objects with shorter lifetimes may not cause issues with memory. Any isolated graph of objects gets unloaded automatically from memory by the garbage collector. An isolated graph of objects is formed by two objects that only reference one another, or by a group of connected objects without any external reference.

Another solution would be to use weak delegates, which are based on weak references. I touch this subject in [my post about events and references](http://weblogs.asp.net/fmarguerie/archive/2004/07/27/198489.aspx). Several articles on the Web demonstrate how to put this into action. [Here is a good one](http://www.codeproject.com/KB/cs/WeakEvents.aspx), for example. Most of the solutions you'll find are based on the [WeakReference](http://msdn.microsoft.com/en-us/library/system.weakreference.aspx) class.

Note that a solution for this exists in WPF, in the form of [the WeakEvent pattern](http://msdn.microsoft.com/en-us/library/aa970850.aspx).

Event handlers with missing unsubscriptions from events on static or long-lived objects are a problem. Another one is static events.

### Static events

Let's see an example directly:

SystemEvents.UserPreferenceChanged += SystemEvents\_UserPreferenceChanged;

This is similar to the previous case, except that this time we subscribe to a static event. Since the event is static, the listener form object will never get released.

Again, the solution is to unsubscribe when we're done:

SystemEvents.UserPreferenceChanged -= SystemEvents\_UserPreferenceChanged;

### Dispose method not invoked

You've paid attention to events, static or not? Great, but that's not enough. You can still get lingering references even with correct cleanup code. This happens sometimes simply because this cleanup code doesn't get invoked...

Using the Dispose method or the Disposed event to unsubscribe from event and to release resources is a best practice, but it's useless if Dispose doesn't get called.

Let's take an interesting example. Here is sample code that creates a context-menu for a form.

ContextMenuStrip menu = new ContextMenuStrip();

menu.Items.Add("Item 1");

menu.Items.Add("Item 2");

this.ContextMenuStrip = menu;

After the form has been closed and disposed, the ContextMenuStrip is still alive in memory! Note: To see the problem happen, show the context-menu with a right-click before closing the form.

Again this is a problem with static events. The solution is the same as usual:

ContextMenuStrip.Dispose();

I guess you start to understand how events can be dangerous in .NET if you don't pay careful attention to them and the references they imply.  
What I want to stress here is that it's easy to introduce a leak with just a line of code. Would you have thought about potential memory leaks when creating a context-menu?

It's even worst than what you imagine. Not only the ContextMenuStrip is maintained alive, but it maintains the complete form alive with it! The ContextMenuStrip references the form. The result is that the form will be alive as long as the ContextMenuStrip is. Oh, of course you should not forget that while the form is alive, it maintains itself a whole set of objects alive - the controls and components it contains, to start with.

This is something that I find important enough to warrant a big warning. A small object can potentially maintain big graphs of other objects in memory. I've seen this happen all the time in project X. This is the same with water: a small leak can cause big damages.

Because a single control points to its parent or to events of its parent, it has the potential to keep a whole chain of container controls alive if it hasn't been disposed. And of course, all the other controls contained in these containers are also kept in memory. This can for example lead to a complete form and all its content that remain in memory for ever (at least until the application stops).

At this point, you may be wondering if this problem always exists with ContextMenuStrip. It doens't. most of the time, you create ContextMenuStrips with the designer directly on their form, and in this case Visual

**Incomplete Dispose method**

It's great to have your classes implement the IDisposable interface and to include calls to Dispose and *using* blocks all over your code, but that's really useful only if the Dispose methods are implemented correctly.

This remark may seem a bit stupid, but if I make it it's because I've seen many cases where the code of Dispose methods was not complete.

You know how it happens. You create your classes; you have them implement IDisposable; you unsubscribe from events and release resources in Dispose; and you call Dispose everywhere. That's fine, until later on you have one of your classes subscribe to a new event or consume a new resource. It's easy to code, you're eager to finish coding and to test your code. It runs fine and you're happy with it. You checkin. Great! But... oops, you've forgotten to update Dispose to release everything. It happens all the time.

I won't provide an example for this. It should be pretty obvious.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Threading**

**What is Multi-tasking?**It’s a feature of modern operating systems with which we can run multiple programs at same time. Example Word, Excel, etc.   
  
**What is Multi-threading?**Multi-threading forms subset of Multi-tasking instead of having to switch between programs this feature switches between different parts of the same program. Example you are writing in word and at the same time word is doing a spell check in background.   
  
**What is a thread?**

*A thread is an independent execution path, able to run simultaneously with other threads.* A thread is the basic unit to which the operating system allocates processor time.

A C# program starts in a single thread created automatically by the CLR and operating system

(the "main" thread), and is made multi-threaded by creating additional threads.

*The CLR assigns each thread its own memory stack so that local variables are kept separate*. In the

next example, we define a method with a local variable, then call the method simultaneously on the

main thread and a newly created thread:

static void Main() {

new Thread (Go).Start(); // Call Go() on a new thread

Go(); // Call Go() on the main thread

}

static void Go() {

// Declare and use a local variable - 'cycles'

for (int cycles = 0; cycles < 5; cycles++) Console.Write ('?');

}

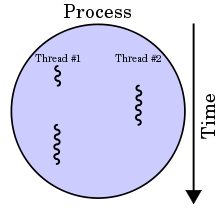
??????????

A separate copy of the **cycles** variable is created on each thread's memory stack, and so the output is, predictably, ten question marks.

**What is a process in OS?**

A program in execution is a process.

**How does threads and process relate to each other?**

****

A process with two threads of execution.

A process has at least one thread.

**What is the difference between thread and process?**

| **Process** | **Thread** |
| --- | --- |
| Processes are typically independent. | Threads exist as subsets of a process. A thread is contained inside a process. |
| Processes carry considerable state information. They don’t share resources. | Multiple threads within a process share state as well as [memory](http://storage) and other resources. |
| Processes have separate [address spaces](http://space). | Threads share their address space. |
| Processes interact only through system-provided [inter-process communication](http://communication) mechanisms. | Synchronization event is required to allow one thread to communicate an event to another. |
| Context switching between processes is slower as switching from one process to another requires a certain amount of time for doing the administration - saving and loading registers and memory maps, updating various tables and list etc. | [Context switching](http://switching) between threads in the same process is typically faster than context switching between processes. |

A *context**switch* is the [computing](http://computing) process of storing and restoring the state (context) of a [CPU](http://unit) so that execution can be resumed from the same point at a later time. This enables multiple processes to share a single CPU resource. The context switch is an essential feature of a [multitasking](http://multitasking) [operating system](http://system). Context switches are usually computationally intensive and much of the design of [operating systems](http://system) is to optimize the use of context switches. A context switch can mean a [register](http://register) context switch, a task context switch, a thread context switch, or a process context switch.

**What are the costs of creating threads?**

Before creating threads, you must keep the costs in mind. In many cases, you may find that the cost is low compared to the benefit. The following list enumerates some of the costs incurred when creating multiple threads in your process.

* Memory is needed for the structures required by threads.
* An application that has a large number of threads consumes extra CPU time in keeping track of those threads.
* An application is responsible for synchronizing access to shared resources by multiple threads. This is true for system resources (such as communications ports or disk drives), handles to resources shared by multiple processes (such as file or pipe handles), or the resources of a single process (such as global variables accessed by multiple threads). If you don't synchronize multiple threads properly (in the same or in different processes), you can run into some nasty problems, including the dreaded deadlock and race conditions.
* Because all threads of a process share the same address space and can access the process's global variables, an application must also synchronize access to these global variables. This means that the developer must decide what data can be process-specific and what data is thread-specific.

