[*Framework Fundamentals* 9](#_Toc437189441)

[**1.** **What is an IL?** 9](#_Toc437189442)

[**2.** **What is a CLR?** 9](#_Toc437189443)

[**3.** **What are CTS and CLS?** 9](#_Toc437189444)

[**4.** **What is a Managed Code?** 10](#_Toc437189445)

[**5.** **What is an Assembly?** 10](#_Toc437189446)

[**6.** **What are different types of Assembly?** 10](#_Toc437189447)

[**7.** **What is Namespace?** 10](#_Toc437189448)

[**8.** **What is Difference between Namespace and Assembly?** 10](#_Toc437189449)

[**9.** **What is Manifest?** 11](#_Toc437189450)

[**10.** **What is GAC?** 11](#_Toc437189451)

[**11.** **How to add and remove an assembly from GAC?** 11](#_Toc437189452)

[**12.** **What’s difference between System exceptions and Application exceptions?** 11](#_Toc437189453)

[**13.** **What is Code Access security?** 11](#_Toc437189454)

[**14.** **What is Weak Reference?** 12](#_Toc437189455)

[**15.** **How to use Weak References?** 12](#_Toc437189456)

[**16.** **What is a satellite assembly?** 12](#_Toc437189457)

[**17.** **What is reflection?** 13](#_Toc437189458)

[**18.** **What is concept of Boxing and Unboxing?** 13](#_Toc437189459)

[**19.** **What’s the difference between Convert.toString and .toString ( ) method?** 13](#_Toc437189460)

[**20.** **What is Native Image Generator (Ngen.exe)?** 13](#_Toc437189461)

[**21.** **How to use different version of same assembly in GAC?** 13](#_Toc437189462)

[**22.** **What is an AppDomain?** 14](#_Toc437189463)

[**23.** **What is Delay signing?** 15](#_Toc437189464)

[*C#* 16](#_Toc437189465)

[**1.** **What is IEnumerable, ICollection, IList and IDictionary?** 16](#_Toc437189466)

[**2.** **What is IComparable, IComparer and IEquatable?** 16](#_Toc437189467)

[**3.** **What are the various collections available in C#?** 17](#_Toc437189468)

[**4.** **What is the using keyword? What happens in case of crash inside the using block?** 18](#_Toc437189469)

[**5.** **What is upcast and downcast? What does "as" operator do?** 19](#_Toc437189470)

[**6.** **What are Generics?** 20](#_Toc437189471)

[**7.** **What are Generics Constraints?** 20](#_Toc437189472)

[**8.** **What is a delegate? What are Generic Delegates?** 21](#_Toc437189473)

[**9.** **What are Multi-Cast delegates?** 22](#_Toc437189474)

[**10.** **What are events? Do Events Have return type?** 22](#_Toc437189475)

[**11.** **Can events have access modifiers?** 23](#_Toc437189476)

[**12.** **What is an Extension method?** 23](#_Toc437189477)

[**13.** **What is an Anonymous method?** 24](#_Toc437189478)

[**14.** **What is an Automatic Property?** 24](#_Toc437189479)

[**15.** **What is Type inference and Anonymous Type?** 25](#_Toc437189480)

[**16.** **What is Dynamic keyword? How it is different from var and Object?** 25](#_Toc437189481)

[**17.** **What are Attributes?** 26](#_Toc437189482)

[**18.** **What is Yield statement?** 26](#_Toc437189483)

[OOPS 27](#_Toc437189484)

[**1.** **What are different properties provided by Object-oriented systems?** 27](#_Toc437189485)

[**2.** **What are association, aggregation and Composition?** 27](#_Toc437189486)

[**3.** **What is Method Overriding and Method Hiding (shadowing)?** 28](#_Toc437189487)

[**4.** **What are Static(C#) variables?** 29](#_Toc437189488)

[**5.** **What is the different accessibility levels defined in .NET?** 29](#_Toc437189489)

[**6.** **Some points regarding access modifiers.** 30](#_Toc437189490)

[**7.** **Can you prevent a class from overriding?** 31](#_Toc437189491)

[**8.** **What is immutable class?** 31](#_Toc437189492)

[**9.** **What is the difference between System.String and System.StringBuilder classes?** 31](#_Toc437189493)

[**10.** **What is Indexer? Can we have static indexer in C#?** 31](#_Toc437189494)

[**11.** **What are some commonly used UML symbols?** 32](#_Toc437189495)

[*Patterns* 33](#_Toc437189496)

[**1.** **What are design patterns? (A) Can you list down all patterns and there classification?** 33](#_Toc437189497)

[**2.** **What is Strategy Pattern?** 33](#_Toc437189498)

[**3.** **Explain Observer pattern** 35](#_Toc437189499)

[**4.** **What is Factory Pattern?** 35](#_Toc437189500)

[**5.** **What is Abstract Factory Pattern?** 38](#_Toc437189501)

[**6.** **How can we implement singleton pattern in .NET?** 39](#_Toc437189502)

[**7.** **Explain Decorator pattern.** 40](#_Toc437189503)

[**8.** **What is Command Pattern?** 42](#_Toc437189504)

[**9.** **How do you implement prototype pattern in .NET?** 42](#_Toc437189505)

[**10.** **Explain Adapter Pattern?** 43](#_Toc437189506)

[**11.** **Explain Façade pattern?** 44](#_Toc437189507)

[**12.** **Explain Template Method** 45](#_Toc437189508)

[**13.** **What is SOLID?** 46](#_Toc437189509)

[**14.** **What is Single responsibility principle?** 46](#_Toc437189510)

[**15.** **What is Liskov substitution principle?** 46](#_Toc437189511)

[**16.** **What is Interface segregation principle?** 46](#_Toc437189512)

[**17.** **What is Dependency Inversion principle?** 46](#_Toc437189513)

[**18.** **What is Inversion of Control (IOC)?** 47](#_Toc437189514)

[**19.** **What is Dependency Injection?** 47](#_Toc437189515)

[**20.** **What is Unit of Work?** 47](#_Toc437189516)

[**21.** **What is Managed Add-in Framework (MAF)?** 47](#_Toc437189517)

[**22.** **What is the Add-in Model?** 48](#_Toc437189518)

[**23.** **What is MEF?** 48](#_Toc437189519)

[**24.** **Explain MVVM.** 49](#_Toc437189520)

[*WCF* 50](#_Toc437189521)

[**1.** **What is WCF?** 50](#_Toc437189522)

[**2.** **What are the important principles of SOA (Service oriented Architecture)?** 50](#_Toc437189523)

[**3.** **What are end points, contract, address and bindings?** 50](#_Toc437189524)

[**4.** **What are bindings?** 50](#_Toc437189525)

[**5.** **What are the various ways of hosting a WCF service?** 51](#_Toc437189526)

[**6.** **What are different bindings supported by WCF?** 51](#_Toc437189527)

[**7.** **What are the main components of WCF?** 52](#_Toc437189528)

[**8.** **What is a service class?** 53](#_Toc437189529)

[**9.** **What are service contract, operation contract and Data Contract?** 53](#_Toc437189530)

[**10.** **What is the difference WCF and Web services?** 53](#_Toc437189531)

[**11.** **What are the message exchange pattern in WCF?** 53](#_Toc437189532)

[**12.** **What is Request-Reply pattern in WCF?** 53](#_Toc437189533)

[**13.** **What is One-way pattern in WCF?** 54](#_Toc437189534)

[**14.** **Are One-way calls same as Asynchronous calls?** 54](#_Toc437189535)

[**15.** **Can you explain duplex message exchange pattern in WCF?** 54](#_Toc437189536)

[**16.** **What are Async calls in WCF?** 54](#_Toc437189537)

[**17.** **Can you explain transactions in WCF?** 55](#_Toc437189538)

[**18.** **What different transaction isolation levels provided in WCF?** 56](#_Toc437189539)

[**19.** **What are channels in WCF?** 57](#_Toc437189540)

[**20.** **How a client request is processed in WCF?** 57](#_Toc437189541)

[**21.** **What is KnownType attribute?** 58](#_Toc437189542)

[**22.** **How to make WCF accept a single value, Array and List in same Data-Member?** 59](#_Toc437189543)

[**23.** **How does KnownType works for Generic type data-contracts?** 59](#_Toc437189544)

[**24.** **How are sessions maintained in WCF?** 59](#_Toc437189545)

[**25.** **How are sessions initiated and Terminated in WCF?** 60](#_Toc437189546)

[**26.** **What is Instancing and what are instantiation modes?** 61](#_Toc437189547)

[**27.** **What is PerCall Instantiation mode?** 61](#_Toc437189548)

[**28.** **What is PerSession Instantiation mode?** 61](#_Toc437189549)

[**29.** **What is Singleton Instantiation mode?** 62](#_Toc437189550)

[**30.** **What are concurrency modes?** 62](#_Toc437189551)

[**31.** **What is Single Concurrency mode?** 62](#_Toc437189552)

[**32.** **What is Multiple Concurrency mode?** 63](#_Toc437189553)

[**33.** **What is Re-Entrant Concurrency mode?** 63](#_Toc437189554)

[**34.** **What is Service Throttling?** 64](#_Toc437189555)

[**35.** **What happens when there is a change in DataContract?** 64](#_Toc437189556)

[**36.** **What is ExtensionDataObject?** 64](#_Toc437189557)

[*Security* 65](#_Toc437189558)

[**1.** **How CAS helps in ASP.Net Security?** 65](#_Toc437189559)

[**2.** **How are configuration files structured in ASP.net?** 65](#_Toc437189560)

[**3.** **What is ASP.Net worker process identity?** 66](#_Toc437189561)

[**4.** **What are IIS Authentication Modes?** 66](#_Toc437189562)

[**5.** **What is Anonymous Authentication?** 67](#_Toc437189563)

[**6.** **What is Basic Authentication?** 67](#_Toc437189564)

[**7.** **What is Digest Authentication?** 68](#_Toc437189565)

[**8.** **What is Client Certificate Mapping Authentication?** 68](#_Toc437189566)

[**9.** **What is Forms Authentication?** 69](#_Toc437189567)

[**10.** **What is ASP.NET Impersonation Authentication?** 69](#_Toc437189568)

[**11.** **What is File Authorization?** 69](#_Toc437189569)

[**12.** **What is the Scope of WCF security?** 70](#_Toc437189570)

[**13.** **What is Transfer Security?** 70](#_Toc437189571)

[**14.** **What is Transport Security Mode?** 70](#_Toc437189572)

[**15.** **What is Message Security Mode?** 71](#_Toc437189573)

[**16.** **What are the authentication options available with Transport security?** 71](#_Toc437189574)

[**17.** **What are the authentication options available with Message security?** 72](#_Toc437189575)

[**18.** **What are the Authorization Options in WCF?** 72](#_Toc437189576)

[**19.** **What are the Identities in WCF?** 72](#_Toc437189577)

[**20.** **What is Claim-based authorization?** 73](#_Toc437189578)

[**21.** **What is Impersonation?** 73](#_Toc437189579)

[**22.** **What is Impersonation and delegation in WCF?** 74](#_Toc437189580)

[**23.** **What is Access Control?** 74](#_Toc437189581)

[**24.** **How will you secure a Web Service?** 75](#_Toc437189582)

[**25.** **What is IIS Access Control?** 75](#_Toc437189583)

[**26.** **What is WS-Security Specifications?** 75](#_Toc437189584)

[*Memory Management* 77](#_Toc437189585)

[**1.** **Some Memory Fundamentals** 77](#_Toc437189586)

[**2.** **What are the conditions for garbage collection** 77](#_Toc437189587)

[**3.** **What is managed heap?** 77](#_Toc437189588)

[**4.** **What are Generations?** 78](#_Toc437189589)

[**5.** **What happens during a Garbage Collection?** 78](#_Toc437189590)

[**6.** **What is garbage collection?** 79](#_Toc437189591)

[**7.** **What’s the significance of Finalize method in .NET?** 79](#_Toc437189592)

[**8.** **What is a Finalizer Queue?** 80](#_Toc437189593)

[**9.** **How can we force garbage collection and how can we suppress a finalize method?** 80](#_Toc437189594)

[**10.** **What is IDisposable, Dispose and Close method?** 80](#_Toc437189595)

[**11.** **How to force the Dispose method to be called automatically during finalization?** 80](#_Toc437189596)

[**12.** **What are Value types and Reference types?** 81](#_Toc437189597)

[*Threading* 82](#_Toc437189598)

[**1.** **Why is Threading used?** 82](#_Toc437189599)

[**2.** **What is Join and Sleep?** 82](#_Toc437189600)

[**3.** **What is Foreground and Background thread?** 82](#_Toc437189601)

[**4.** **What is a thread pool?** 83](#_Toc437189602)

[**5.** **How a thread pool is accessed?** 83](#_Toc437189603)

[**6.** **What is Background Worker?** 83](#_Toc437189604)

[**7.** **How a BackgroundWorker is used and how to cancel the BackgroundWorker thread?** 84](#_Toc437189605)

[**8.** **What is Asynchronous Delegates?** 85](#_Toc437189606)

[**9.** **What is QueueUserWorkItem and how it is different than Async Delegates?** 86](#_Toc437189607)

[**10.** **What is Task?** 86](#_Toc437189608)

[**11.** **What is Synchronization and how it is implemented?** 87](#_Toc437189609)

[**12.** **What is Thread Blocking?** 87](#_Toc437189610)

[**13.** **What is Locking?** 88](#_Toc437189611)

[**14.** **How Lock is used and how it is different from Monitors?** 88](#_Toc437189612)

[**15.** **What is Mutex and how it is used?** 88](#_Toc437189613)

[**16.** **What is Thread Safety?** 89](#_Toc437189614)

[**17.** **What is Lazy Initialization?** 89](#_Toc437189615)

[**18.** **What are Asynchronous methods?** 90](#_Toc437189616)

[**19.** **What is Thread Affinity?** 91](#_Toc437189617)

[**20.** **What is Thread Local Storage?** 91](#_Toc437189618)

[**21.** **What is Signaling method of Synchronization?** 91](#_Toc437189619)

[**22.** **What is AutoResetEvent and ManualResetEvent?** 91](#_Toc437189620)

[**23.** **What is CountDownEvent?** 92](#_Toc437189621)

[*WPF* 94](#_Toc437189622)

[**1.** **What is Dispatcher?** 94](#_Toc437189623)

[**2.** **What is DispatcherObject?** 94](#_Toc437189624)

[**3.** **What is Dependency property?** 94](#_Toc437189625)

[**4.** **How does Dependency property works?** 94](#_Toc437189626)

[**5.** **What are the advantages of Dependency property?** 95](#_Toc437189627)

[**6.** **What are attached properties?** 96](#_Toc437189628)

[**7.** **What are the type of controls available in WPF?** 96](#_Toc437189629)

[**8.** **What are Binary Resources?** 96](#_Toc437189630)

[**9.** **What are Logical Resources?** 96](#_Toc437189631)

[**10.** **What are the differences between Static and Dynamic resources?** 97](#_Toc437189632)

[**11.** **What are Freezable object?** 98](#_Toc437189633)

[**12.** **What are resource dictionary?** 98](#_Toc437189634)

[**13.** **What is Visual Tree and Logical tree?** 98](#_Toc437189635)

[**14.** **Are Content Elements part of VisualTree?** 99](#_Toc437189636)

[**15.** **What are Routed Events? Explain Bubbling and Tunneling.** 100](#_Toc437189637)

[**16.** **What is EventManager class?** 101](#_Toc437189638)

[**17.** **What are Styles? Explain its few properties.** 101](#_Toc437189639)

[**18.** **How a Style is created and applied?** 102](#_Toc437189640)

[**19.** **What are Triggers?** 103](#_Toc437189641)

[**20.** **What are Property Trigger and MultiTrigger?** 103](#_Toc437189642)

[**21.** **What are Data Trigger and Multi-Data Trigger?** 104](#_Toc437189643)

[**22.** **What are Data Templates?** 105](#_Toc437189644)

[**23.** **What is an Event Trigger?** 106](#_Toc437189645)

[**24.** **What are Control Template?** 106](#_Toc437189646)

[**25.** **What is TemplateBinding?** 107](#_Toc437189647)