**What are the different ways through which threads can share data? Is it a problem if data is shared between threads? If yes, how can it be solved?**

* Threads share data if they have a common reference to the same object instance. Here's an example:

class ThreadTest {

bool done;

static void Main() {

ThreadTest tt = new ThreadTest(); // Create a common instance

new Thread (tt.Go).Start();

tt.Go();

}

// Note that Go is now an instance method

void Go() {

if (!done) { done = true; Console.WriteLine ("Done"); }

}

}

Because both threads call **Go()** on the same **ThreadTest** instance, they share the **done** field. This

results in "Done" being printed once instead of twice:

Done

* Static fields offer another way to share data between threads. Here's the same example with **done** as a static field:

class ThreadTest {

static bool done; // Static fields are shared between all threads

static void Main() {

new Thread (Go).Start();

Go();

}

static void Go() {

if (!done) { done = true; Console.WriteLine ("Done"); }

}

}

Both of these examples illustrate another key concept – that of *thread safety* (or, rather, lack of it!)

The output is actually indeterminate: it's possible (although unlikely) that "Done" could be printed

twice. If, however, we swap the order of statements in the **Go** method, then the odds of "Done" being printed twice go up dramatically:

static void Go() {

if (!done) { Console.WriteLine ("Done"); done = true; }

}

Done

Done (usually!)

The problem is that one thread can be evaluating the **if** statement right as the other thread is executing the **WriteLine** statement – before it's had a chance to set **done** to **true**.

**What is multithreading?**

*Multithreading is a technique that can be used to perform time consuming tasks in a separate additional thread other than the main application thread.*

When you, for example, have a time-consuming function, you may need to call this function as a response of a button click. Now, instead of freezing all your application waiting for this function to return / to finish, you can create a new thread and assign this function to. When you do this, your application interface will not be blocked and you can use it to perform other tasks. At the same time, your time-consuming task is being carried out in the background.

You can think of it as the two threads: the main one, and the newly created one. Both are running in parallel, and this improves the performance and responsiveness of your application.

**What are the advantages and disadvantages of using multithreading?**

Despite improving your application's performance, and avoiding unresponsive user interface, multithreading has the following disadvantages:

* There is a runtime overhead associated with creating and destroying threads. When your application creates and destroys threads frequently, this overhead affects the overall application performance.
* Having too many threads running at the same time decreases the performance of your entire system. This is because your system is attempting to give each thread a time slot to operate inside.
* You should design your application well when you are going to use multithreading, or otherwise your application will be difficult to maintain and extend.
* You should be careful when you implement a multithreading application, because threading bugs are difficult to debug and resolve.

**Notes:**

* Each time a thread is created, a certain amount of memory is consumed to hold this thread context information. Hence, the number of threads that can be created is limited by the amount of available memory.
* More threads does not mean a faster responsive application, instead it can decrease the performance of your application.

**How a C# application can become multithreaded?**

A C# application can become multi-threaded in two ways: either by

* explicitly creating and running additional threads,
* or using a feature of the .NET framework that implicitly creates threads – such as BackgroundWorker, thread pooling, a threading timer, a Remoting server, or a Web Services or ASP.NET application. In these latter cases, one has no choice but to embrace multithreading.

**What are background and foreground threads?**

By default, threads are foreground threads, meaning they keep the application alive for as long as any one of them is running. C# also supports background threads, which don’t keep the application alive on their own – terminating immediately once all foreground threads have ended.

A thread's **IsBackground** property controls its background status.

**By default whether threads are background or foreground threads?**

Threads are foreground threads, by default.

**How having worker threads as background threads can be beneficial? What is the problem of having worker threads as foreground threads?**

**What is thread priority?**

A thread’s **Priority** property determines how much execution time it gets relative to other active

threads in the same process, on the following scale:

public enum ThreadPriority

{ Lowest, BelowNormal, Normal, AboveNormal, Highest }

This becomes relevant only when multiple threads are simultaneously active.  
**What is the outcome of an unhandled exception in the thread?**

In the .NET Framework version 2.0, an unhandled exception in either foreground or background threads results in termination of the application.

**What is the output of below program? Or what happens when an exception is thrown in a different thread?**

**using System;**

**using System.Threading;**

**namespace ConsoleApplication1**

**{**

**class Program**

**{**

**public static void Main()**

**{**

**try**

**{**

**new Thread(Go).Start();**

**}**

**catch (Exception ex)**

**{**

**// We'll never get here!**

**Console.WriteLine("Exception!");**

**}**

**}**

**static void Go() { throw null; }**

**}**

**}**

The **try/catch** statement here is effectively useless, and the newly created thread will be

encumbered with an unhandled **NullReferenceException**. This behavior makes sense when you

consider a thread has an independent execution path.

**What is the remedy for exception thrown by a different thread?**

The remedy is for thread entry methods to have their own exception handlers:

public static void Main() {

new Thread (Go).Start();

}

static void Go() {

try {

...

throw null; // this exception will get caught below

...

}

catch (Exception ex) {

Typically log the exception, and/or signal another thread

that we've come unstuck

...

}

One can cut the work by using a wrapper or helper class to perform the job, such as *BackgroundWorker.*

**Why a try/catch block is required in every thread entry method – at least in production applications?**

An unhandled exception on any thread shuts down the whole application, meaning ignoring the exception is generally not an option. Hence a **try/catch** block is required in every thread entry method – at least in production applications – in order to avoid unwanted application shutdown in case of an unhandled exception. This can be somewhat cumbersome – particularly for Windows Forms programmers, who commonly use the "global" exception handler, as follows:

static class Program {

static void Main() {

Application.ThreadException += HandleError;

Application.Run (new MainForm());

}

static void HandleError (object sender,

ThreadExceptionEventArgs e) {

Log exception, then either exit the app or continue...

}

}

**Are all exceptions caught by the central exception handler - Application.ThreadException?**

No. Exceptions thrown on worker threads are a good example of exceptions not caught by **Application.ThreadException** (the code inside the **Main** method is another – including the main

form's constructor, which executes before the Windows message loop begins).

**What is a blocked thread? What is the new state after a thread is blocked?**

*When a thread waits or pauses as a result of using the constructs like sleep and join, it's said to*

*be blocked.*

Once blocked, a thread immediately relinquishes its allocation of CPU time, adds

**WaitSleepJoin** to its ThreadState property, and doesn’t get re-scheduled until unblocked.

**How can a blocked thread be unblocked?**

Unblocking happens in one of four ways (the computer's power button doesn't count!):

by the blocking condition being satisfied

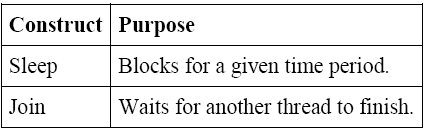
by the operation timing out (if a timeout is specified)

by being interrupted via Thread.Interrupt

by being aborted via Thread.Abort

A thread is not deemed blocked if its execution is paused via the (deprecated) Suspend method.

**What are the simple blocking methods?**

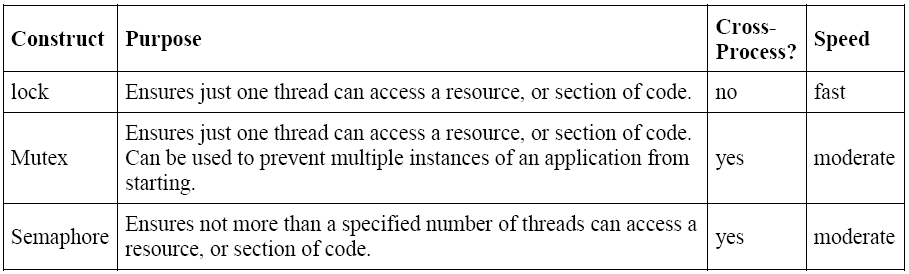
****

**What is synchronization? Why synchronizing is required?**

**What is mutual exclusion?**

**What is a deadlock? How can deadlocks happen? Give an example.**

**What are the different locking constructs?**

****

**Why lock must not be obtained on a Value Type?**

Do NOT attempt to lock a field that's not a reference (object) type, such as int/Integer. You'll get a compiler error saying “'int' is not a reference type as required by the lock statement”.

**Why lock must not be obtained on a public type or instances beyond your code's control? Or why lock should be avoided on constructs such as lock (this), lock (typeof (MyType)), and lock ("myLock")?**

* lock (this) is a problem if the instance can be accessed publicly.
* lock (typeof (MyType)) is a problem if MyType is publicly accessible.
* lock(“myLock”) is a problem because any other code in the process using the same string, will share the same lock.

Best practice is to define a **private** object to lock on, or a **private static** object variable to protect data common to all instances.

lock(this) can be problematic if the instance can be accessed publicly, because code beyond your control may lock on the object as well. This could create deadlock situations where two or more threads wait for the release of the same object. Locking on a public data type, as opposed to an object, can cause problems for the same reason. It breaks encapsulation and can lead to deadlocks. Locking on literal strings is especially risky because literal strings are *interned* by the common language runtime (CLR). This means that there is one instance of any given string literal for the entire program, the exact same object represents the literal in all running application domains, on all threads. As a result, a lock placed on a string with the same contents anywhere in the application process locks all instances of that string in the application. As a result, it is best to lock a private or protected member that is not interned. Some classes provide members specifically for locking. The *Array* type, for example, provides *SyncRoot*. Many collection types provide a *SyncRoot* member as well.

**Are lock and monitor same?**

Like the **lock** keyword, monitors prevent blocks of code from simultaneous execution by multiple threads. The [Enter](http://msdn.microsoft.com/en-us/library/de0542zz(VS.80).aspx) method allows one and only one thread to proceed into the following statements; all other threads are blocked until the executing thread calls [Exit](http://msdn.microsoft.com/en-us/library/system.threading.monitor.exit(VS.80).aspx). This is just like using the **lock** keyword. In fact, the **lock** keyword is implemented with the [Monitor](http://msdn.microsoft.com/en-us/library/system.threading.monitor(VS.80).aspx) class. For example:

lock (x)

{

DoSomething();

}

This is equivalent to:

System.Object obj = (System.Object)x;

System.Threading.Monitor.Enter(obj);

try

{

DoSomething();

}

finally

{

System.Threading.Monitor.Exit(obj);

}

**Why lock is preferred over Monitor?**

* + *lock* is more concise,
  + *lock* insures that the underlying *monitor* is released, even if the protected code throws an exception. This is accomplished with the *finally* keyword, which executes its associated code block regardless of whether an exception is thrown.

**What is the advantage and limitation of lock and monitor?**

A lock or monitor is useful for preventing the simultaneous execution of thread-sensitive blocks of code, but these constructs do not allow one thread to communicate an event to another.

**What is the Mutex and Semaphore? In C#, where do we need to implement?**

* System.Threading.Mutex: A synchronization primitive that can also be used for interprocess synchronization.
* System.Threading.Semaphore: Limits the number of threads that can access a resource or pool of resources concurrently.

Generally you only use a Mutex across processes, e.g. if you have a resource that multiple applications must share, or if you want to build a single-instanced app (i.e. only allow 1 copy to be running at one time).

A semaphore allows you to limit access to a specific number of simultaneous threads, so that you could have, for example, a maximum of two threads executing a specific code path at a time.

**What is Mutex? What is it’s advantage over lock?**

**Mutex** provides the same functionality as C#'s lock statement, making **Mutex** mostly redundant. Its

one advantage is that it can work across multiple processes – providing a computer-wide lock rather

than an application-wide lock.

**Give an example of Mutex.**

A common use for a cross-process **Mutex** is to ensure that only instance of a program can run at a

time. Here's how it's done:

class OneAtATimePlease {

// Use a name unique to the application (eg include your company URL)

static ***Mutex*** mutex = new ***Mutex*** (false, "oreilly.com OneAtATimeDemo");

static void Main() {

// Wait 5 seconds if contended – in case another instance

// of the program is in the process of shutting down.

if (!mutex.***WaitOne*** (TimeSpan.FromSeconds (5), false)) {

Console.WriteLine ("Another instance of the app is running. Bye!");

return;

}

try {

Console.WriteLine ("Running - press Enter to exit");

Console.ReadLine();

}

finally { mutex.***ReleaseMutex***(); }

}

}

**When and how do you use semaphore?**

<http://www.albahari.com/threading/part2.aspx>

A semaphore with a capacity of one is similar to a Mutex or lock, except that the semaphore has no “owner” — it’s *thread-agnostic*. Any thread can call Release on a Semaphore, whereas with Mutex and lock, only the thread that obtained the lock can release it.

There are two functionally similar versions of this class: Semaphore and SemaphoreSlim. The latter was introduced in Framework 4.0 and has been optimized to meet the low-latency demands of [parallel programming](http://www.albahari.com/threading/part5.aspx#_Parallel_Programming). It’s also useful in traditional multithreading because it lets you specify a [cancellation token](http://www.albahari.com/threading/part3.aspx#_Cancellation_Tokens) when waiting. It cannot, however, be used for interprocess signaling.

Semaphore incurs about 1 microsecond in calling WaitOne or Release; SemaphoreSlim incurs about a quarter of that.

Semaphores can be useful in limiting concurrency — preventing too many threads from executing a particular piece of code at once. In the following example, five threads try to enter a nightclub that allows only three threads in at once:

class TheClub // No door lists!

{

static SemaphoreSlim \_sem = new SemaphoreSlim (3); // Capacity of 3

static void Main()

{

for (int i = 1; i <= 5; i++) new Thread (Enter).Start (i);

}

static void Enter (object id)

{

Console.WriteLine (id + " wants to enter");

\_sem.Wait();

Console.WriteLine (id + " is in!"); // Only three threads

Thread.Sleep (1000 \* (int) id); // can be here at

Console.WriteLine (id + " is leaving"); // a time.

\_sem.Release();

}

}

1 wants to enter

1 is in!

2 wants to enter

2 is in!

3 wants to enter

3 is in!

4 wants to enter

5 wants to enter

1 is leaving

4 is in!

2 is leaving

5 is in!

If the Sleep statement was instead performing intensive disk I/O, the Semaphorewould improve overall performance by limiting excessive concurrent hard-drive activity.

A Semaphore, if named, can span processes in the same way as a Mutex.