[**26.** **What are the difference between Style and Control Template?** 107](#_Toc437189648)

[**27.** **What is Data Binding?** 107](#_Toc437189649)

[**28.** **Explain the Binding class** 107](#_Toc437189650)

[**29.** **What is DataContext?** 108](#_Toc437189651)

[**30.** **What is Converter?** 108](#_Toc437189652)

[**31.** **What is Multi-Converter?** 109](#_Toc437189653)

[**32.** **What is difference between DataTrigger and Converter?** 110](#_Toc437189654)

[**33.** **What is Command?** 110](#_Toc437189655)

[**34.** **What is WPF Command Model?** 111](#_Toc437189656)

[**35.** **What is ICommandSource?** 111](#_Toc437189657)

[**36.** **What is UI Element and Framework Element?** 112](#_Toc437189658)

[**37.** **What is Property Element and Object Element?** 112](#_Toc437189659)

[**38.** **What is DelegateCommand?** 112](#_Toc437189660)

[**39.** **What is Command Object?** 112](#_Toc437189661)

[*MVC* 113](#_Toc437189662)

[**1.** **What is Worker Process?** 113](#_Toc437189663)

[**2.** **What is Application Pool?** 113](#_Toc437189664)

[**3.** **What is WWW Service?** 113](#_Toc437189665)

[**4.** **What is Windows Process Activation Service (WAS)?** 114](#_Toc437189666)

[**5.** **What is Http.Sys?** 115](#_Toc437189667)

[**6.** **What are HTTP Modules?** 115](#_Toc437189668)

[**7.** **What are HTTP Handlers?** 115](#_Toc437189669)

[**8.** **How IIS processes an HTTP request?** 115](#_Toc437189670)

[**9.** **How is the ASP.Net routing different from MVC routing?** 117](#_Toc437189671)

[**10.** **What is URL Routing Module?** 117](#_Toc437189672)

[**11.** **What is MVC page life cycle?** 117](#_Toc437189673)

[**12.** **What are Routes? What is Route Table?** 118](#_Toc437189674)

[**13.** **How are Routes Configured?** 118](#_Toc437189675)

[**14.** **What is Caching and what are its types?** 119](#_Toc437189676)

[**15.** **What is Server side caching and Client side caching?** 119](#_Toc437189677)

[**16.** **What is Application-scoped caching?** 120](#_Toc437189678)

[**17.** **What is ASP.Net cache (application cache)?** 120](#_Toc437189679)

[**18.** **How is cached items Expired** 120](#_Toc437189680)

[**19.** **What are Cache dependencies?** 121](#_Toc437189681)

[**20.** **What is Cache Scavenging?** 121](#_Toc437189682)

[**21.** **What is Output Cache?** 121](#_Toc437189683)

[**22.** **How do you configure the Cache location?** 122](#_Toc437189684)

[**23.** **How can you vary the output cache?** 122](#_Toc437189685)

[**24.** **What are DataAnnotations?** 123](#_Toc437189686)

[**25.** **What is CORS?** 123](#_Toc437189687)

[**26.** **What is REST and RESTful WebApi?** 123](#_Toc437189688)

[**27.** **What is jQuery?** 124](#_Toc437189689)

[**28.** **What is NuGet?** 125](#_Toc437189690)

[*ADO.Net* 126](#_Toc437189691)

[**1.** **What is the namespace in which .NET have the data functionality classes?** 126](#_Toc437189692)

[**2.** **Can you give an overview of ADO.NET architecture?** 126](#_Toc437189693)

[**3.** **What is difference between dataset and DataReader?** 126](#_Toc437189694)

[**4.** **What is the use of command objects and what are the methods provided by the command object?** 126](#_Toc437189695)

[**5.** **What are basic methods of DataAdapter?** 127](#_Toc437189696)

[**6.** **What is Dataset object?** 127](#_Toc437189697)

[**7.** **What are the various objects in Dataset?** 127](#_Toc437189698)

[**8.** **How can we force the connection object to close after my DataReader is closed?** 127](#_Toc437189699)

[**9.** **How can we fine tune the command object when we are expecting a single row or a single value?** 128](#_Toc437189700)

[**10.** **Which is the best place to store ConnectionString in .NET projects?** 128](#_Toc437189701)

[**11.** **What are the various methods provided by the dataset object to generate XML?** 128](#_Toc437189702)

[**12.** **How can we save all data from dataset?** 128](#_Toc437189703)

[**13.** **How can we check that some changes have been made to dataset since it was loaded?** 128](#_Toc437189704)

[**14.** **How can we add/remove row’s in “DataTable” object of “DataSet”?** 128](#_Toc437189705)

[**15.** **What’s basic use of “DataView”?** 129](#_Toc437189706)

[**16.** **How can we add relation’s between tables in a DataSet?** 129](#_Toc437189707)

[**17.** **What’s the use of CommandBuilder?** 129](#_Toc437189708)

[**18.** **What’s difference between “Optimistic” and “Pessimistic” locking?** 129](#_Toc437189709)

[**19.** **How many ways are there to implement locking in ADO.NET?** 130](#_Toc437189710)

[**20.** **How can we perform transactions in .NET?** 130](#_Toc437189711)

[**21.** **What are the Isolation Levels for a transaction?** 130](#_Toc437189712)

# 

# *Framework Fundamentals*

1. **What is an IL?**

* (IL)Intermediate Language is also known as MSIL (Microsoft Intermediate Language) or CIL (Common Intermediate Language).
* It 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.

1. **What is a CLR?**

CLR (Common Language Runtime) forms the heart of the .NET framework. All Languages have runtime and it’s the responsibility of the runtime to take care of the code execution of the program. For example VC++ has MSCRT40.DLL, VB6 has MSVBVM60.DLL, and Java has Java Virtual Machine etc. Similarly .NET has **CLR (MSCorEE.dll)**

Following are the responsibilities of CLR

* **Garbage Collection**: - CLR automatically manages memory thus eliminating memory leaks. When objects are not referred GC automatically releases those memories thus providing efficient memory management.
* **Code Access Security**: - CAS grants rights to program depending on the security configuration of the machine. Example the program has rights to edit or create a new file but the security configuration of machine does not allow the program to delete a file. CAS will take care that the code runs under the environment of machines security configuration.
* **Code Verification**: - This ensures proper code execution and type safety while the code runs. It prevents the source code to perform illegal operation such as accessing invalid memory locations etc.
* **IL (Intermediate language)**-to-native translators and optimizer’s:- CLR uses JIT and compiles the IL code to machine code and then executes. CLR also determines depending on platform what is optimized way of running the IL code.

1. **What are CTS and CLS?**

In order that two languages communicate smoothly CLR has CTS (Common Type System). Example in VB you have “Integer” and in C++ you have “long”. These data types are not compatible so the interfacing between them is very complicated.

In order that two different languages can communicate, Microsoft introduced Common Type System. So “Integer” data type in VB6 and “int” data type in C++ will convert it to System.int32 which is data type of CTS. CLS which is covered in the coming question is subset of CTS.

CLS (Common Language Specifications) is a subset of the CTS which all .NET languages are expected to support. Microsoft has defined CLS which are nothing but guidelines that language to follow so that it can communicate with other .NET languages in a seamless manner.

1. **What is a Managed Code?**

Managed code runs inside the environment of CLR i.e. .NET runtime. In short all IL are managed code. But if you are using some third party software example VB6 or VC++ component they are unmanaged code as .NET runtime (CLR) does not have control over the source code execution of the language.

1. **What is an Assembly?**

* Assembly is unit of deployment like EXE or a DLL.
* An assembly consists of one or more files (dlls, exe’s, html files etc.), and represents a group of resources, type definitions, and implementations of those types. An assembly may also contain references to other assemblies. These resources, types and references are described in a block of data called a manifest. The manifest is part of the assembly, thus making the assembly self-describing.
* An assembly is completely self-describing. An assembly contains metadata information, which is used by the CLR for everything from type checking and security to actually invoking the components methods. As all information is in assembly itself it is independent of registry. This is the basic advantage as compared to COM where the version was stored in registry.
* Multiple versions can be deployed side by side in different folders. These different versions can execute at the same time without interfering with each other. Assemblies can be private or shared. For private assembly deployment, the assembly is copied to the same directory as the client program that references it. No registration is needed, and no fancy installation program is required. When the component is removed, no registry cleanup is needed, and no uninstall program is required. Just delete it from the hard drive.
* In shared assembly deployment, an assembly is installed in the Global Assembly Cache (or GAC). The GAC contains shared assemblies that are globally accessible to all .NET applications on the machine.

1. **What are different types of Assembly?**

There are two types of assembly Private and Public assembly. A private assembly is normally used by a single application, and is stored in the application's directory, or a sub-directory beneath. A shared assembly is normally stored in the global assembly cache, which is a repository of assemblies maintained by the .NET runtime. Shared assemblies are usually libraries of code which many applications will find useful, e.g. Crystal report classes which will be used by all application for Reports.

1. **What is Namespace?**

Namespace has two basic functionalities:-

* Namespace Logically group types. Example System.Web.UI logically groups our UI related features.
* In Object Oriented world many times it is possible that programmers will use the same class name. By qualifying Namespace with class name this collision can be removed.

1. **What is Difference between Namespace and Assembly?**

Following are the differences between namespace and assembly:

* Assembly is physical grouping of logical units. Namespace logically groups classes.
* Namespace can span multiple assemblies.

1. **What is Manifest?**

Assembly metadata is stored in Manifest. Manifest contains all the metadata needed to do the following things (See Figure Manifest View for more details):

* Version of assembly
* Security identity
* Scope of the assembly
* Resolve references to resources and classes.

The assembly manifest can be stored in either a PE file (an .exe or .dll) with Microsoft intermediate language (MSIL) code or in a stand-alone PE file that contains only assembly manifest information.

1. **What is GAC?**

*Twist: - What are situations when you register .NET assembly in GAC?*

GAC (Global Assembly Cache) is used where shared .NET assembly reside. GAC is used in the following situations:-

* If the application has to be shared among several application.
* If the assembly has some special security requirements like only administrators can remove the assembly. If the assembly is private then a simple delete of assembly the assembly file will remove the assembly.

*Note: - Registering .NET assembly in GAC can lead to the old problem of DLL hell, Where COM version was stored in central registry. So GAC should be used when absolutely necessary.*

1. **How to add and remove an assembly from GAC?**

There are two ways to install .NET assembly in GAC:-

* Using Microsoft Installer Package. You can get download of installer from http://www.microsoft.com.
* Using Gacutil. Goto “Visual Studio Command Prompt” and type “gacutil –I (assembly\_name)”. Where (assembly\_name) is the DLL name of the project.

1. **What’s 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. Application exceptions are used when we want to define user defined exception. System exceptions are the one which are defined by .NET.

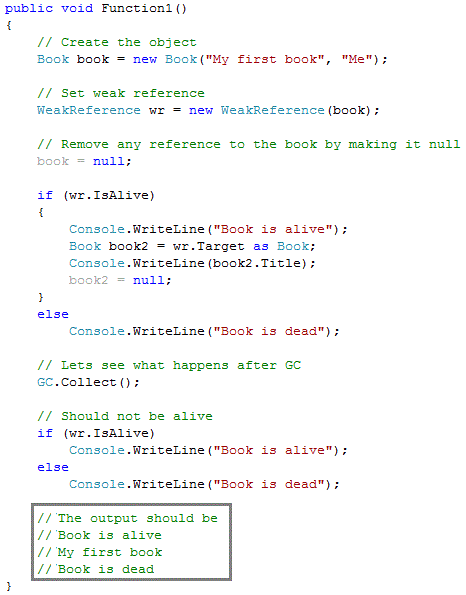
1. **What is Code Access security?**

CAS is part of .NET security model that determines whether or not a piece of code is allowed to run and what resources it can use while running. Example CAS will allow an application to read but now write and delete rights are given to the application.

1. **What is Weak Reference?**

The garbage collector cannot collect an object in use by an application while the application's code can reach that object. The application is said to have a strong reference to the object.

A weak reference permits the garbage collector to collect the object while still allowing the application to access the object. A weak reference is valid only during the indeterminate amount of time until the object is collected when no strong references exist.



When you use a weak reference, the application can still obtain a strong reference to the object, which prevents it from being collected. However, there is always the risk that the garbage collector will get to the object first before a strong reference is reestablished.

Weak references are useful for objects that use a lot of memory, but can be recreated easily if they are reclaimed by garbage collection.

Garbage collectors start cleaning memory for objects that do not have any reference. A weak reference is a way to have some pointer to an object that does have any reference (strong reference). When we need to access a weak referenced object we can just check if the object is alive and then access it if the object is alive at all. Since .NET is a garbage collection based runtime environment, like all GC based runtimes it does not immediately clean up the memory allocated for the instantiated objects.

1. **How to use Weak References?**

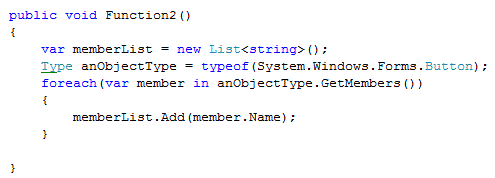
All you need to do is create a Weak-Reference class with the object in question passed into the constructor. Keep a strong reference to the weak reference object. When need later then check if object is alive by checking 'IsAlive' property and then use it. The code sample shows the lifetime of an object when using a weak reference.

1. **What is a satellite assembly?**

In multilingual application in .NET to support multilingual functionality you can have modules which are customized for localization. These assemblies are called as satellite assemblies. You can distribute these assemblies separately than the core modules.

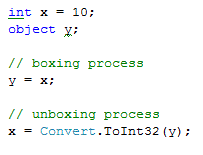
1. **What is reflection?**

All .NET assemblies have metadata information stored about the types defined in modules. This metadata information can be accessed by mechanism called as “Reflection”. System.Reflection can be used to browse through the metadata information. Using reflection you can also dynamically invoke methods using System.Type.Invokemember.



In the example, the sample source code uses reflection to browse through “Button” class of “Windows.Forms”. If you compile and run the program following is output as shown in “Sample Reflection Display”. Using reflection you can also dynamically invoke a method using “System.Type.InvokeMember”.

1. **What is concept of Boxing and Unboxing?**



Boxing permits any value type to be implicitly converted to type object or to any interface type implemented by value type. Boxing is process in which an object instances created and copying value types value in to that instance. Unboxing is vice versa of boxing operation where the value is copied from the instance in to appropriate storage location.

Given here is the sample code of boxing and un-boxing where integer data type is converted in to object and then vice versa.

1. **What’s the difference between Convert.toString and .toString ( ) method?**

Just to give an understanding of what the above question means seethe below code.

int i =0;

MessageBox.Show (i.ToString ());

MessageBox.Show (Convert.ToString (i));

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.

1. **What is Native Image Generator (Ngen.exe)?**

The Native Image Generator utility (Ngen.exe) allows you to run the JIT compiler on your assembly's MSIL and generate native machine code which is cached to disk. After the image is created .NET runtime will use the image to run the code rather than from the hard disk. Running Ngen.exe on an assembly potentially allows the assembly to load and execute faster, because it restores code and data structures from the native image cache rather than generating them dynamically.

1. **How to use different version of same assembly in GAC?**

By default it always refers the latest one. However in case the older one is required to be referred then we need to specify **bindingRedirect** in the config file.

For instance in the below case “**ClassLibraryVersion**” has two versions “1.1.1830.10493” and “1.0.1830.10461” from which “1.1.1830.10493” is the recent version. But using the **bindingRedirect** we can specify saying “1.0.1830.10461” is the new version. So the client will not use “1.1.1830.10493”.

<configuration>

<runtime>

<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">

<dependentAssembly>

<assemblyIdentity name="ClassLibraryVersion"

publicKeyToken = "b035c4774706cc72" culture="neutral"/>

<bindingRedirect oldVersion = “1.1.1830.10493"

newVersion = "1.0.1830.10461"/>

</dependentAssembly>

</assemblyBinding>

</runtime>

</configuration>

1. **What is an AppDomain?**

In windows, every program, like Notepad, executes inside a separate container known as process. If there are 4 Notepads open, then there will be 4 processes dedicated to each of the notepad instance.

A process contains the executable code and data of a program inside memory it has reserved from the operating system. There will be at least one thread executing instructions inside of the process, and in most cases there are multiple threads. If the program opens any files or other resources, those resources will belong to the process.

A process is also boundary. Erroneous code inside of a process cannot corrupt areas outside of the current process. It is easy to communicate inside of a process, but special techniques are required to communicate from one process to another. Each process also runs under a specific security context which can dictate what the process can do on the machine and network.

A process is the smallest unit of isolation available on the Windows operating system. This could pose a problem for an ISP who wants to host hundreds of ASP.NET applications on a single server. The ISP will want to isolate each ASP.NET application to prevent one application from interfering with another company’s application on the same server, but the relative cost of launching and executing a process for hundreds of applications may be prohibitive.

.NET introduces the concept of an **application domain**, or **AppDomain**. Like a process, the **AppDomain** is both a container and a boundary. The .NET runtime uses an **AppDomain** as a container for code and data, just like the operating system uses a process as a container for code and data. As the operating system uses a process to isolate misbehaving code, the .NET runtime uses an AppDomain to isolate code inside of a secure boundary.

An **AppDomain** belongs to only a single process, but single process can hold multiple **AppDomains**. An **AppDomain** is relatively cheap to create (compared to a process), and has relatively less overhead to maintain than a process. For these reasons, an **AppDomain** is a great solution for the ISP who is hosting hundreds of applications. Each application can exist inside an isolated AppDomain, and many of these AppDomains can exist inside of a single process – a cost savings.

An object lives in one **AppDomain**. Each ASP.NET application will have its own set of global variables: Cache, Application, and Session objects are not shared. Even though the code for both of the applications resides inside the same process, the unit of isolation is the .NET **AppDomain**. If there are classes with shared or static members, and those classes exist in both applications, each AppDomain will have its own copy of the static fields – the data is not shared. The code and data for each application is safely isolated and inside of a boundary provided by the **AppDomain**

1. **What is Delay signing?**

An organization can have a closely guarded key pair that developers do not have access to on a daily basis. The public key is often available, but access to the private key is restricted to only a few individuals. When developing assemblies with strong names, each assembly that references the strong-named target assembly contains the token of the public key used to give the target assembly a strong name. This requires that the public key be available during the development process.

You can use delayed or partial signing at build time to reserve space in the portable executable (PE) file for the strong name signature, but defer the actual signing until some later stage (typically just before shipping the assembly).

The following steps outline the process to delay sign an assembly:

1. Obtain the public key portion of the key pair from the organization that will do the eventual signing. Typically this key is in the form of an .snk file, which can be created using the [Strong Name tool (Sn.exe)](http://msdn.microsoft.com/en-us/library/k5b5tt23(VS.80).aspx) provided by the .NET Framework SDK.
2. Annotate the source code for the assembly with two custom attributes from [System.Reflection](http://msdn.microsoft.com/en-us/library/system.reflection(VS.80).aspx):
   * **AssemblyKeyFileAttribute**, which passes the name of the file containing the public key as a parameter to its constructor.
   * [**AssemblyDelaySignAttribute**](http://msdn.microsoft.com/en-us/library/system.reflection.assemblydelaysignattribute(VS.80).aspx), which indicates that delay signing is being used by passing true as a parameter to its constructor. For example:

C#

[assembly:AssemblyKeyFileAttribute("myKey.snk")]

[assembly:AssemblyDelaySignAttribute(true)]

1. The compiler inserts the public key into the assembly manifest and reserves space in the PE file for the full strong name signature. The real public key must be stored while the assembly is built so that other assemblies that reference this assembly can obtain the key to store in their own assembly reference.
2. Because the assembly does not have a valid strong name signature, the verification of that signature must be turned off. You can do this by using the –Vr option with the Strong Name tool. The following example turns off verification for an assembly called myAssembly.dll.

sn –Vr myAssembly.dll

1. Later, usually just before shipping, you submit the assembly to your organization's signing authority for the actual strong name signing using the –R option with the Strong Name tool.

The following example signs an assembly called myAssembly.dll with a strong name using the sgKey.snk key pair.

sn -R myAssembly.dll sgKey.snk

# 

# *C#*

1. **What is IEnumerable, ICollection, IList and IDictionary?**

**IEnumerable** is the base interface for all non-generic collections that can be enumerated. It exposes just one method, **GetEnumerator**, which returns an **IEnumerator**. **IEnumerator** provides the ability to iterate through the collection by exposing a **Current** property and **MoveNext** and **Reset** methods. The **foreach** statement doesn’t really need this interface implemented in the collection class. It’s enough to have a method with the name **GetEnumerator** () that returns an object implementing the IEnumerator interface.

**ICollection** is the base interface for classes in the System.Collections namespace. The **ICollection** interface extends **IEnumerable**. **IList** and **IDictionary** are more specialized interfaces that extend **ICollection**. This interface is mainly used to get the number of elements in a collection using **Count** property and for synchronization. It also exposes **CopyTo** method to copy one array to another.

**IList** interface extends from **ICollection** and defines additional properties and methods. The major reason why the Array class implements the **IList** interface is that the ***IList*** interface defines the Item property for accessing the elements using an **indexer**. Apart from **Count** and **CopyTo** members from **ICollection**, **IList** exposes methods to **add**, **insert**, **remove** and **clear** items. The interface also exposes methods so that item can be individually accessed by Index.

**IDictionary** is implemented by non - generic collections whose elements have a key and a value. Each element is a **key*/*value** pair stored in a **DictionaryEntry** object. Each pair must have a unique key. Implementations can vary in whether they allow the key to be null. The value can be null and does not have to be unique. The **IDictionary** interface allows the contained keys and values to be enumerated, but it does not imply any particular sort order.

All the above interfaces are **non-generic** type. The class which implements any of these interface will be capable of handling heterogeneous object type. For example a class that implements IList will be able to Add, Remove, and Insert objects of reference types as well as value types in a single collection. **Generic** type of these interfaces are type safe.

1. **What is IComparable, IComparer and IEquatable?**

**IComparable** defines a generalized type-specific comparison method that a value type or class implements to order or sort its instances. This interface is implemented by types whose values can be ordered or sorted. It requires that implementing types define a single method, **CompareTo (Object)** that indicates whether the position of the current instance in the sort order is before, after, or the same as a second object of the same type.

**IComparer** interface exposes just one method, **Compare ()**. The Compare method compares two objects. It provides a way to customize the sort order of a collection. It is used in case the type, which is used in sort, cannot be changed (like implementing **IComparable**). The interface **IComparable** must be implemented by the class that should be compared. The **IComparer** interface is independent of the class to compare. That’s why the **Compare()** method defines two arguments that should be compared. The return value is similar to the **CompareTo()** method of the **IComparable** interface.

**IComparable** and **IComparer** also have their generic counter-part, which is type safe.

**IEquatable<T>** defines a generalized method that a value type or class implements to create a type-specific method for determining equality of instances. The interface exposes just one method, **Equals**. The **IEquatable<T>** interface is used by generic collection objects such as **Dictionary***<TKey, TValue>,***List***<T>*, and **LinkedList***<T>* when testing for equality in such methods as **Contains**, **IndexOf**, **LastIndexOf**, and **Remove**. It should be implemented for any object that might be stored in a generic collection.

If you implement **IEquatable<T>**, you should also override the base class implementations of **Object.Equals(Object)**and **GetHashCode** so that their behavior is consistent with that of the **IEquatable<T>.Equals**method. If you do override **Object.Equals(Object)**, your overridden implementation is also called in calls to the static **Equals(System.Object*,* System.Object)**method on your class. In addition, you should overload the op\_Equality and op\_Inequality operators. This ensures that all tests for equality return consistent results.

The **IComparable<T>**interface defines the CompareTo method, which determines the sort order of instances of the implementing type. The **IEquatable<T>** interface defines the Equals method, which determines the equality of instances of the implementing type.

1. **What are the various collections available in C#?**

The collection available in C# are:

***Array and ArrayList***: **Array** is a collection of single type. Before using the array, the size has to be defined. The elements in the array are indexed and can be accessed individually using index. **ArrayList** is a collection which can grow in size as required. An **ArrayList** can store heterogeneous objects. Thus there is an overhead of typecasting the elements of ArrayList. With value type elements, boxing and unboxing is performed while storing and retrieving activity. This degrades performance.

***Queue<T>***: **Queue** represents First In First Out collection. Queues are useful for storing messages in the order they were received for sequential processing. Objects stored in a **Queue**<T> are inserted at one end and removed from the other.

The capacity of a **Queue**<T> is the number of elements the **Queue**<T> can hold. As elements are added to a **Queue**<T>, the capacity is automatically increased as required by reallocating the internal array. The capacity can be decreased by calling TrimExcess.

**Queue**<T> accepts null as a valid value for reference types and allows duplicate elements.

***Stack<T>***: **Stack** represents List In First Out collection. **Stack**<T> is implemented as an array. The capacity of a Stack<T> is the number of elements the **Stack**<T> can hold. As elements are added to a **Stack**<T>, the capacity is automatically increased as required by reallocating the internal array. The capacity can be decreased by calling TrimExcess.

***Dictionary<Tkey, Tvalue>***: **Dictionary** represents a collection of keys and values. The main feature of dictionaries is fast lookup based on keys. Retrieving a value by using its key is very fast, close to O(1), because the Dictionary<TKey, TValue> class is implemented as a hash table.

As long as an object is used as a key in the **Dictionary**<TKey, TValue>, it must not change in any way that affects its hash value. Every key in a **Dictionary**<TKey, TValue> must be unique according to the dictionary's equality comparer. A key cannot be null, but a value can be, if the value type TValue is a reference type.

**Dictionary**<TKey, TValue> requires an equality implementation to determine whether keys are equal. You can specify an implementation of the IEqualityComparer<T> generic interface by using a constructor that accepts a comparer parameter; if you do not specify an implementation, the default generic equality comparer **EqualityComparer**<T>.Default is used. If type TKey implements the **System**.**IEquatable**<T> generic interface, the default equality comparer uses that implementation.

***SortedDictionary<Tkey, Tvalue>***: **SortedDictionary**<Tkey, Tvalue> is a type of Dictionary where the items are sorted based on the keys. It represents a collection of key/value pairs that are sorted on the key.

**SortedDictionary**<TKey, TValue> requires a comparer implementation to perform key comparisons. You can specify an implementation of the IComparer<T> generic interface by using a constructor that accepts a comparer parameter; if you do not specify an implementation, the default generic comparer **Comparer**<T>.**Default** is used. If type TKey implements the **IComparable**<T> generic interface, the default comparer uses that implementation.

***SortedList<Tkey, Tvalue>***: **SortedList**<Tkey, Tvalue> represents a collection of key/value pairs that are sorted by key based on the associated **IComparer**<T> implementation. **SortedList**<TKey, TValue> is implemented as an array of key/value pairs, sorted by the key. Each element can be retrieved as a **KeyValuePair**<TKey, TValue> object. Key objects must be immutable as long as they are used as keys in the **SortedList**<TKey, TValue>. Every key in a **SortedList**<TKey, TValue> must be unique. A key cannot be null, but a value can be, if the type of values in the list, TValue, is a reference type.

**SortedList**<TKey, TValue> requires a comparer implementation to sort and to perform comparisons. The default comparer **Comparer**<T>.Default checks whether the key type TKey implements **System.IComparable**<T> and uses that implementation, if available. If not, **Compare**r<T>.Default checks whether the key type Tkey implements System.IComparable. If the key type TKey does not implement either interface, you can specify a **System.Collections.Generic.IComparer**<T> implementation in a constructor overload that accepts a comparer parameter.

***Lookup<Tkey, Telement>***: **Lookup**<Tkey, Telement> represents a collection of keys each mapped to one or more values. A **Lookup**<TKey, TElement> resembles a **Dictionary**<TKey, TValue>. The difference is that a **Dictionary**<TKey, TValue> maps keys to single values, whereas a Lookup<TKey, TElement> maps keys to collections of values. You can create an instance of a **Lookup**<TKey, TElement> by calling ToLookup on an object that implements **IEnumerable**<T>. There is no public constructor to create a new instance of a **Lookup**<TKey, TElement>. Additionally, **Lookup**<TKey, TElement> objects are immutable, that is, you cannot add or remove elements or keys from a **Lookup**<TKey, TElement> object after it has been created.

***HashSet<T>*** : A **HashSet** holds a set of objects, but in a way that it allows you to easily and quickly determine whether an object is already in the set or not. It does so by internally managing an array and storing the object using an index which is calculated from the hash-code of the object **HashSet** is an unordered collection containing unique elements. It has the standard collection operations Add, Remove and Contains.

The **HashSet**<T> class provides high-performance set operations. A set is a collection that contains no duplicate elements, and whose elements are in no particular order.

1. **What is the using keyword? What happens in case of crash inside the using block?**

Many classes encapsulate unmanaged resources, such as file handles, graphics handles, or database connections. These classes implement System.IDisposable, which defines a single parameterless method named Dispose to clean up these resources. The using statement provides an elegant syntax for calling Dispose on an IDisposable object within a finally block.

The following:

  using (StreamReader reader = File.OpenText ("file.txt"))

{

...

}

is precisely equivalent to:

StreamReader reader = File.OpenText ("file.txt");

try

{

...

}

finally

{

if (reader!= null)

((IDisposable) reader).Dispose();

}

In case of exception inside the using block, the object(s), initialized inside the using statement, is Disposed. However, for that, the object must implement the IDisposable interface.

1. **What is upcast and downcast? What does "as" operator do?**

An object reference can be:

* + Implicitly upcast to a base class reference
  + Explicitly downcast to a subclass reference

Upcasting and Downcasting between compatible reference types performs reference conversions: a new reference is created that points to the same object. An upcast always succeeds; a downcast succeeds only if the object is suitably typed.

public class Asset

{

public string Name;

}

public class Stock: Asset // inherits from Asset

{

public long SharesOwned;

}

Stock msft = new Stock ();

Asset a = msft; // Upcast

Stock s = (Stock) a; // Downcast

In Upcast and Downcast, only references are affected—not the underlying object. A downcast requires an explicit cast because it can potentially fail at runtime:

  House h = new House ();

Asset a = h; // Upcast always succeeds

Stock s = (Stock) a; // Downcast fails: a is not a Stock

If a downcast fails, an InvalidCastException is thrown. This is an example of runtime type checking.

The **as** operator performs a downcast that evaluates to null (rather than throwing an exception) if the downcast fails:

  Asset a = new Asset ();

Stock s = a as Stock; // s is null; no exception thrown

This is useful when you’re going to subsequently test whether the result is null:

if (s != null)

Console.WriteLine (s.SharesOwned);

1. **What are Generics?**

C# has two separate mechanisms for writing code that is reusable across different types: Inheritance and Generics. Whereas inheritance expresses reusability with a base type, Generics express reusability with a “template” that contains “placeholder” types. Generics, when compared to inheritance, can increase type safety and reduce casting and boxing.

public class Stack<T>

{

int position;

T [] data = new T [100];

public void Push (T obj) {data [position++] = obj; }

public T Pop () {return data [--position] ;}

}

Generics exist to write code that is reusable across different types. Suppose we needed a stack of integers, but we didn’t have generic types. One solution would be to hardcode a separate version of the class for every required element type (e.g., IntStack, StringStack, etc.). Clearly, this would cause considerable code duplication. Another solution would be to write a stack that is generalized by using object as the element type:

public class ObjectStack

{

int position;

object [] data = new object [10];

public void Push (object obj) { data[position++] = obj; }

public object Pop () { return data[--position]; }

}

An ObjectStack, however, wouldn’t work as well as a hardcoded IntStack for specifically stacking integers. Specifically, an ObjectStack would require boxing and Downcasting that could not be checked at compile time:

// Suppose we just want to store integers here:

ObjectStack stack = new ObjectStack();

stack.Push ("s"); // Wrong type, but no error!

int i = (int) stack.Pop (); // Downcast - runtime error

What we need is both a general implementation of a stack that works for all element types, and a way to easily specialize that stack to a specific element type for increased type safety and reduced casting and boxing. Generics give us precisely this, by allowing us to parameterize the element type. Stack<T> has the benefits of both ObjectStack and IntStack. Like ObjectStack, Stack<T> is written once to work generally across all types. Like IntStack, Stack<T> is specialized for a particular type— the beauty is that this type is T, which we substitute on the fly.