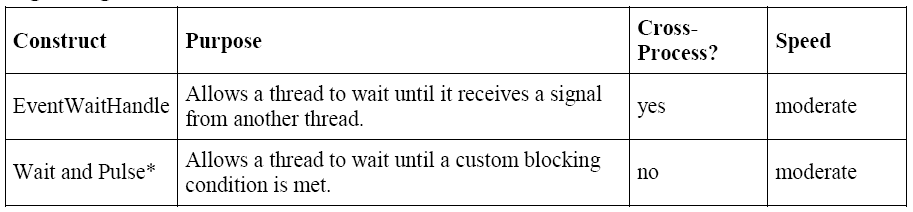
**What are synchronization events? Why is it required?**

*Synchronization events* are objects that have one of two states, signaled and un-signaled, that can be used to activate and suspend threads. Threads can be suspended by being made to wait on a synchronization event that is unsignaled, and can be activated by changing the event state to signaled. If a thread attempts to wait on an event that is already signaled, then the thread continues to execute without delay.

There are two kinds of synchronization events: [AutoResetEvent](http://msdn.microsoft.com/en-us/library/system.threading.autoresetevent(VS.80).aspx), and [ManualResetEvent](http://msdn.microsoft.com/en-us/library/system.threading.manualresetevent(VS.80).aspx). Both are derived from *EventWaitHandle class***.**

Synchronization event is required to allow one thread to communicate an event to another.

**What are the available signaling constructs?**

****

**What is the difference between AutoResetEvent, and ManualResetEvent?**

They differ only in that **AutoResetEvent** changes from signaled to unsignaled automatically any time it activates a thread. Conversely, a **ManualResetEvent** allows any number of threads to be activated by its signaled state, and will only revert to an unsignaled state when its *Reset* method is called.

**How can threads be made to wait on events? Give an example.**

Threads can be made to wait on events by calling one of the wait methods, such as [WaitOne](http://msdn.microsoft.com/en-us/library/58195swd(VS.80).aspx), [WaitAny](http://msdn.microsoft.com/en-us/library/tdykks7z(VS.80).aspx), or [WaitAll](http://msdn.microsoft.com/en-us/library/z6w25xa6(VS.80).aspx). [System.Threading.WaitHandle.WaitOne](http://msdn.microsoft.com/en-us/library/system.threading.waithandle.waitone(VS.80).aspx) causes the thread to wait until a single event becomes signaled, [System.Threading.WaitHandle.WaitAny](http://msdn.microsoft.com/en-us/library/system.threading.waithandle.waitany(VS.80).aspx) blocks a thread until one or more indicated events become signaled, and [System.Threading.WaitHandle.WaitAll](http://msdn.microsoft.com/en-us/library/system.threading.waithandle.waitall(VS.80).aspx) blocks the thread until all of the indicated events become signaled.

An event becomes signaled when its [Set](http://msdn.microsoft.com/en-us/library/system.threading.eventwaithandle.set(VS.80).aspx) method is called.

*Example*

class BasicWaitHandle

{

static ***EventWaitHandle*** wh = new ***AutoResetEvent*** (false);

static void Main() {

new Thread (Waiter).Start();

Thread.Sleep (1000); // Wait for some time...

wh.***Set***(); // OK - wake it up

}

static void Waiter() {

Console.WriteLine ("Waiting...");

wh.***WaitOne***(); // Wait for notification

Console.WriteLine ("Notified");

}

}

Waiting... (pause) Notified.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**What is Apartment threading? Why do we need in .NET?**

*Apartment threading* is an automatic thread-safety regime, closely allied to COM – Microsoft's

legacy Component Object Model. While .NET largely breaks free of legacy threading models, there

are times when it still crops up because of the need to interoperate with older APIs. Apartment

threading is most relevant to Windows Forms, because much of Windows Forms uses or wraps the

long-standing Win32 API – complete with its apartment heritage.

**What is an apartment? What is the default apartment for a thread created in .NET? How to assign explicitly a** **.NET thread to an apartment?**

*An apartment is a logical "container" for threads*. Apartments come in two sizes – "single" and

"multi". A single-threaded apartment contains just one thread; multi-threaded apartments can contain any number of threads. The single-threaded model is the more common and interoperable of the two.

A .NET thread is automatically assigned an apartment upon entering apartment-savvy Win32 or

legacy COM code. By default, it will be allocated a multi-threaded apartment, unless one requests a

single-threaded apartment as follows:

Thread t = new Thread (...);

t.SetApartmentState (ApartmentState.STA);

One can also request that the main thread join a single-threaded apartment using the **STAThread**

attribute on the main method:

class Program {

[STAThread]

static void Main() {

...

**Why a Windows Forms program should have the [STAThread] attribute on its main method?**

The types in the *System.Windows.Forms*namespace extensively call Win32 code designed to work

in a single-threaded apartment. For this reason, a Windows Forms program should have the

*[STAThread]*attribute on its main method, otherwise one of two things will occur upon reaching

Win32 UI code:

it will marshal over to a single-threaded apartment

it will crash

**Can we call a method or property on a control from any thread other than the one that created it in a multi-threaded application? If no, what are the solutions to manage worker threads in Windows Forms and WPF applications?**

No. In a multi-threaded Windows Forms application, it's illegal to call a method or property on a control from any thread other than the one that created it. There are two solutions using:

* *Control.Invoke*or *Control.BeginInvoke*method: All cross-thread calls must be explicitly marshaled to the thread that created the control (usually the main thread), using the *Control.Invoke*or *Control.BeginInvoke*method. One cannot rely on automatic marshalling because it takes place too late – only when execution gets well into unmanaged code, by which time plenty of internal .NET code may already have run on the "wrong" thread – code which is not thread-safe.
* *BackgroundWorker* class: An excellent solution to managing worker threads in Windows Forms and WPF applications is to use *BackgroundWorker*. This class wraps worker threads that need to report progress and completion, and automatically calls *Control.Invoke*or *Dispatcher.Invoke*as required.

**I receive this exception message: "System.InvalidOperationException: Cross-thread operation not valid: Control [your\_control\_name\_here] accessed from a thread other than the thread it was created on". How can it be fixed?**Yes, you can fix it. The exception means that your program accessed a GUI element from a thread other than the main thread, and that's a no-no. Use Control.Invoke (or BeginInvoke) to execute the call in the correct thread. See below example.

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Text;

using System.Windows.Forms;

using System.Threading;

namespace WindowsFormsApplication1

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e)

{

*// this is the thread that will change the label text*

Thread myThread = new Thread((ThreadStart)delegate()

{

UpdateLabel("Hello New Label!");

});

myThread.Start();

}

private void UpdateLabel(string text)

{

*// we have packaged the functionality into a method that can be //called without regard to what thread context we are in*

*//if we are not on the same thread that create the label we will //call the method again this time using Invoke()*

if (this.label1.InvokeRequired == true)

this.label1.Invoke((MethodInvoker)delegate()

{

UpdateLabel(text);

});

else

this.label1.Text = text;

}

}

}

**If we remove STAThread attribute from Main method, whether application would run?**

No

**How do you find the sum of numbers from 1 to 1 million - use a thread or without use of thread? Justify which you would choose.**

**You've a grid and 10 million records to show. The data comes from a background thread and you need to update the grid. The update of grid happens in UI thread. This is a performance bottleneck and makes the update of data slow. How to improve the performance in updating of grid?**

**Or can we create multiple UI threads in an Windows Forms or WPF application?**

**<http://eprystupa.wordpress.com/2008/07/28/running-wpf-application-with-multiple-ui-threads/>**

##### Introduction

It’s a good, time-proven practice to perform long, CPU intensive tasks on some sort of a background thread to improve your UI thread responsiveness. Sometimes though UI-related tasks themselves can be quite expensive. WPF, for examples, forces you to do all UI work on the thread that created the UI. A very flexible WPF measure/layout paradigm for UI rendering also comes with high CPU usage cost. In a very UI intensive application (for example, trading app with about ten windows showing real-time montage and blotter data) simply the cost of generating and laying out visuals can become too high for a single thread to keep up. When your UI thread saturates individual windows may start skip rendering cycles, become slow to response to user input, or even freeze. If your UI thread approaches this kind of saturation you should consider creating dedicated UI threads for some (or all) of your UI-intensive windows. This post is a step by step walk-through of doing just that.