1. **What are Generics Constraints?**

By default, a type parameter can be substituted with any type whatsoever. Constraints can be applied to a type parameter to require more specific type arguments. These are the possible constraints:

where T: base-class // Base class constraint

where T: interface // Interface constraint

where T: class // Reference-type constraint

where T: struct // Value-type constraint (excludes Nullable types)

where T: new () // Parameterless constructor constraint

where U: T // Naked type constraint

In the following example, GenericClass<T,U> requires T to derive from SomeClass and implement Interface1, and requires U to provide a parameterless constructor:

class SomeClass { … }

interface Interface1 { … }

class GenericClass<T> where T : SomeClass, Interface1

where U : new()

{ … }

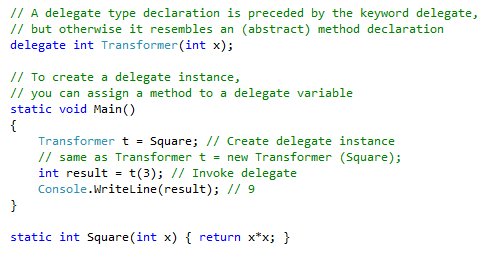
Constraints can be applied wherever type parameters are defined, in both methods and type definitions. A base class constraint or interface constraint specifies that the type parameter must subclass or implement a particular class or interface. This allows instances of that type to be implicitly cast to that class or interface. The class constraint and struct constraint specify that T must be a reference type or (non-Nullable) value type.

1. **What is a delegate? What are Generic Delegates?**

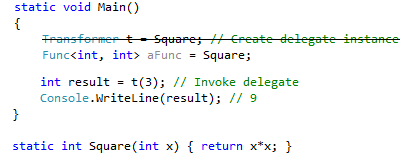
It is a type-safe method reference. With delegates, a program can dynamically call different methods at runtime. Delegate enables late-bound operations such as method invocation and call-back procedures. A delegate dynamically wires up a method caller to its target method. There are two aspects to a delegate: ***type*** and ***instance***. A delegate ***type*** defines a protocol to which the caller and target will conform, comprising a list of parameter types and a return type. A delegate ***instance*** is an object that refers to one (or more) target methods conforming to that protocol.

A delegate instance literally acts as a delegate for the caller: the caller invokes the delegate, and then the delegate calls the target method. This indirection decouples the caller from the target method.

A delegate type declaration is preceded by the keyword delegate, but otherwise it resembles an (abstract) method declaration. For example:



With generic delegates, it becomes possible to write a small set of delegate types that are so general they can work for methods of any return type and any (reasonable) number of arguments. These delegates are the **Func** and **Action** delegates, defined in the System namespace. **Func** is used when the target method returns a value and **Action** is used when the target method does not return any value. The above program can be re-written in generic delegates as:



Note that the delegate declaration is omitted as it is not required anymore, due to generic delegates.

1. **What are Multi-Cast delegates?**

A multicast delegate is a single delegate made up of two or more target methods. The (+) and (+=) operators combine delegate instances. For example:

SomeDelegate d = SomeMethod1;

d += SomeMethod2; // Same as d = d + SomeMethod2;

Invoking d will now call both SomeMethod1 and SomeMethod2. Delegates are invoked in the order they are added. The (−) and (−=) operators remove the right delegate operand from the left delegate operand. For example:

d -= SomeMethod1;

Delegates are *immutable*, so when you call += or −=, you’re in fact creating a new delegate instance and assigning it to the existing variable.

If a multicast delegate has a **nonvoid** return type, the caller receives the return value from the last method to be invoked. The preceding methods are still called, but their return values are discarded. In most scenarios in which multicast delegates are used, they have **void** return types, so this subtlety does not arise.

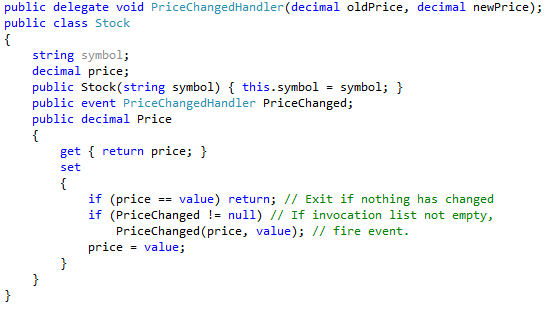
1. **What are events? Do Events Have return type?**

An **event** is a way for a class to provide notifications to clients of that class when some interesting thing happens to an object. Events enable a class or object to notify other classes or objects when something of interest occurs. The class that sends (or raises) the event is called the *publisher* (or *broadcaster*) and the classes that receive (or *handle*) the event are called *subscribers*.

The broadcaster is a type that contains a delegate field. The broadcaster decides when to broadcast, by invoking the delegate. The subscribers are the method target recipients. A subscriber decides when to start and stop listening, by calling += and −= on the broadcaster’s delegate. A subscriber does not know about, or interfere with, other subscribers.

Events are a language feature that formalizes this pattern. An event is a construct that exposes just the subset of delegate features required for the broadcaster/subscriber model. The main purpose of events is to *prevent subscribers from interfering with each other*.

Consider the following example. The Stock class fires its PriceChanged event every time the Price of the Stock changes:



If we remove just the **event** keyword from above example so that PriceChanged becomes an ordinary **delegate** field, the example would give the same results. However, Stock would be less robust, in that *subscribers* could do the following things to interfere with each other:

* + Replace other *subscribers* by reassigning PriceChanged (instead of using the +=operator).
  + Clear all *subscribers* (by setting PriceChanged to null).
  + Broadcast to other *subscribers* by invoking the **delegate**.

These are the differences between delegate and event.

1. **Can events have access modifiers?**

Events are always public as they are meant to serve every one registering to it. But you add can access modifiers in events. You can have events with protected keyword which will be accessible only to inherited classes. You can have private events only for object in that class.

1. **What is an Extension method?**

There are many ways to extend a class. If you have the source for the class, then inheritance is a great way to add functionality to your objects. What if the source code isn't available? Extension methods can help by allowing you to change a class without requiring the source code for the class.

Extension methods are static methods that can appear to be part of a class without actually being in the source code for the class. The string data type does not have any method called ReturnAsDecimal. If we want to have such method to be there with the string class then we can have following code.

public static class MyStaticClass

{

public static decimal ReturnAsDecimal (this string input)

{

decimal result;

return (decimal.TryParse(input, out result)) ? result : 0;

}

}

It can be used as:

String str = “23”;

str.ReturnAsDecimal();

Some points to note:

* The extension method has to be static and must be defined in static class only.
* The first parameter must be the class for which the extension method is being written. Here it is the string class.
* In the extension method you have access to all the public methods and properties of the type being extended.
* If the extension method has the same name as a method in the class, the extension method will never be called. Any instance methods already in the class take precedence.
* Extension methods *do not exhibits* polymorphic *behavior*. An extension method written for a Base class cannot be overridden in child classes. However, the Child classes can have their own extension method of same name.

1. **What is an Anonymous method?**

In order to delegate to work, a method must already exist with the same signature as delegate. However, there is another way to use delegates — with anonymous methods. An anonymous method is a block of code that is used as the parameter for the delegate.

The syntax for defining a delegate with an anonymous method doesn't change. It's when the delegate is instantiated that things change. The following is a very simple console application that shows how using an anonymous method can work:

class Program

{

delegate string DelegateTest(string val);

static void Main()

{

string mid = ", middle part,";

DelegateTest anonDel = delegate(string param)

{

param += mid;

param += " and this was added to the string.";

return param;

};

Console.WriteLine(anonDel("Start of string"));

}

}

The delegate DelegateTest is defined inside the class Program. It takes a single string parameter. When *anonDel* is defined, instead of passing in a known method name, a simple block of code is used, prefixed by the delegate keyword, followed by a parameter.

The benefit of anonymous methods is to reduce the code you have to write. You don't have to define a method just to use it with a delegate. This becomes very evident when defining the delegate for an event. (Events are discussed later in this chapter.) This can help reduce the complexity of code, especially where there are several events defined. With anonymous methods, the code does not perform faster. The compiler still defines a method; the method just has an automatically assigned name that you don't need to know.

A couple of rules must be followed when using anonymous methods. You can't have a jump statement (break, goto, or continue) in an anonymous method that has a target outside of the anonymous method. The reverse is also true — a jump statement outside the anonymous method cannot have a target inside the anonymous method.

Unsafe code cannot be accessed inside an anonymous method. Also, ref and out parameters that are used outside of the anonymous method cannot be accessed. Other variables defined outside of the anonymous method can be used.

If you have to write the same functionality more than once, don't use anonymous methods. In this case, instead of duplicating the code, writing a named method is the preferred way. You only have to write it once and reference it by its name.

1. **What is an Automatic Property?**

Automatic properties are a feature that instructs the compiler to automatically add a default implementation for the get/set methods of a class property. The following code is valid in an interface in previous versions of C#. It is perfectly legal, instead, in a class compiled with the newest C# compiler:

public string CompanyName {get; set ;}

The compiler automatically expands the code as shown here:

private string companyName;

public string CompanyName

{

get { return companyName; }

set { companyName = value; }

}

It is worth noting that automatically generated get/set properties are not equivalent to public fields. From a metadata perspective, properties and fields are quite different entities. The code generated for each is different. The key thing that is going on here is that you delegate the creation of some repetitive code to the compiler. Automatic properties are clearly not an option when you need to do more than just store a value in the get/set methods. Using automatic properties is not a non-return option: at any later time, you can always come back and provide your own get/set methods that contain any logic you need.

1. **What is Type inference and Anonymous Type?**

The var keyword is another interesting new entry in the latest version of C#. Used to qualify a variable, it doesn't indicate a late-bound reference, as its rather popular JavaScript counterpart does. Instead, it merely indicates that the developer doesn't know the type of the variable at the time he's writing the code.

However, the type won't be determined at run time (late-binding); instead, the compiler will infer the type from the expression assigned to the var variable. For this reason, an initial value assignment is required to avoid a compiler error. When the var keyword is used, a strongly typed reference is always generated.

The var keyword enables another cool feature of the latest C# language—anonymous types. Quite simply, an anonymous type is an unnamed type that you define using the same object initializer syntax mentioned earlier:

var person = new { FirstName="Nancy", LastName="Davolio", Age=28 };

For the CLR, anonymous and named types are exactly the same entity. Anonymous types can be used in a variety of scenarios, but they have been introduced primarily to support LINQ queries. This can be used in a place where we want to return multiple values from a function instead of a single value.

1. **What is Dynamic keyword? How it is different from var and Object?**

The **dynamic** type enables the operations in which it occurs to bypass compile-time type checking. Instead, these operations are resolved at run time. The **dynamic** type simplifies access to COM APIs such as the Office Automation APIs, and also to dynamic APIs such as IronPython libraries, and to the **HTML** Document Object Model (**DOM**).

Type **dynamic** behaves like type **object** in most circumstances. Structurally, there is no difference between dynamic reference and object reference. A dynamic reference simply enables the dynamic operations on the object it points to. However, operations that contain expressions of type **dynamic** are not resolved or type checked by the compiler. The compiler packages together information about the operation, and that information is later used to evaluate the operation at run time. As part of the process, variables of type **dynamic** are compiled into variables of type **object**. Therefore, type **dynamic** exists only at compile time, not at run time.

The **var** is used for implicitly type local variables and anonymous type. The compiler knows the type of var during compile time. In case of mismatched assignments (like, assigning string to int), var throws compile time error. However, in case of dynamic, the mismatch exception is thrown at runtime.

1. **What are Attributes?**

An **attribute** is a declarative tag that is used to convey information to runtime about the behaviors of various elements like classes, methods, structures, enumerators, assemblies etc., in a program. We can add declarative information to a program by using an **attribute**. A declarative tag is depicted by square (**[ ]**) brackets placed above the element it is used for.

Attributes are used for adding metadata, such as compiler instruction and other information such as comments, description, methods and classes to a program. An attribute is defined by a class that inherits (directly or indirectly) from System.Attribute abstract class.

1. **What is Yield statement?**

# 

# *OOPS*

1. **What are different properties provided by Object-oriented systems?**

Following are characteristics of Object Oriented System’s:-

**Abstraction**: It means focusing on the essential, inherent aspects of any entity and ignoring its accidental properties. In system development, this means focusing on what an object is and does, before deciding how it should be implemented.

**Encapsulation**: It is also known as Information Hiding. It consists of separating the external aspects of an object, which are accessible to other objects, from the internal implementation details of the object, which are hidden from other objects. Encapsulation prevents a program from becoming so interdependent that a small change has massive ripple effects. The implementation of an object can be changed without affecting the application that uses it.

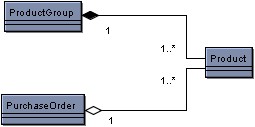
**Inheritance**: It is the sharing of attributes and operations among classes based on a hierarchical relationship. A class can be defined broadly and then refined into successively finer subclasses. Each subclass inherits all of the properties of its super-class and adds its own unique properties.

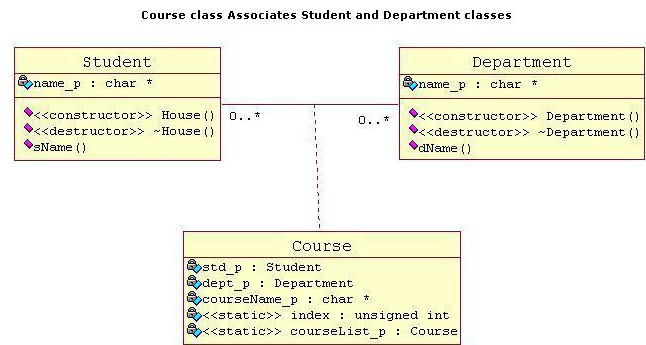
**Polymorphism**: It means that the same operation may behave differently on different classes. It is the capability of a program to carry out dynamic operations by implementing methods of multiple derived classes through a common base class reference.

1. **What are association, aggregation and Composition?**

In object oriented world, objects have relation and hierarchies in between them. There are basically three kind of relationship in Object Oriented world:-

**Association** is a simple structural connection or channel between classes and is a relationship where all objects have their own lifecycle and there is no owner. Let’s take an example of Student and Department. Multiple students can associate with a single Department and single student can associate with multiple Departments, but there is no ownership between the objects and both have their own lifecycle. Both can create and delete independently.





**Aggregation** is a specialize form of Association where all object have their own lifecycle but there is an ownership like parent and child. Child object cannot belong to another parent object at the same time. We can think of it as "has-a" relationship. Let’s take an example of Department and teacher. A single teacher cannot belongs to multiple departments, but if we delete the department, then the teacher object will not destroy.

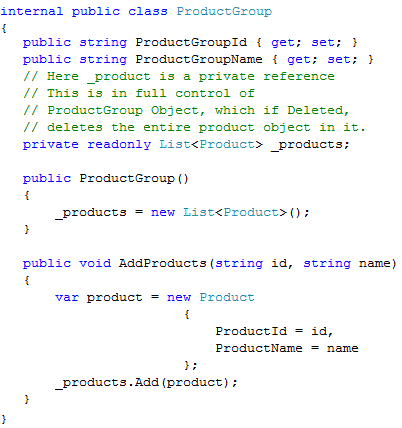
**Composition** is a type of association where the lifetime of an object depends on the lifetime of another object. It indicates that one object belongs to the other. A polygon is made up of several points. If the polygon is destroyed, so are the points.

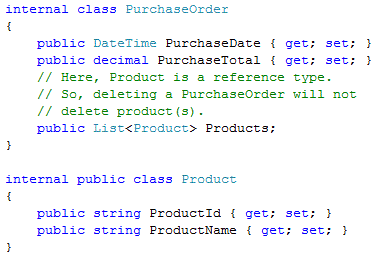
The example shows an aggregation and a composition.

The **composition** is represented by the solid diamond.  *Products is compose and is contained by ProductGroup*.  This means that if a *ProductGroup* is destroyed, the *Products* within the group are destroyed as well.