##### Creating single-threaded WPF UI application

To start lets create a basic WPF application we can work with. In Visual Studio go File/New/Project, then Visual C#, Windows, WPF Application. Name the project “WpfThreadLab” and click “OK”. The automatically generated Window1.xaml window doesn’t display anything by default, so let add something useful for out lab. We will display the ID for the thread that owns the window. First open Window1.xaml.cs and add the following code”:

using System.Threading;

using System.Windows;

namespace WpfThreadLab

{

/// <summary>

/// Interaction logic for Window1.xaml

/// </summary>

public partial class Window1 : Window

{

public Window1()

{

InitializeComponent();

Thread thread = Thread.CurrentThread;

this.DataContext = new

{

ThreadId = thread.ManagedThreadId

};

}

private void OnCreateNewWindow(

object sender,

RoutedEventArgs e)

{

Window1 w = new Window1();

w.Show();

}

}

}

Constructor creates data context containing thread’s ID that we can data bind to. We also add a click event handler that we will use to create new windows on demand. This is how corresponding Window1.xaml may look like:

<Window

x:Class="WpfThreadLab.Window1"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="Window1" Height="100" Width="200"

>

<StackPanel>

<StackPanel Orientation="Horizontal">

<TextBlock Text="Thread's ID is "/>

<TextBlock Text="{Binding ThreadId}"/>

</StackPanel>

<Button

Click="OnCreateNewWindow"

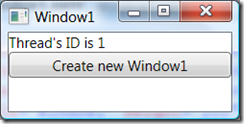
Content="Create new Window1"

/>

</StackPanel>

</Window>

If you run the application now you should see the following window:



Clicking on the button will create new clones of the window. You will see that they all belong to default UI thread with managed ID 1.

##### Adding code to create dedicated UI threads for each window

The quick and dirty solution may include the following modifications to our OnCreateNewWindow handler:

private void OnCreateNewWindow(

object sender,

RoutedEventArgs e)

{

**Thread thread = new Thread(() =>**

**{**

Window1 w = new Window1();

w.Show();

**});**

**thread.SetApartmentState(ApartmentState.STA);**

**thread.Start();**

}

The window creation code is exactly the same, except that we wrapped it in a thread delegate. This delegate is passed to a newly created thread that is started at the end of the handler. Note that we explicitly setting new thread’s apartment state to STA, this is a WPF requirement.

If you run the application now and click on the button, you will notice that a new window appears momentarily and then dies. The reason is that our newly created thread is not enabled to support WPF window infrastructure. In particular, it provides no support for Windows message pumping, and this is something we will fill in next.

##### Making a thread “UI thread”

WPF window like any other window relies on Windows message pump. In WPF word this functionality is provided by a class called Dispatcher. WPF Application object takes care of starting dispatcher for the main UI thread for us, but we have to explicitly start it for our own private UI threads. The easiest way to do it is to call static method Run() on the Dispatcher class:

private void OnCreateNewWindow(

object sender,

RoutedEventArgs e)

{

Thread thread = new Thread(() =>

{

Window1 w = new Window1();

w.Show();

**System.Windows.Threading.Dispatcher.Run();**

});

thread.SetApartmentState(ApartmentState.STA);

thread.Start();

}

The code in bold shows the modification. This blocks the thread and starts a message pump on it. If you run application now and start clicking the button you will notice that fully functional window is created every time you click, and every time on a new thread.

There is one catch to our simplified solution. Closing a particular window does NOT terminate this window’s thread dispatcher, so the thread keeps running and, after closing all windows, the process will not terminate and will become a ghost process. Simple (and not correct) solution to this is to mark our threads as background (using thread.IsBackground = true;). This will force them to terminate when main UI thread terminates. The proper implementation will gracefully shut down the dispatcher when it is no longer needed. The code below is an example of a strategy to terminate the dispatcher when window closes:

private void OnCreateNewWindow(

object sender,

RoutedEventArgs e)

{

Thread thread = new Thread(() =>

{

Window1 w = new Window1();

w.Show();

**w.Closed += (sender2, e2) =>**

**w.Dispatcher.InvokeShutdown();**

System.Windows.Threading.Dispatcher.Run();

});

thread.SetApartmentState(ApartmentState.STA);

thread.Start();

}

The code above in bold accomplishes the task of shutting down the dispatcher for window’s UI thread.

##### Conclusion

Running multiple UI threads in WPF application is actually rather trivial. In the nutshell it boils down to creating your dedicated thread and creating your windows inside this thread’s proc. Remember to conclude your UI thread with a call to WPF dispatcher. Also, decide on your “exit” strategy (often, shutting down dispatcher in response to closing the primary window for this thread).

***Thread Pool***

**What is thread pool?**

A *thread pool* is a collection of threads that can be used to perform a number of tasks in the background. This leaves the primary thread free to perform other tasks asynchronously.

**Give an example where thread pool can be used.**

Thread pools are often employed in server applications. Each incoming request is assigned to a thread from the thread pool, so the request can be processed asynchronously, without tying up the primary thread or delaying the processing of subsequent requests.

**Give example from .NET framework where it uses thread pool.**

The .NET Framework uses thread pool threads for many purposes, including asynchronous I/O completion, timer callbacks, registered wait operations, asynchronous method calls using delegates, and [System.Net](http://msdn.microsoft.com/en-us/library/system.net(VS.80).aspx) socket connections.

**Name the different ways to enter the thread pool.**

There are a number of ways to enter the thread pool:

Via the Task Parallel Library (from Framework 4.0)

By calling ThreadPool.QueueUserWorkItem

Via asynchronous delegates

Via BackgroundWorker

Please note that the following constructs use the thread pool *indirectly*:

WCF, Remoting, ASP.NET, and ASMX Web Services application servers

System.Timers.Timer and System.Threading.Timer

Framework methods that end in *Async*, such as those on WebClient (the event-based asynchronous pattern),

and most Begin*XXX* methods (the *asynchronous programming model* pattern)

PLINQ

The *Task Parallel Library* (TPL) and PLINQ are sufficiently powerful and high-level that you’ll want to use them to assist in multithreading even when thread pooling is unimportant.

**How do enter the thread pool via TPL?**

You can enter the thread pool easily using the Task classes in the Task Parallel Library. The Task classes were introduced in Framework 4.0: if you’re familiar with the older constructs, consider the nongeneric Task class a replacement for ThreadPool.QueueUserWorkItem, and the generic Task<TResult> a replacement for asynchronous delegates. The newer constructs are faster, more convenient, and more flexible than the old.

To use the nongeneric Task class, call Task.Factory.StartNew, passing in a delegate of the target method:

static void Main() // The Task class is in **System.Threading.Tasks**

{

Task.Factory.StartNew (Go);

}

static void Go()

{

Console.WriteLine ("Hello from the thread pool!");

}

Task.Factory.StartNew returns a Task object, which you can then use to monitor the task—for instance, you can wait for it to complete by calling its Wait method.

Any unhandled exceptions are conveniently rethrown onto the host thread when you call a task's Wait method.

(If you don’t call Wait and abandon the task, an unhandled exception will shut down the process as with an

ordinary thread.)

The generic Task<TResult> class is a subclass of the nongeneric Task. It lets you get a return value back from the task after it finishes executing. In the following example, we download a web page using Task<TResult>:

static void Main()

{

// Start the task executing:

Task<string> task = Task.Factory.StartNew**<string>**

( () => DownloadString ("http://www.linqpad.net") );

// We can do other work here and it will execute in parallel:

RunSomeOtherMethod();

// When we need the task's return value, we query its Result property:

// If it's still executing, the current thread will now block (wait)

// until the task finishes:

string result = **task.Result**;

}

static string DownloadString (string uri)

{

using (var wc = new System.Net.WebClient())

return wc.DownloadString (uri);

}

(The <string> type argument in boldface is for clarity: it would be *inferred* if we omitted it.)

Any unhandled exceptions are automatically rethrown when you query the task's Result property, wrapped in an AggregateException. However, if you fail to query its Result property (and don’t call Wait) any unhandled exception will take the process down.

The Task Parallel Library has many more features, and is particularly well suited to leveraging multicore processors.

**How do enter the thread pool without TPL?**

You can't use the Task Parallel Library if you're targeting an earlier version of the .NET Framework (prior to 4.0). Instead, you must use one of the older constructs for entering the thread pool: ThreadPool.QueueUserWorkItem and asynchronous delegates. The difference between the two is that asynchronous delegates let you return data from the thread. Asynchronous delegates also marshal any exception back to the caller.

***QueueUserWorkItem***

To use QueueUserWorkItem, simply call this method with a delegate that you want to run on a pooled thread:

static void Main()

{

ThreadPool.QueueUserWorkItem (Go);

ThreadPool.QueueUserWorkItem (Go, 123);

Console.ReadLine();

}

static void Go (object data) // *data* will be **null** with the first call.

{

Console.WriteLine ("Hello from the thread pool! " + data);

}

// Output:

Hello from the thread pool!

Hello from the thread pool! 123

Our target method, Go, must accept a single object argument (to satisfy the WaitCallback delegate). This provides a convenient way of passing data to the method, just like with ParameterizedThreadStart. Unlike with Task, QueueUserWorkItem doesn't return an object to help you subsequently manage execution. Also, you must explicitly deal with exceptions in the target code—unhandled exceptions will take down the program.

***Asynchronous delegates***

ThreadPool.QueueUserWorkItem doesn’t provide an easy mechanism for getting return values back from a thread after it has finished executing. Asynchronous delegate invocations (asynchronous delegates for short) solve this, allowing any number of typed arguments to be passed in both directions. Furthermore, unhandled exceptions on asynchronous delegates are conveniently rethrown on the original thread (or more accurately, the thread that calls EndInvoke), and so they don’t need explicit handling.

Don’t confuse asynchronous delegates with asynchronous methods (methods starting with *Begin* or *End*, such

as File.BeginRead/File.EndRead). Asynchronous methods follow a similar protocol outwardly, but they

exist to solve a much harder problem, which we describe in Chapter 23 of C# 4.0 in a Nutshell.

Here’s how you start a worker task via an asynchronous delegate:

1. Instantiate a delegate targeting the method you want to run in parallel (typically one of the predefined Func

delegates).

2. Call BeginInvoke on the delegate, saving its IAsyncResult return value.

BeginInvoke returns immediately to the caller. You can then perform other activities while the pooled thread is working.

3. When you need the results, call EndInvoke on the delegate, passing in the saved IAsyncResult object.

In the following example, we use an asynchronous delegate invocation to execute concurrently with the main thread, a simple method that returns a string’s length:

static void Main()

{

Func<string, int> method = Work;

IAsyncResult cookie = method.BeginInvoke ("test", null, null);

//

// ... here's where we can do other work in parallel...

//

int result = method.EndInvoke (cookie);

Console.WriteLine ("String length is: " + result);

}

static int Work (string s) { return s.Length; }

EndInvoke does three things. First, it waits for the asynchronous delegate to finish executing, if it hasn’t already. Second, it receives the return value (as well as any ref or out parameters). Third, it throws any unhandled worker exception back to the calling thread.

If the method you’re calling with an asynchronous delegate has no return value, you are still (technically)

obliged to call EndInvoke. In practice, this is open to debate; there are no EndInvoke police to administer

punishment to noncompliers! If you choose not to call EndInvoke, however, you’ll need to consider exception handling on the worker method to avoid silent failures.

You can also specify a callback delegate when calling BeginInvoke—a method accepting an IAsyncResult object that’s automatically called upon completion. This allows the instigating thread to “forget” about the asynchronous delegate, but it requires a bit of extra work at the callback end:

static void Main()

{

Func<string, int> method = Work;

method.BeginInvoke ("test", Done, method);

// ...

//

}

static int Work (string s) { return s.Length; }

static void Done (IAsyncResult cookie)

{

var target = (Func<string, int>) cookie.AsyncState;

int result = target.EndInvoke (cookie);

Console.WriteLine ("String length is: " + result);

}

The final argument to BeginInvoke is a user state object that populates the AsyncState property of IAsyncResult. It can contain anything you like; in this case, we’re using it to pass the method delegate to the completion callback, so we can call EndInvoke on it.

**How do you decide whether you need a dedicated thread or a thread pool thread?**

.NET’s thread pool has some shortcomings which can affect the choice between using a dedicated thread instead of a thread from the thread pool. It is usually said that a dedicated thread is favorable in the following scenarios:

* When a foreground thread is required: All thread pool threads are initialized as background threads.
* When it is required to have a thread with a particular priority.
* When a thread is required to be aborted prematurely
* When a thread must be placed in a single-threaded apartment (STA): All thread pool threads are set in the MTA apartment by default
* For long running tasks when the thread pool thread is often blocked for long periods (This may starve other parts of the application which rely on threads from the thread pool)

The thread pool is shared by all AppDomains in a process. Consider this if you have more than one AppDomain in your process.

**How do you decide whether you need a BackgroundWorker or ThreadPool?**

If you are using Windows Forms, prefer the BackgroundWorker for more simple threading requirements. BackgroundWorker does well with network accesses and other simple stuff. For batch processing with many processors, you need ThreadPool.

Requirement: Your program does batch processing

Consider: ThreadPool

Requirement: Your program makes many (3+) threads

Consider: ThreadPool

Requirement: Your program uses Windows Forms

Consider: BackgroundWorker

**Thread considerations.** Also, the specifics of how you use your threads can help you find the best code. This next table compares the threading scenarios and which class is best.

If: You need one extra thread

Then use: BackgroundWorker

If: You have many short-lived threads

Then use: ThreadPool

Generally, you should prefer ThreadPool when you need many threads. Threads aren't always useful, such as for I/O operations, which many computers can't multithread as well. With the era of multicore systems, we need threads and BackgroundWorker is an excellent shortcut.