The **aggregation** is represented by the hollow diamond.  *Products is compose without belonging to PurchaseOrder*. If a *PurchaseOrder* is destroyed, the *Products* still exist.

If you have trouble remembering the difference between **composition** and **aggregation**, just think of the alphabet.  Composition means destroy and the letters 'c' and 'd' are next to each other.





1. **What is Method Overriding and Method Hiding (shadowing)?**

Whenever, there’s an inheritance involved, there are only two ways to deal with common signature methods,

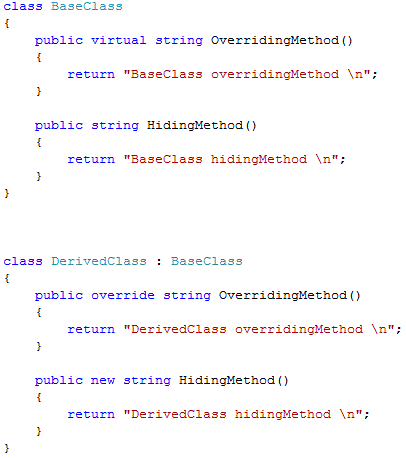
* Method Overriding using **virtual** & **override** or **Abstract** **& Override**
* Method Hiding using **new**.

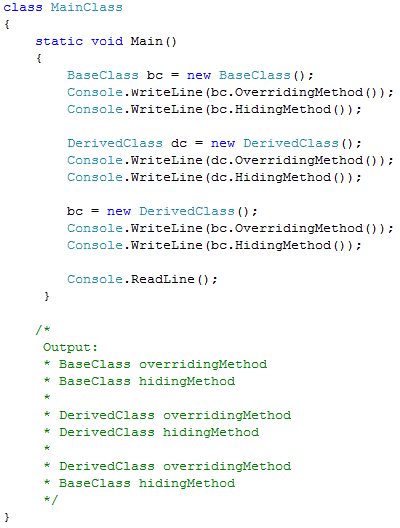
**Method Hiding** is also known as **shadowing**. This is achieved using **new** operator. **Method hiding**, as the name suggest, hides the base class method. The user of the derived class will be able to see only the derived version of the method. This is the default behavior, meaning that if nothing is specified then its method hiding.

**Method Overriding** provides an alternate implementation to the base class method in the derived class. The version of a method that is executed will be determined by the [object](http://en.wikipedia.org/wiki/Object_(computer_science)) that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.

When disambiguating when which method will be called, I like to think of shadowing vs. overriding with the following

* **Shadowing**: The method called depends on the type of the reference at the point the call is made. (**S-R**)
* **Overriding**: The method called depends on the type of the object at the point the call is made. (**O-O**)





1. **What are Static(C#) variables?**

Static classes are used when a class provides functionality which is not specific to any instance. In short if you want an object to be shared between multiple instances you will use a static/Shared class.

Following are features of Static classes:-

* They cannot be instantiated. By default an object is created on the first method call to that object.
* Static classes cannot be inherited.
* Static classes can have only static members.
* Static classes can have only static constructor.

1. **What is the different accessibility levels defined in .NET?**

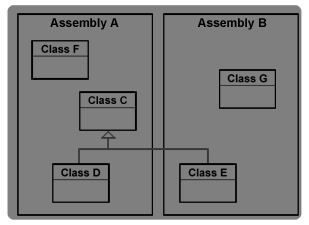
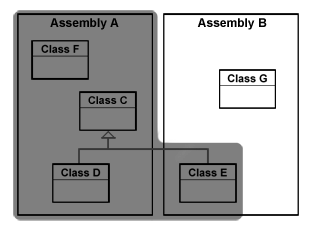
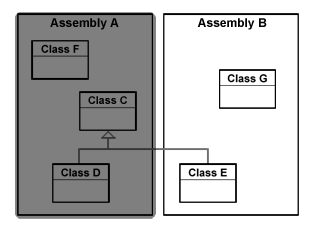
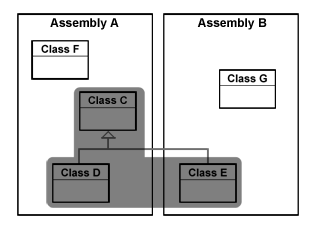
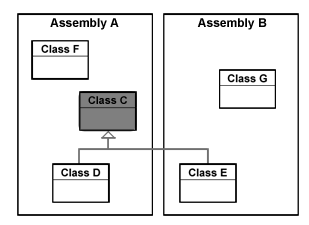
With the use of appropriate access modifiers, we can manage class encapsulation. It decides who can access class members.

**Private**: It is the most restrictive. This allows members, only within a class, to access another member marked as private. Anyone outside the class cannot access this member. They won’t even know about its existence. It is useful because it allows modification of a private member implementation without anyone knowing.

**Protected**: - It is a little restrictive than private. Users may know the member is in a class, but they cannot access protected members directly. The only way to use a protected member is through inheritance. A derived class has full access to protected class members. This is regardless of the depth of the hierarchy. The protected member need not be in the derived class’ immediate base class. Protected access is good for optimization when a derived class needs frequent access to base class information.

**Internal**: - It means the member is accessible to other types that are defined in the same assembly. Internal members are accessible only within files in the same assembly (.dll). In other words, access is limited exclusively to classes defined within the current project assembly.

**Protected** **Internal**: - It is a combination of protected and internal modifiers. A protected internal member is accessible from all code within the same assembly or from code in a derived class. The members in class A that are marked protected internal are accessible to



* Methods of class A,
* To methods of classes derived from class A, and also
* To any class in A's assembly.

This is effectively protected OR internal (There is no concept of protected AND internal.)

**Public:** - All members have access in all classes and assemblies.

1. **Some points regarding access modifiers.**

* Namespaces can’t have access modifiers. They’re always public.   
  So: public namespace AwesomeNewNamespace { } will produce a compile-time error.
* Classes can be either public or internal but not protected or private.   
  So: private class AwesomeNewClass { } will produce a compile-time error. This makes sense if you think about it how exactly would you use a private or a protected class?
* Interfaces, like classes, can be either public or internal but not protected or private.
* Methods defined in interfaces can’t have access modifiers.
* Enums are always public. They can’t have access modifiers either.
* By default, classes are internal.  
  So: class NotSoCool { }   
  is equivalent to  
  internal class NotSoCool { }
* By default, methods in class are private. So   
  void SomeAwesomeMethod () { }   
  is equivalent to   
  private void SomeAwesomeMethod () { }.   
  Same goes for variables (fields): private unless specified otherwise

1. **Can you prevent a class from overriding?**

If you define a class as “Sealed” in C# and “NotInheritable” in VB.NET you can inherit the class any further.

1. **What is immutable class?**

An immutable object is one whose state cannot be altered—externally or internally. The fields in an immutable object are typically declared read-only and are fully initialized during construction. If the state of an object is needed to be altered then a new object is cloned out of the existing object and then the existing object is discarded or kept for future reference, in a different variable. Thus the new object replaces the old one. Immutable are thread safe.

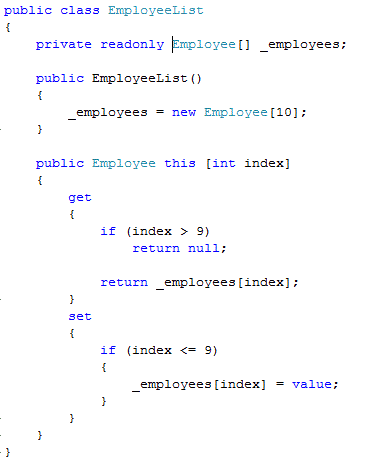
1. **What is the difference between System.String and System.StringBuilder classes?**

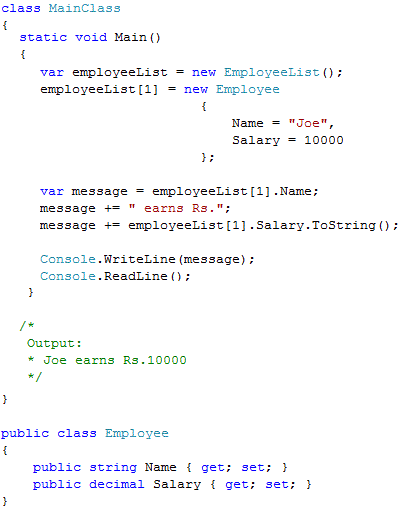
**System.String** is immutable; **System.StringBuilder** can have mutable string where a variety of operations can be performed. An **immutable** class is a class which does not allow to alter its state. Once created, the object of such class cannot be altered. In case an altered version of the object is required, then the whole object is created again. And the old one is discarded. **String** is **immutable** in the sense that every time the string is altered, a new memory space is created and the old one is orphaned. As with immutable objects, strings are also thread safe.

1. **What is Indexer? Can we have static indexer in C#?**

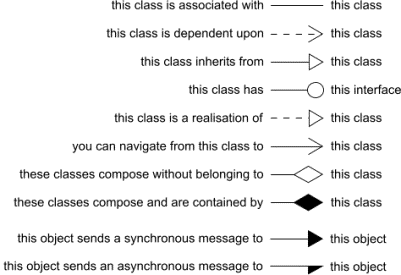
An indexer allows you to access members of a class similar to the way the arrays are used.

* Indexer Concept is object act as an array.
* Indexer an object to be indexed in the same way as an array.
* Indexer modifier can be private, public, protected or internal.
* The return type can be any valid C# types.
* Indexers in C# must have at least one parameter. Else the compiler will generate a compilation error.
* Indexer cannot be Static





1. **What are some commonly used UML symbols?**



http://www.somanyword.com/2014/06/uml-relationships-like-association-aggregation-compositionabstraction-generalization-specialization-realization-and-dependency/

# 

# *Patterns*

1. **What are design patterns? (A) Can you list down all patterns and there classification?**

Design patterns are recurring solution to recurring problems in software architecture. There are three basic classifications of patterns Creational, Structural and Behavioral patterns.

**Creational Patterns**

* Abstract Factory:- Creates an instance of several families of classes
* Builder: - Separates object construction from its representation
* Factory Method:- Creates an instance of several derived classes
* Prototype:- A fully initialized instance to be copied or cloned
* Singleton:- A class of which only a single instance can exist

*Note: - The best way to remember Creational pattern is by* ***ABFPS*** *(Abraham Became First President of States).*

**Structural Patterns**

* Adapter:-Match interfaces of different classes.
* Bridge:-Separates an object’s interface from its implementation.
* Composite:-A tree structure of simple and composite objects.
* Decorator:-Add responsibilities to objects dynamically.
* [Façade](#Facade_Pattern):-A single class that represents an entire subsystem.
* Flyweight:-A fine-grained instance used for efficient sharing.
* Proxy:-An object representing another object.

*Note: To remember structural pattern best is (ABCDFFP)*

**Behavioral Patterns**

* Mediator:-Defines simplified communication between classes.
* Memento:-Capture and restore an object's internal state.
* Interpreter:-A way to include language elements in a program.
* Iterator:-Sequentially access the elements of a collection.
* Chain of Responsibility:-A way of passing a request between a chain of objects.
* Command:-Encapsulate a command request as an object.
* State:-Alter an object's behavior when its state changes.
* Strategy:-Encapsulates an algorithm inside a class.
* Observer:-A way of notifying change to a number of classes.
* Template Method:-Defer the exact steps of an algorithm to a subclass.
* Visitor:-Defines a new operation to a class without change.

*Note: - Just remember Music....... 2 MICS On TV (MMIICCSSOTV).*

1. **What is Strategy Pattern?**

In Software design pattern, the strategy pattern enables an algorithm’s behavior to be selected at runtime. The Strategy pattern:

* Defines a family of algorithms,
* Encapsulates each algorithm, and
* Makes the algorithms interchangeable within that family.

**Strategy** lets the algorithm vary independently from clients that use it. For instance, a class that performs validation on incoming data may use a strategy pattern to select a validation algorithm based on the type of data, the source of the data, user choice, or other discriminating factors. These factors are not known for each case until run-time, and may require radically different validation to be performed. The validation strategies, encapsulated separately from the validating object, may be used by other validating objects in different areas of the system (or even different systems) without code duplication.

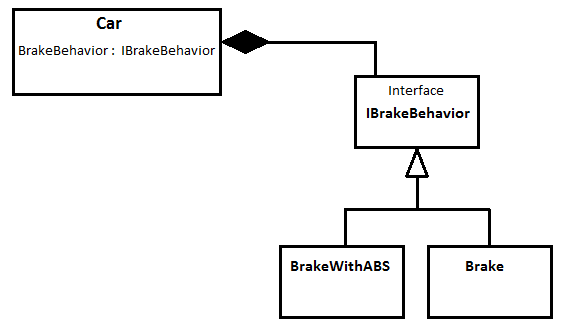
*According to the strategy pattern, the behaviors of a class should not be inherited. Instead they should be encapsulated using interfaces*.

As an example, consider a car class. Two possible functionalities for car are *brake* and *accelerate*. Since *accelerate* and *brake* behaviors change frequently between models, a common approach is to implement these behaviors in subclasses. This approach has significant drawbacks: *accelerate* and *brake* behaviors must be declared in each new Car model. The work of managing these behaviors increases greatly as the number of models increases, and requires code to be duplicated across models. Additionally, it is not easy to determine the exact nature of the behavior for each model without investigating the code in each.

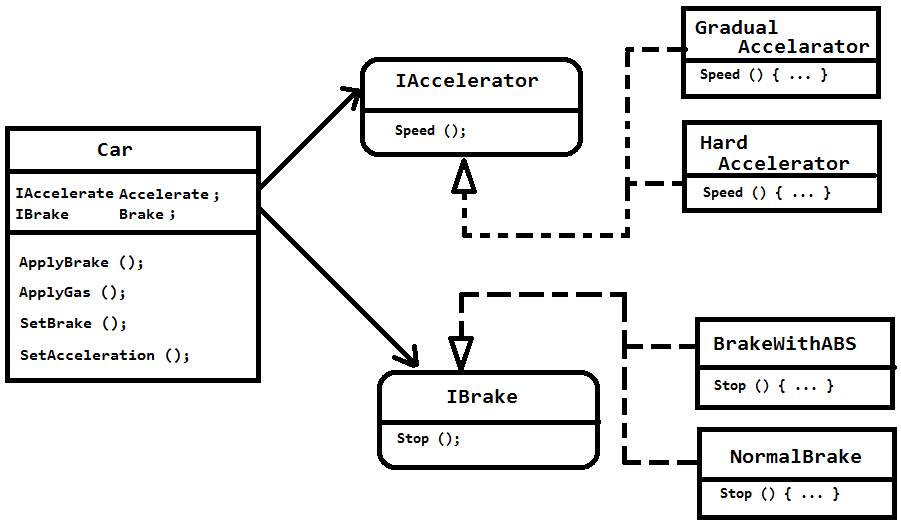
The strategy pattern uses composition instead of inheritance. In the strategy pattern, behaviors are defined as separate interfaces and specific classes that implement these interfaces. This allows better decoupling between the behavior and the class that uses the behavior. The behavior can be changed without breaking the classes that use it, and the classes can switch between behaviors by changing the specific implementation used without requiring any significant code changes. Behaviors can also be changed at run-time as well as at design-time. For instance, a car object’s brake behavior can be changed from BrakeWithABS () to Brake () by changing the brakeBehavior member to: brakeBehavior = new Brake ();



UML of Strategy



Brake Behavior Strategy

Implementation (here **IAccelerator** and **IBrake** acts as Strategy)

1. **Explain Observer pattern**

The **observer** pattern is based on **Publisher**-**Subscriber** model. This pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependencies are notified and updated automatically. Traditionally, an **observer** pattern is implemented by having a **publisher** that maintains a list of its **subscriber**. Whenever there is a change in the state of the **publisher**, it runs through the list of **subscribers** and notifies them about the change. A **subscriber** can register and un-register itself from the publisher.

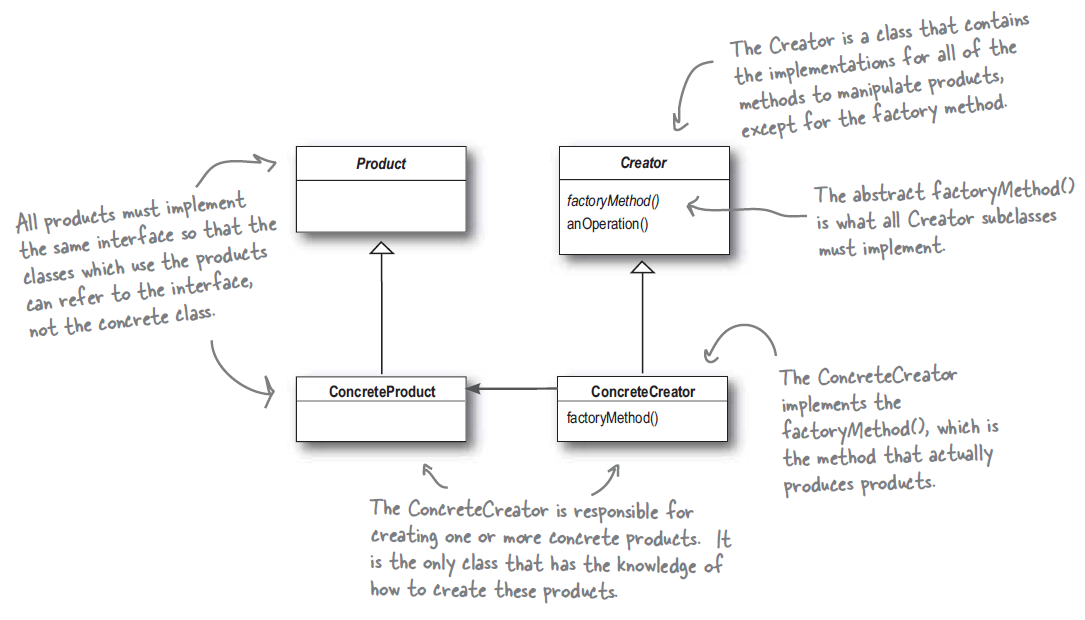
The **Observer** pattern can also be implemented using the Event construct. In this the **publisher** exposes an **event handler**, which is used by subscriber to attach a member method so as to handle the event.

1. **What is Factory Pattern?**

It defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. The Scenarios where factory pattern is applicable, are:

* A class can't anticipate the class of object it must create.
* A class wants its subclasses to specify the objects it creates.
* Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

[http://msdn.microsoft.com/en-us/library/Ee817667%28pandp.10%29.aspx]



**Figure 2: Factory pattern physical model**

The Factory pattern gives us a way to encapsulate the instantiations of concrete types. In the above UML diagram, the abstract Creator gives an interface with a method to create objects, also known as Factory Method. Any methods implemented in the abstract Creator are written to operate on the products produced by the Factory method. Only subclasses actually implement the factory method and create Products. Thus the abstract Creator is shielded from the actual object creation.

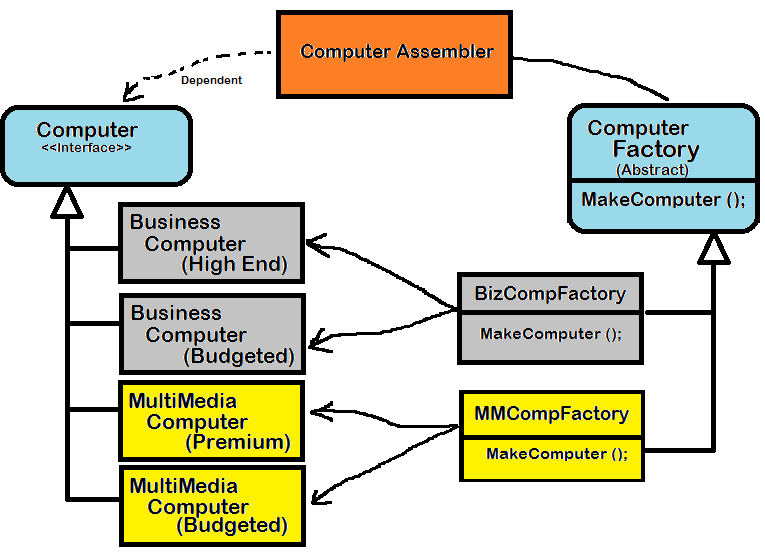
The **Factory** completely abstracts the creation and initialization of the product from the client. This indirection enables the client to focus on its discrete role in the application without concerning itself with the details of how the product is created. Thus, as the product implementation changes over time, the client remains unchanged.

While this indirection is a tangible benefit, the most important aspect of this pattern is the fact that the client is abstracted from both the type of product and the type of factory used to create the product. Presuming that the product interface is invariant, this enables the factory to create any product type it deems appropriate. Furthermore, presuming that the factory interface is invariant, the entire factory along with the associated products it creates can be replaced in a wholesale fashion. Both of these radical modifications can occur without any changes to the client.

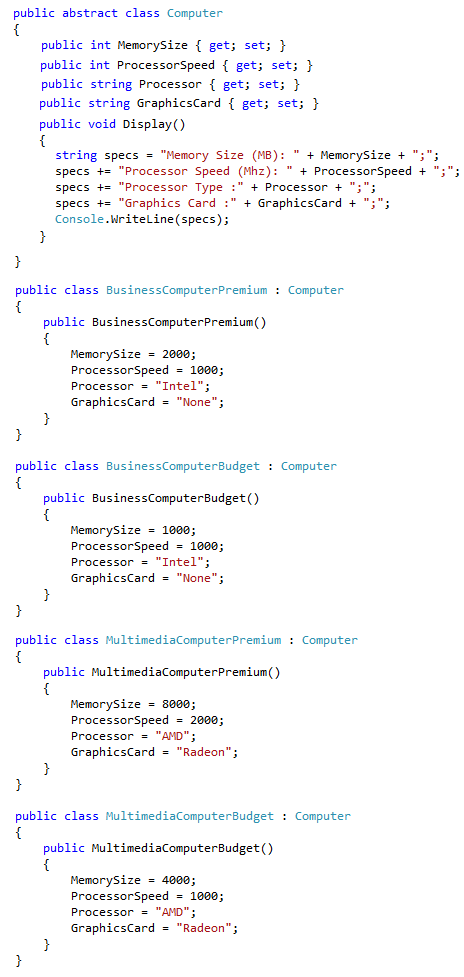
As shown in Figure, the Client, *Computer Assembler*, uses an instance of a concrete subclass of **Factory** (*BizCompFactory or MMCompFactory*) to create an instance of a concrete Product subclass (*Business Computers or Multimedia Computers*). Since all variables, parameters, and method return values are typed to the *Factory* and *Product* abstract classes, the Client is unaware of the introduction of these subclasses. This enables the introduction of new *Factory* and *Product* subclasses as requirements warrant; nothing needs to change within the Client.

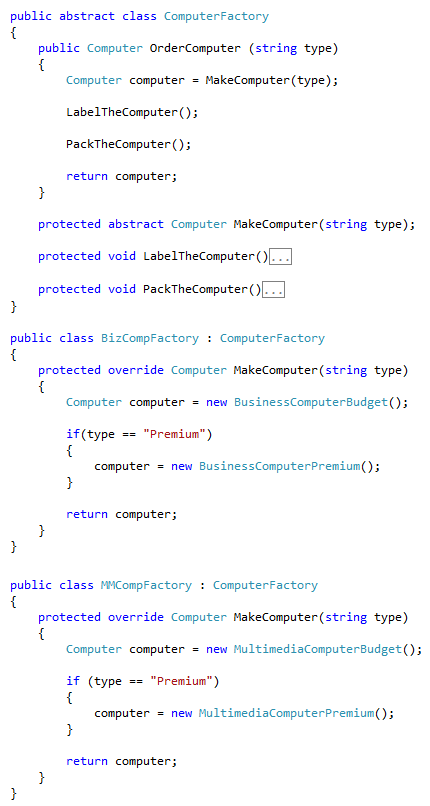
The below written code is the implementation of the above diagram. There are two abstract classes, Computer and Computer Factory. The abstract class Computer is implemented by four classes. Two classes each of different behavior i.e. Business and Multimedia computers.

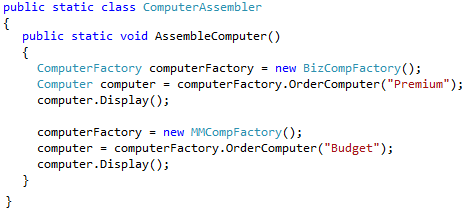
On the other hand, the abstract Computer factory has two concrete implementations. Each of the classes has their own implementation of the factory method, Make-Computer. In the method, Make-Computer, the factory classes decide which type computer to make. There could have been just one concrete factory, to make all the four types of computer. However, there can be differences between two concrete factories in implementing other members of the parent Abstract factory class.



The main theme of factory pattern is to differ the creation of concrete objects to the concrete sub-classes. The client will be still referring to the abstract classes, thus isolating itself from the underneath mechanism and changes.





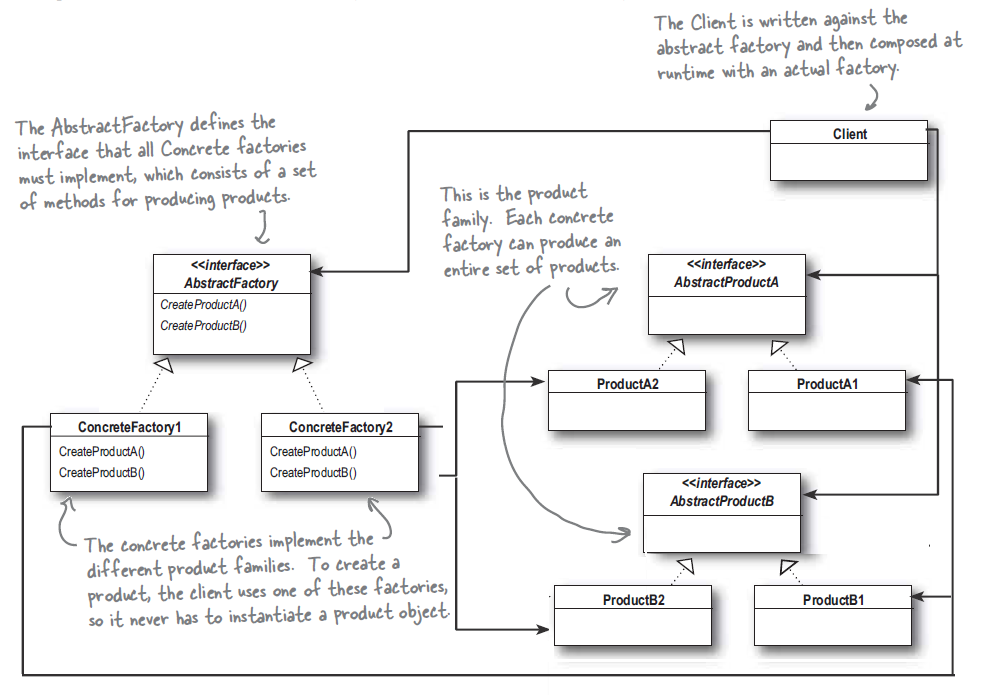


1. **What is Abstract Factory Pattern?**

An abstract factory provides an interface for creating families of related objects without specifying their concrete classes. Sometimes one wants to construct an instance of one of a suite of classes, deciding between the classes at the time of instantiation. In order to avoid duplicating the decision making everywhere an instance is created, we need a mechanism for creating instances of related classes without necessarily knowing which will be instantiated. We use abstract factory to create instances of concrete classes. The class of the resultant instance is unknown to the client of the abstract factory. (See http://www.codeguru.com/csharp/.net/net\_general/patterns/article.php/c4673)

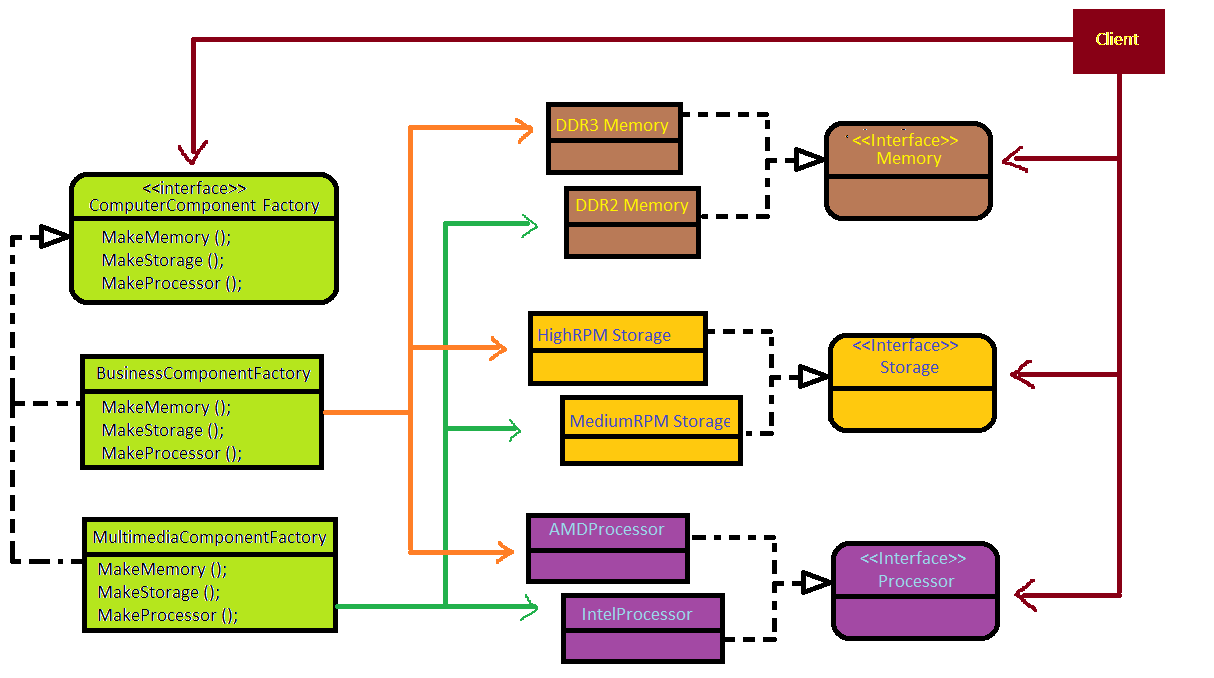
**Applicability**

* Need to abstract from details of implementation of products -The system shall be independent of how its constituent pieces are created, composed, and represented.
* Need to have multiple families of products - The system shall be configured with one of multiple families of products.
* Need to enforce families of products that must be used together - A family of related product objects is designed to be used together, and you need to enforce this constraint.
* Need to hide product implementations and just present interfaces - You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.



**Characteristics:**

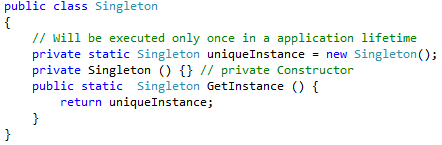
* An abstract factory is an object maker.
* It typically can produce more than one type of object.
* Each object that it produces is known to the receiver of the created object only by that object's interface, not by the object's actual concrete implementation.
* The different types of objects that the abstract factory can produce are related -- they are from a common family.
* An abstract factory isolates concrete classes.
* It makes exchanging product families easy.
* It promotes consistency among products.
* It supports adding new kinds of products and their families.

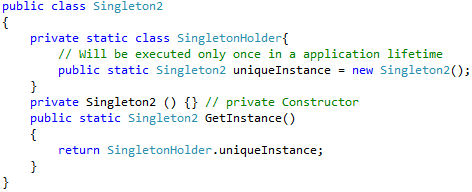


The above diagram is an extension to the Factory pattern scenario. Instead of making just different types of computers, a variety of computer components are made. Thus, an abstract factory deals with a suite of products whereas the Factory pattern deals with just one type of product.