**What are the advantages/benefits of thread pooling?**

Using the thread pool is the easiest technique you can use to create a multithreaded application for the following reasons:

* You do not have to create, manage, schedule, and terminate your thread, the thread pool class do all of this for you.
* There are no worries about creating too many threads and hence affecting system performance. Thread pool size is constrained by the .NET runtime. The number of threads you can use at the same time is limited.
* You need to write less code, because the .NET framework manages your thread internally with a set of well tested, and bug free routines.

**What are the limitations of using the Thread Pool?**

Despite the ease of use, the thread pool has the following limitations or disadvantages when compared to manually managing your threads:

* With thread pool, you have no control over the state and priority of the thread.
* With thread pool, you cannot give a stable identity to your thread and keep tracking it.
* When submitting a process to the thread pool, you have no idea when the process will be executed. Your process may be delayed when there are high demands on the thread pool.
* The thread pool is not suitable when you want to run two tasks or processes using two threads, and need these two tasks to be processed simultaneously in a deterministic fashion.
* The .NET framework uses the thread pool for asynchronous operations, and this places additional demand on the limited number of available threads.
* Despite of robust application isolation, there are situations where your application code can be affected by another application code.

In situations where you cannot use the thread pool because of its limitations, you can create new threads manually and manage them yourself. This technique is much more complex than using the thread pool, but it gives you more control over your threads.

**When Not to Use Thread Pool Threads?**

There are several scenarios in which it is appropriate to create and manage your own threads instead of using thread pool threads:

* You require a foreground thread.
* You require a thread to have a particular priority.
* You have tasks that cause the thread to block for long periods of time. The thread pool has a maximum number of threads, so a large number of blocked thread pool threads might prevent tasks from starting.
* You need to place threads into a single-threaded apartment. All ThreadPool threads are in the multithreaded apartment.
* You need to have a stable identity associated with the thread, or to dedicate a thread to a task.

**What are the characteristics of Thread Pool?**

* Thread pool threads are background threads.
* Each thread uses the default stack size, runs at the default priority, and is in the multithreaded apartment.
* There is only one thread pool per process.

**What is the maximum number of Thread Pool Threads?**

The number of operations that can be queued to the thread pool is limited only by available memory; however, the thread pool limits the number of threads that can be active in the process simultaneously. By default, the limit is 25 worker threads per CPU and 1,000 I/O completion threads.

You can control the maximum number of threads by using the [*GetMaxThreads*](http://msdn.microsoft.com/en-us/library/system.threading.threadpool.getmaxthreads(VS.80).aspx) and [*SetMaxThreads*](http://msdn.microsoft.com/en-us/library/system.threading.threadpool.setmaxthreads(VS.80).aspx) methods.

**What is the minimum number of Idle Threads?**

The thread pool also maintains a minimum number of available threads, even when all threads are idle, so that queued tasks can start immediately. Idle threads in excess of this minimum are terminated to save system resources. By default, one idle thread is maintained per processor.

The thread pool has a built-in delay (half a second in the .NET Framework version 2.0) before starting new idle threads. If your application periodically starts many tasks in a short time, a small increase in the number of idle threads can produce a significant increase in throughput. Setting the number of idle threads too high consumes system resources needlessly.

You can control the number of idle threads maintained by the thread pool by using the [*GetMinThreads*](http://msdn.microsoft.com/en-us/library/system.threading.threadpool.getminthreads(VS.80).aspx) and [*SetMinThreads*](http://msdn.microsoft.com/en-us/library/system.threading.threadpool.setminthreads(VS.80).aspx) methods.

**How to use the Thread Pool?**

You use the thread pool by calling [*ThreadPool.QueueUserWorkItem*](http://msdn.microsoft.com/en-us/library/system.threading.threadpool.queueuserworkitem(VS.80).aspx) from managed code (or *CorQueueUserWorkItem* from unmanaged code) and passing a [*WaitCallback*](http://msdn.microsoft.com/en-us/library/system.threading.waitcallback(VS.80).aspx) delegate representing the method that performs the task. You can also queue work items that are related to a wait operation by using the [*ThreadPool.RegisterWaitForSingleObject*](http://msdn.microsoft.com/en-us/library/system.threading.threadpool.registerwaitforsingleobject(VS.80).aspx) method and passing a [*WaitHandle*](http://msdn.microsoft.com/en-us/library/system.threading.waithandle(VS.80).aspx) that, when signaled or when timed out, raises a call to the method represented by the [*WaitOrTimerCallback*](http://msdn.microsoft.com/en-us/library/system.threading.waitortimercallback(VS.80).aspx) delegate. In both cases, the thread pool uses a background thread to invoke the callback method.

Example

**Are all ThreadPool threads are in the multithreaded apartment?**

Yes

**[What’s the difference between Invoke() and BeginInvoke()](http://stackoverflow.com/questions/229554/whats-the-difference-between-invoke-and-begininvoke)**

Invoke and BeginInvoke methods exist in Delegate.Invoke/BeginInvoke or Control.Invoke/BeginInvoke.

* *Delegate.Invoke:* Executes synchronously, on the same thread.
* *Delegate.BeginInvoke:* Executes asynchronously, on a thread pool thread.
* *Control.Invoke:* Executes the specified delegate on the thread (generally UI thread) that owns the control's underlying window handle, but calling thread waits for completion before continuing.
* *Control.BeginInvoke:* Executes the specified delegate on the thread (generally UI thread) that owns the control's underlying window handle, and calling thread doesn't wait for completion.

**For Windows Forms apps, which method call Control.BeginInvoke or Control.Invoke should be used?**

For Windows Forms apps, you should usually use BeginInvoke. That way you don't need to worry about deadlock, for example - but you need to understand that the UI may not have been updated by the time you next look at it! In particular, you shouldn't modify data which the UI thread might be about to use for display purposes. For example, if you have a Person with FirstName and LastName properties, and you did:

person.FirstName = "Kevin"; // person is a shared reference

person.LastName = "Spacey";

control.BeginInvoke(UpdateName);

person.FirstName = "Keyser";

person.LastName = "Soze";

then the UI may well end up displaying "Keyser Spacey". (There's an outside chance it could display "Kevin Soze" but only through the weirdness of the memory model.)

Unless you have this sort of issue, however, Control.BeginInvoke is easier to get right, and will avoid your background thread from having to wait for no good reason. Note that the Windows Forms team has guaranteed that you can use Control.BeginInvoke in a "fire and forget" manner - i.e. without ever calling EndInvoke. This is not true of async calls in general: normally every BeginXXX should have a corresponding EndXXX call, usually in the callback.

***Using Threads***

**What is event-based asynchronous pattern?**

The event-based asynchronous pattern (EAP) provides a simple means by which classes can offer multithreading capability without consumers needing to explicitly start or manage threads.

<http://msdn.microsoft.com/en-us/library/wewwczdw.aspx>

**What are the advantages of EAP?**

It also provides the following features:

A cooperative cancellation model

The ability to safely update WPF or Windows Forms controls when the worker completes

Forwarding of exceptions to the completion event

The EAP is just a pattern, so these features must be written by the implementer. Just a few classes in the Framework follow this pattern, most notably BackgroundWorker (which we’ll cover next), and WebClient in System.Net.

**Give Examples of the Event-based Asynchronous Pattern from .NET framework.**

The [*SoundPlayer*](http://msdn.microsoft.com/en-us/library/system.media.soundplayer.aspx) and [*PictureBox*](http://msdn.microsoft.com/en-us/library/system.windows.forms.picturebox.aspx) components represent simple implementations of the Event-based Asynchronous Pattern. The [*WebClient*](http://msdn.microsoft.com/en-us/library/system.net.webclient.aspx) and [*BackgroundWorker*](http://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker.aspx) components represent more complex implementations of the Event-based Asynchronous Pattern.