1. **How can we implement singleton pattern in .NET?**

Singleton pattern mainly focuses on having one and only one instance of the object running. Example a windows directory service which has multiple entries but you can only have single instance of it throughout the network. In its simplest form, singleton class is implement by declaring a private constructor and a private static reference variable. It also exposes a static method, GetInstance, which actually creates the instance. However, in multi-threaded environment the traditional implementation fails. If two threads accesses the method GetInstance, then two object are created thus failing the singleton pattern purpose. In order to make it thread-safe, there are two ways to achieve it. One way is to lock the method GetInstance. This is a costly affair as it puts the performance over head. The other way is to create the instance eagerly rather than lazily. Below are the two ways to implement thread-safe Singleton. In second style, there’s a class that wraps the instantiation code, rest same as first one.



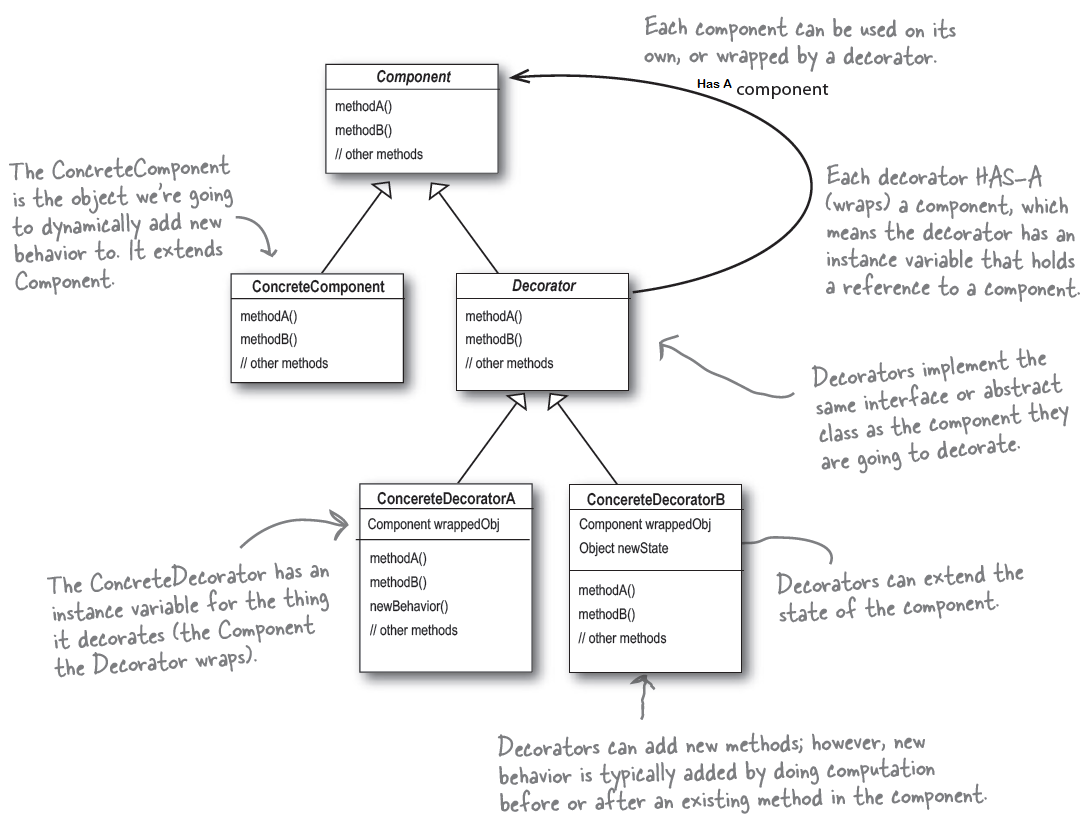


1. **Explain Decorator pattern.**

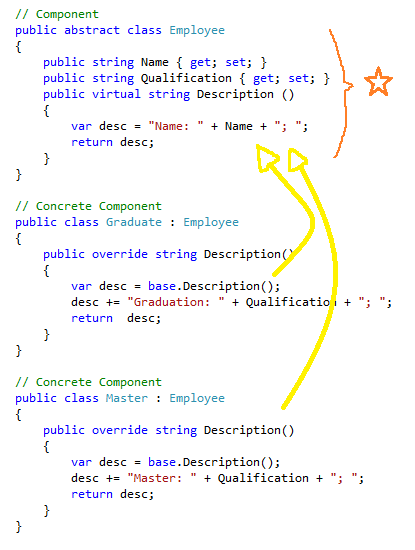
The decorator pattern is a design pattern that allows behavior to be added to an individual object around the original object, either statically or dynamically without affecting the behavior of other objects from the same class.

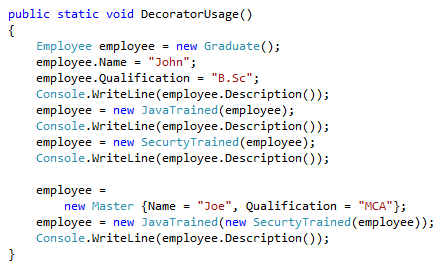
A decorator pattern can be implemented by using the following steps:

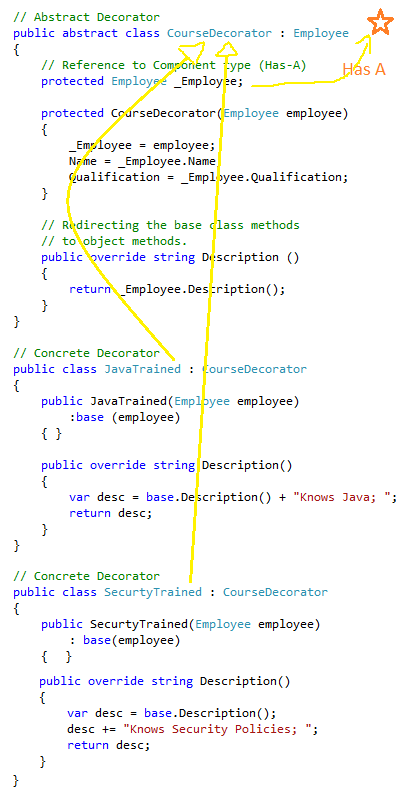
* Subclass the original "Component" class into a "Decorator" class (see UML diagram);
* In the Decorator class, add a reference variable to the Component as a field (indicated by *Has a Component*);
* Pass a Component to the Decorator constructor to initialize the Component reference variable;
* In the Decorator class, redirect all "Component" methods to the "Component" reference methods; and
* In the ConcreteDecorator class, override any Component method(s) whose behavior needs to be modified.

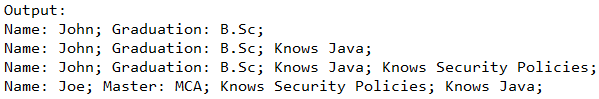


UML of Decorator Pattern







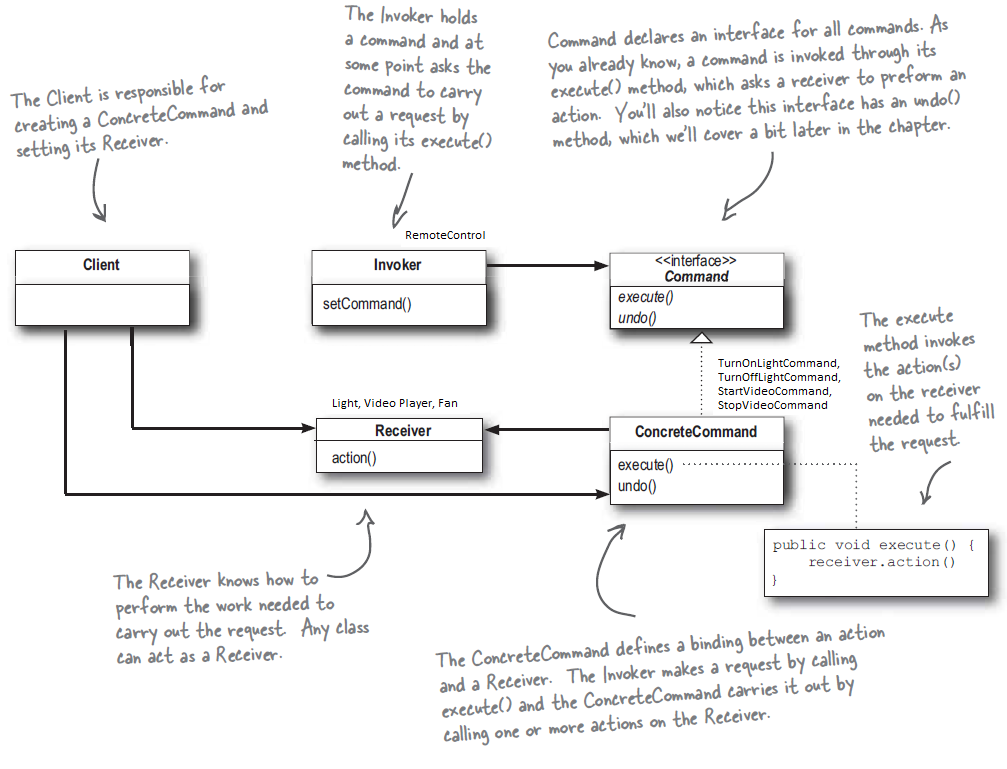


In the above program, all the classes directly or indirectly inherits Employee class. That is why, the same reference variable is used for passing as well as assigning to the decorator object. This way the object creation can be chained as well. An employee of type Graduate or Master can be decorated with different types of learnings like Java or Security.

1. **What is Command Pattern?**

The Command pattern encapsulates a request as an object, there by letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.

Four terms always associated with the command pattern are command, receiver, invoker and client. A command object encapsulates a request by binding together a set of action on a specific receiver. To achieve this, it packages the actions and the receiver up into an object that exposes just one method, execute (). When called, execute () causes the actions to be invoked on the receiver. From outside, no there objects really know what actions get performed on what receiver. They just know that if they call execute () method, their request will be serviced.



1. **How do you implement prototype pattern in .NET?**

The Prototype pattern comes under the classification of Creational Patterns, which deals with the best way to create objects. This helps to copy or clone the existing objects to create new ones rather than creating from the scratch. (http://www.devarticles.com/c/a/C-Sharp/Creational-Patterns-in-C-sharp/4/)

The prototype pattern is used when creating an instance of a class is very time consuming or complex in some way. Then rather than creating more instances, it is possible to make copies of the original instances and modifying them as appropriate.

When we are not in a position to call a constructor for an object directly, we could alternatively clone a pre-existing object (a prototype) of the same class. When there are many subclasses that differ only in the kind of objects they create a Prototype Pattern can be used to reduce the number of subclasses by cloning a prototype. Prototype Design Pattern helps in reducing number of classes.



**Example: How to do Deep Copy**

For example suppose we have to do say Sales Analysis on a set of data in the database. Normally we will create an object encapsulating this data and do the Sales Analysis. Suppose now we have to do another type of analysis say Promotion Analysis on the same data. Now instead of creating another object corresponds to the data from the scratch, we can clone the existing object and do the analysis. This is one of the classical use of prototype pattern.

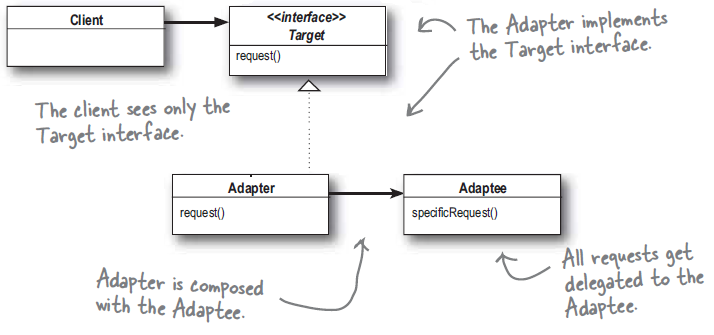
Cloning is achieved by using ICloneable of the System namespace. It has a “Clone” method which actually returns the reference of the same copy. Clone method allows a Shallow copy and not a deep copy. In Shallow copy, if you make changes to the cloned object it actually makes the change on the main object itself.

So how is deep copy achieved, by using “ISerializable” interface? In order to do so, first serialize the object and then De-Serialize back to a complete new copy. Now any changes to this new copy do not reflect on the original copy of the object, this is called as Deep copy.

1. **Explain Adapter Pattern?**

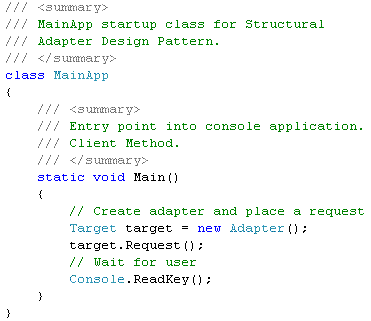
The Adapter pattern converts the interface of a class into another interface the client expects. The Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

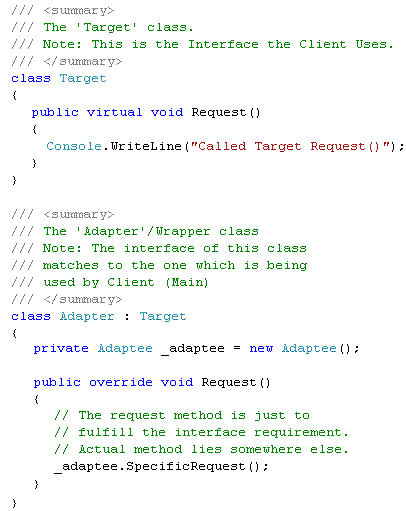
The adapter pattern allows us to use a client with an incompatible interface by creating an Adapter that does the conversion. This acts to decouple the client from the implemented interface, and if we expect the interface to change over time, the adapter encapsulates that change so that the client doesn’t have to be modified each time it needs to operate against a different interface.



The Adapter pattern is similar to Decorator pattern. While adapter deals with two main interfaces, the decorator deals with just one interface. All the classes that are created to work-out a decorator pattern, inherits from a single abstract class.

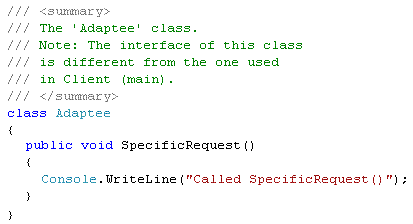
Adapter and Decorator, even though look similar, however, the intention of both are different. The intent of a decorator is to extend the behavior of the existing class. Whereas, Adapter provides a means to combine two different interfaces.





Output:

**Called Specific Request**

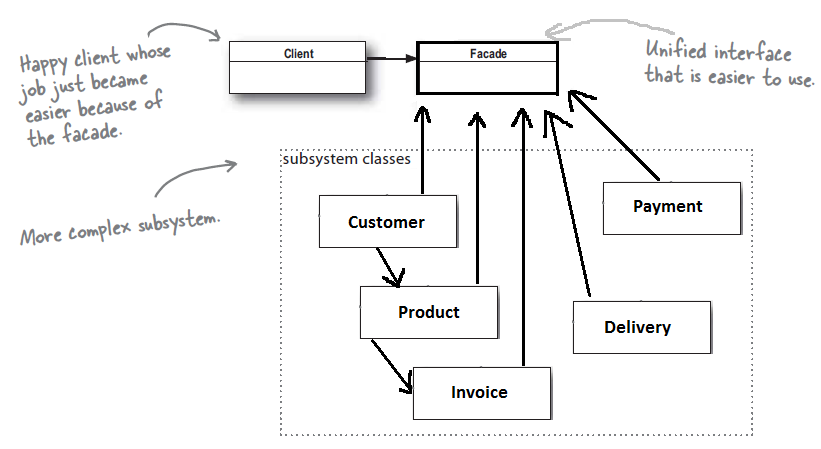


1. **Explain Façade pattern?**

The **Façade** pattern provides a unified interface to a set of interfaces in a subsystem. **Façade** defines a higher level interface that makes the subsystem easier to use.

In **Façade** pattern, a class is created that simplifies and unifies a set of more complex classes that belong to a subsystem.