**Give an example class declaration that conforms to the EAP pattern.**

Below is an example class declaration that conforms to the pattern:

public class AsyncExample

{

// Synchronous methods.

public int Method1(string param);

public void Method2(double param);

// Asynchronous methods.

public void Method1Async(string param);

public void Method1Async(string param, object userState);

public event Method1CompletedEventHandler Method1Completed;

public void Method2Async(double param);

public void Method2Async(double param, object userState);

public event Method2CompletedEventHandler Method2Completed;

public void CancelAsync(object userState);

public bool IsBusy { get; }

// Class implementation not shown.

}

The fictitious AsyncExample class has two methods, both of which support synchronous and asynchronous invocations. The synchronous overloads behave like any method call and execute the operation on the calling thread; if the operation is time-consuming, there may be a noticeable delay before the call returns. The asynchronous overloads will start the operation on another thread and then return immediately, allowing the calling thread to continue while the operation executes "in the background."

**What is a BackgroundWorker? What are it’s characteristics?**

*BackgroundWorker*is a helper class in the *System.ComponentModel*namespace for managing

a worker thread. It provides the following features:

A "cancel" flag for signaling a worker to end without using Abort

A standard protocol for reporting progress, completion and cancellation

An implementation of *IComponent*allowing it be sited in the Visual Studio Designer

Exception handling on the worker thread

The ability to update Windows Forms and WPF controls in response to worker progress or completion.

*The last two features are particularly useful – it means you don't have to include a* ***try/catch*** *block*

*in your worker method, and can update Windows Forms and WPF controls without needing to call*

***Control.Invoke****.*

**Does BackgroundWorker uses thread pool?**

*BackgroundWorker*uses the thread-pool, which recycles threads to avoid recreating them for

each new task. This means one should never call Abort on a *BackgroundWorker*thread.

**Explain the minimum steps in using BackgroundWorker.**

Here are the minimum steps in using *BackgroundWorker*:

Instantiate *BackgroundWorker*, and handle the *DoWork*event

Call *RunWorkerAsync*, optionally with an object argument.

This then sets it in motion. Any argument passed to *RunWorkerAsync*will be forwarded to

*DoWork's* event handler, via the event argument's *Argument*property. Here's an example:

class Program

{

static BackgroundWorker bw = new BackgroundWorker();

static void Main()

{

bw.***DoWork*** += bw\_DoWork;

bw.***RunWorkerAsync*** ("Message to worker");

Console.ReadLine();

}

static void bw\_DoWork (object sender, DoWorkEventArgs e)

{

// This is called on the worker thread

Console.WriteLine (e.Argument); // writes "Message to worker"

// Perform time-consuming task...

}

}

**What is the advantage/use of BackgroundWorker?**

The *BackgroundWorker* component gives you the ability to execute time-consuming operations asynchronously ("in the background"), on a thread different from your application's main UI thread. To use a *BackgroundWorker*, you simply tell it what time-consuming worker method to execute in the background, and then you call the [*RunWorkerAsync*](http://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker.runworkerasync(VS.80).aspx) method. Your calling thread continues to run normally while the worker method runs asynchronously. When the method is finished, the *BackgroundWorker* alerts the calling thread by firing the [*RunWorkerCompleted*](http://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker.runworkercompleted(VS.80).aspx) event, which optionally contains the results of the operation.

**Give examples where BackgroundWorker can be used in real life.**

There are many commonly performed operations that can take a long time to execute. For example:

* Image downloads
* Web service invocations
* File downloads and uploads (including for peer-to-peer applications)
* Complex local computations
* Database transactions
* Local disk access, given its slow speed relative to memory access

Operations like these can cause your user interface to hang while they are running. When you want a responsive UI and you are faced with long delays associated with such operations, the [BackgroundWorker](http://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker(VS.80).aspx) component provides a convenient solution.

**What are the different timer classes in the .NET Framework Class Library?**

The .NET Framework provides four timers. Two of these are general-purpose multithreaded timers:

System.Threading.Timer

System.Timers.Timer

The other two are special-purpose single-threaded timers:

System.Windows.Forms.Timer (Windows Forms timer)

System.Windows.Threading.DispatcherTimer (WPF timer)

The multithreaded timers are more powerful, accurate, and flexible; the single-threaded timers are safer and more convenient for running simple tasks that update Windows Forms controls or WPF elements.

**What are the common uses of timers?**

Some of the most common uses of timers are

* to start a process at a regularly scheduled time,
* to set intervals between events, and
* to maintain consistent animation speeds (regardless of processor speed) when working with graphics.

**Compare the different timers.**

| **System.Windows.Forms.Timer (Windows Forms)** | **System.Windows.Threading.DispatcherTimer (WPF)** | **System.Threading.Timer** | **System.Timers.Timer** |
| --- | --- | --- | --- |
| A Windows Forms timer does not use the thread pool, instead firing its "Tick" event always on the same thread that originally created the timer. The timer's event handler is then able to interact with the forms and controls without violating thread-safety – or the impositions of apartment-threading. | Instead of using the thread pool to generate timer events, the WPF and Windows Forms timers rely on the message pumping mechanism of their underlying user interface model. This means that the Tick event always fires on the same thread that originally created the timer—which, in a normal application, is the same thread used to manage all user interface elements and controls. | The threading timer takes advantage of the thread pool, allowing many timers to be created without the overhead of many threads. | The System.Timers.Timer class will, by default, calls timer event handler on a worker thread obtained from the common language runtime (CLR) thread pool.  It is a wrapper around Win32 waitable timer objects and raises an Elapsed event on a worker thread rather than a Tick event on the UI thread. The Elapsed event must be connected to an event handler that matches the ElapsedEventHandler delegate. The event handler receives an argument of type ElapsedEventArgs. |
| You can update user interface elements and controls directly from Tick event handling code, without calling **Control.Invoke.** | You can update user interface elements and controls directly from Tick event handling code, without calling **Dispatcher.Invoke.** | **Control.Invoke** is required. | This means that the code inside your Elapsed event handler must conform to a golden rule of Win32 programming: an instance of a control should never be accessed from any thread other than the thread that was used to instantiate it. But, it exposes a public SynchronizingObject property. Setting this property to an instance of a Windows Form (or a control on a Windows Form) will ensure that the code in your Elapsed event handler runs on the same thread on which the SynchronizingObject was instantiated. |
| The Windows timer is, in effect, a *singlethreaded timer*. Timer event runs on UI thread. | The WPF timer is, in effect, a *singlethreaded timer*. Timer event runs on UI thread. | Timer event runs on worker thread. | Timer event runs on UI or worker thread. The .NET Framework documentation refers to the System.Timers.Timer class as a server-based timer that was designed and optimized for use in multithreaded environments. |
| Instances are not thread safe. |  | Instances are not thread safe. | Instances of this timer class can be safely accessed from multiple threads. |
| Requires Windows Forms. Windows Forms and WPF timers are intended for jobs that may involve updating the user interface and which execute quickly. Quick execution is important because the **Tick** event is called on the main thread – which if tied up, will make the user interface unresponsive. |  | Does not require Windows Forms. | Does not require Windows Forms. |