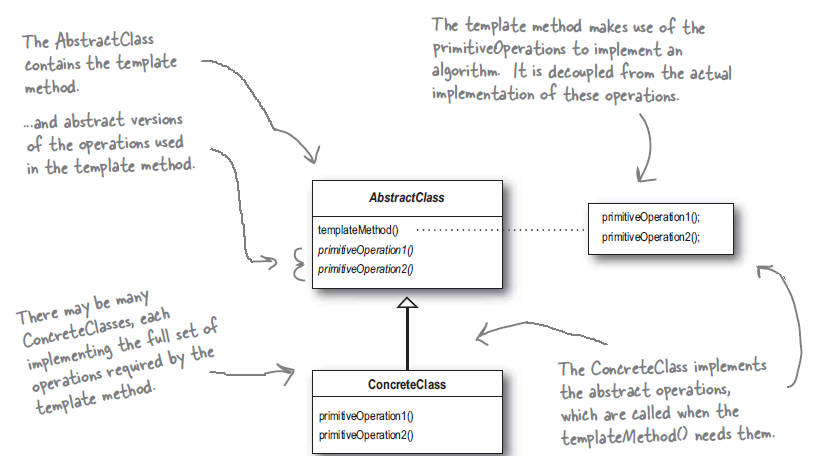
For example, consider an order processing system consisting of following classes: *Customer*, *Product*, *Payment*, *Delivery* and *Invoicing*. To make an order, the system has to create a *Customer* object, assign the *products* to the *customer*, collect the *payment*, generate an *invoice* and finally *deliver* the *product*. This is a complex activity where a single step omission in step can be costly.



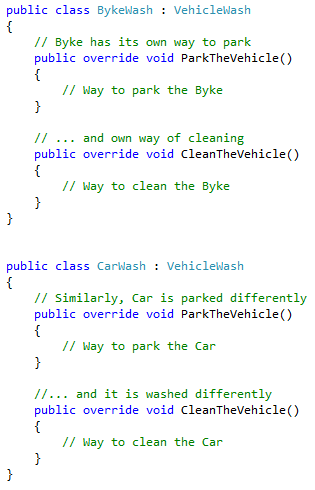
Instead of making the client do all the steps, we create a class say *OrderProcessor*. The class *OrderProcessor* will be composed of all the above class objects: Customer, Product, Payment, Delivery and Invoice. It will expose just one method, say, *ProcessOrder* and everything will be encapsulated in the method. Thus the client is abstracted from the details.

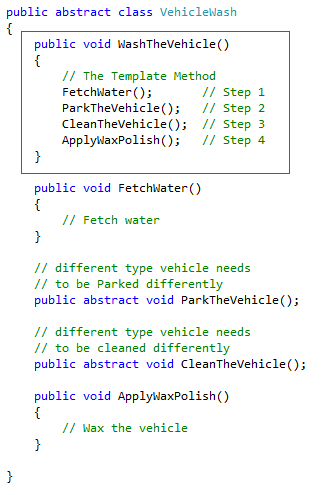
1. **Explain Template Method**

The Template method pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.



Here, a template means a method that defines an algorithm as a set of steps. While some of the steps are defined in the abstract class itself, other steps are declared as abstract and is left to the subclass to define those steps. This ensures that the steps of an algorithm stays intact, however the behavior of the algorithm might differ in specialized classes.





1. **What is SOLID?**

The acronym SOLID stands for

* **S**ingle responsibility principle,
* **O**pen and close principle,
* **L**iskov substitution principle,
* **I**nterface segregation principle and
* **D**ependency Inversion principle.

1. **What is Single responsibility principle?**

The Single responsibility principle states that every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.

In this context a responsibility is considered to be one reason to change. This principle states that, if we have 2 reasons to change for a class, then we have to split the functionality in two classes. Each class will handle only one responsibility and on future if we need to make one change we are going to make it in the class which handle it. When we need to make a change in a class having more responsibilities the change might affect the other functionality of the classes.

1. **What is Liskov substitution principle?**

The Liskov’s substitution principle states that if a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module.

We must make sure that the new derived classes just extend without replacing the functionality of old classes. Otherwise the new classes can produce undesired effects when they are used in existing program modules. For example, if a program has been coded for calculating area of a shape then the shape object can be replaced by more specialized sub-type of shape like square or circle.

1. **What is Interface segregation principle?**

The **Interface Segregation Principle** (**ISP**) states that clients should not be forced to implement interfaces they don't use. Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving one sub-module.

The **ISP** splits interfaces which are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called role interfaces. The **ISP** is intended to keep a system decoupled and thus easier to refactor, change, and redeploy.

1. **What is Dependency Inversion principle?**

The Dependency Inversion principle states that an application should depend upon abstractions, but not the concrete classes. This principle is similar to “Program to an interface, not an implementation”. However, the dependency inversion principle emphasizes more on the abstractness. It suggest that the high level component should not depend upon low level component. Rather, they should both depend upon the abstractions. The Factory pattern is one of the ways to implement the Dependency Inversion principle.

For example, say, there is a class called *ComputerStore*. This class has a method that creates computer of different type like *Business computer*, *Multimedia computer* etc. The *Business computer* and *Multimedia computer* are low level components. The Class *ComputerStore*, a high level component, is now dependent on the low level component for its primary function.

So, instead, if we create an abstract class called Computer, then we can derive all the low level components (Business Computers and Multimedia computers) from the abstract Computer class.

1. **What is Inversion of Control (IOC)?**

The **Inversion of control** (IoC) describes a design in which custom-written (Developer-written) portions of a computer program receive the flow of control from a generic, reusable library. A software architecture, with this design, inverts control as compared to traditional procedural programming where the custom code, that expresses the purpose of the program, calls into reusable libraries to take care of generic tasks. With inversion of control, it is the reusable code that calls into the custom, or task-specific, code.

IOC is the main theme of **Framework** and it distinguishes a **Framework** and a **Library**. A **library** is essentially a set of functions to which a developer’s code usually make calls. Each call does some work and return control to the client. In the other hand, A **Framework** embodies some abstract design, with more behavior built in. In order to use it you need to insert your behavior into various places in the framework either by sub-classing or by plugging in your own classes. The framework's code then calls your code at these points.

IOC can be implemented in various ways. Most famous is the **Dependency** **Injection**. Other ways are **Observer** pattern, **Template** method etc.

1. **What is Dependency Injection?**

Dependency injection is a software design pattern in which one or more dependencies (or services) are injected, or passed by reference, into a dependent object (or client) and are made part of the client's state.

Suppose a class has two different types of objects. If these objects are instantiated in the class itself then in order to implement a different behavior, there is no way to replace the objects with different objects of same type. For example, if a class has a string that is hard coded for “Hello World”, then that class will print same message every time. So, we say that the class is dependent on the string variable that holds the value. If we can provide a way to alter the string then that is known as Dependency Injection. There are three ways to accomplish the dependency injection: Constructor Injection, Setter Injection and Interface Injection (not recognized by Microsoft). The Interface injection is same as Setter injection except that the Client implements an Interface having just one setter method and that the dependent object is also an interface.

1. **What is Unit of Work?**

During database operations, the objects are frequently manipulated. For example, an object can be altered, deleted or created. Whenever there is a change in the model, the database needs to be changed. This leads to lots of very small database calls thus hitting on the performance. Also, there are transactional changes which keeps the transaction open for whole interaction. This is bad for a business with multiple requests.

A **Unit of Work** (**UoW**) keeps track of everything done in a business transaction that can alter the database. When the activity is done, the Unit of Work figures out everything that needs to be done to alter the database as a result of the activity. The Entity framework already implements the **UoW**. So, it keeps track of altered or deleted data objects. Once commit issued, the entire changes are submitted to the database as a transaction.

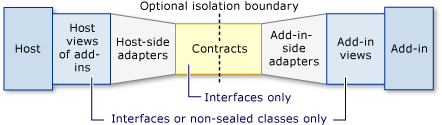
1. **What is Managed Add-in Framework (MAF)?**

**Add-ins** allow to add functionality to an application at a later time. Using **Add-ins**, a hosting application can be created that gains more and more functionality over the time. The **add-ins** can be an in-house development or a third-party program. Internet explorer is one such application for which there are many add-ins available like Google Toolbar or Shockwave Flash player. Microsoft’s **Managed Add-in Framework** (**MAF**) offers a framework for hosting and creating **Add-ins (Plug-ins)**.

Traditionally, **Add-ins** are implemented using *Reflection*, by dynamically loading the assemblies at run-time. However, there are issues which are resolved using MAF. These are: *Discovery*, *Activation*, *Isolation*, *Lifetime* and *Versioning*.

1. **What is the Add-in Model?**

The add-in model consists of a series of segments that make up the add-in pipeline (also known as the communication pipeline), that is responsible for all communication between the add-in and the host. The pipeline is a symmetrical communication model of segments that exchange data between an add-in and its host. Developing these segments between the host and the add-in provides the required layers of abstraction that support versioning and isolation of the add-in.



1. **What is MEF?**

The **Managed Extensibility Framework** or **MEF** is a library for creating lightweight, extensible applications. It allows application developers to discover and use extensions with no configuration required. It also lets extension developers easily encapsulate code and avoid fragile hard dependencies. **MEF** not only allows extensions to be reused within applications, but across applications as well.

A **MEF** component, called a part, declaratively specifies both its dependencies (known as **imports**) and what capabilities (known as **exports**) it makes available. When a part is created, the **MEF** composition engine satisfies its imports with what is available from other parts.

**MEF** aims to solve the runtime extensibility problem. Without **MEF**, any application that want to support a plugin model needs to create its own infrastructure from scratch. Those plugins will often be application-specific and cannot be reused across multiple implementations.

* **MEF** aims to provide a standard way for the host application to expose itself and consume external extensions. Extensions, by their nature, could be reused amongst different applications. However, an extension can still be implemented in a way that is application-specific. Extensions themselves can depend on one another and **MEF** aims to make sure they are wired together in the correct order, sparing the developer from doing it manually.
* **MEF** offers a set of discovery approaches for the application to locate and load available extensions.
* **MEF** allows tagging extensions with additional metadata which aims to facilitate rich querying and filtering.

1. **Explain MVVM.**

<http://blogs.msdn.com/b/dancre/archive/2006/07/23/676300.aspx>

<https://msdn.microsoft.com/en-us/magazine/dd419663.aspx#id0090016>

http://stackoverflow.com/questions/1405739/mvvm-tutorial-from-start-to-finish

# 

# *WCF*

1. **What is WCF?**

To create distributed applications, there are several technologies like *Web Services*, *Remoting*, *Enterprise services*, *MSMQ* etc. If there is a requirement to create a distributed application using windows then there are several tradeoffs among which technology to use and later how feasible it would be to switch to the other technology?

The purpose of WCF is to provide a unified programming model for many of these technologies, enabling us to build applications that are as independent as possible from the underlying mechanism being used to connect services and applications together.

It is actually very difficult, if not impossible, to completely divorce the programmatic structure of an application or service from its communications infrastructure, but WCF lets you come very close to achieving this aim much of the time. Additionally, using WCF enables you to maintain backwards compatibility with many of the preceding technologies. For example, a WCF client application can easily communicate with a Web service that you created by using WSE.

1. **What are the important principles of SOA (Service oriented Architecture)?**

The following guiding principles define the ground rules for development, maintenance, and usage of the SOA:

**Reuse**, **granularity**, **modularity**, **composing**-**ability**, **componentization** and **interoperability**

Standards compliance (both common and industry-specific)

Services identification and categorization, provisioning and delivery, and monitoring and tracking

1. **What are end points, contract, address and bindings?**

Every service must have Address that defines where the service resides, Contract that defines what the service does and a Binding that defines how to communicate with the service. In WCF the relationship between Address, Contract and Binding is called Endpoint.

The Endpoint is the fusion of Address, Contract and Binding. An endpoint contains information that gives the path through which the service is available.

One service can have multiple endpoints, which makes it flexible and interoperable for any application requirements. Each of these endpoints can differ in the address, binding requirements, or contract getting implemented.

WCF provides a unique way to create services independent of the transport being used. The same service can be exposed with two different endpoints. Both of the endpoints have different binding requirements.

For example, Endpoint 1 and Endpoint 2 have the transaction support but run on different transport protocols. In the future, if you need to have another client that has different binding requirements, all you need to do is create another endpoint in the configuration file. This enables you to serve the needs of two or more clients requiring the same business logic encapsulated in the service with different technical capabilities.

1. **What are bindings?**

The binding for a service describes how a client can connect to the service and the format of the data expected by the service. A binding can include the following information:

* **The transport protocol**. This must conform to the requirements of the service address. For example, if you are using IIS to host the service, you should specify the HTTP or HTTPS transport protocol. WCF also has built-in support for the TCP protocol, named-pipes, and message queues.
* **The encoding format of messages**. In many cases, request and response messages will be transmitted in XML format, encoded as ordinary text. However, in some cases you might need to transmit data using a binary encoding, especially if you are transmitting images or handling streams.
* **The security requirements of the service**. You can implement security at the transport level and at the message level, although different transport protocols have their own limitations and requirements.
* **The transactional requirements of the service**. A service typically provides access to one or more resources. Client applications update these resources by sending requests to the service. If a client makes multiple requests of a service that result in multiple updates, it can be important to ensure that all of these updates are made permanent. In the event of a failure, the service should undo all of these updates. This is the definition of a transaction.
* **The reliability of communications with the service**. Clients usually connect to services across a network. Networks are notoriously unreliable and can fail at any time. If a client application is performing a conversation (an ordered exchange of several messages) with a service, information about the reliability of the service is important. For example, the service should try and ensure that it receives all messages sent by the client and receives them in the order that the client sent them. A service can ensure the integrity of conversations by implementing a reliable messaging protocol.

1. **What are the various ways of hosting a WCF service?**

A WCF service can be hosted via one of the following ways.

* Windows application: Simple win-form application
* Console application: Simple console application
* Windows service: WCF service can be controlled by the Service control manager
* IIS: Can be hosted in IIS provided the service exposes at least one HTTP end point
* WAS (Windows Activation Service—comes with IIS 7.0): Removes dependability upon HTTP
* Windows presentation foundation

1. **What are different bindings supported by WCF?**

WCF supports nine types of bindings.

* **BasicHttpBinding**: This binding conforms to the WS-I Basic Profile 1.1. It can use the http and https transport protocols and encodes messages as XML text. Use this binding to maintain compatibility with client applications previously developed to access ASMX-based Web services.
* **WSHttpBinding**: This binding conforms to the WS-\* specifications that support distributed transactions, and secure, reliable sessions. It supports the http and https transport protocols. Messages can be encoded as XML text or by using the *Message Transmission Optimization Mechanism* (MTOM). MTOM is an efficient encoding mechanism for transporting messages that contain binary data.
* **WSDualHttpBinding**: This binding is similar to *WSHttpBinding*, but it is suitable for handling duplex communications. Duplex messaging enables a client and service to perform two-way communication without requiring any form of synchronization (the more common pattern of communication is the request/reply model where a client sends a request and waits for a reply from the service). Using this binding, messages can be encoded as XML Text or by using *MTOM*. However, this binding only supports the http transport protocol, not https.
* **WSFederationBinding**: This binding supports the WS-Federation specification. This specification enables Web services operating in different security realms to agree on a common mechanism for identifying users. A collection of cooperating Web services acting in this way is called a federation. An end-user that successfully connects any member of the federation has effectively logged into all of the members. WS-Federation defines several models for providing federated security, based on the *WS-Trust*, *WS-Security*, and *WS-SecureConversation* specifications.
* **NetTcpBinding**: This binding uses the TCP transport protocol to transmit messages using a binary encoding. It offers higher performance than the bindings based on the http protocols but less interoperability. It supports transactions, reliable sessions, and secure communications. It is ideally suited for use in a local area network, and between computers using the Windows operating system.
* **NetPeerTcpBinding**: This binding supports peer-to-peer communications between applications using the TCP protocol. This binding supports secure communications and reliable, ordered delivery of messages. Messages are transmitted by using a binary encoding.
* **NetNamedPipeBinding**: This binding uses named pipes to implement high-performance communication between processes running on the same computer. This binding supports secure, reliable sessions and transactions. You cannot use this binding to connect to a service across a network.
* **NetMsmqBinding**: This binding uses *Microsoft Message Queue* (*MSMQ*) as the transport to transmit messages between a client applications and service both implemented by using WCF. This binding enables temporal isolation; messages are stored in a message queue, so the client and the service do not both have to be running at the same time. This binding supports secure, reliable sessions and transactions. Messages use a binary encoding.
* **MsmqIntegrationBinding**: This binding enables you to build a WCF application that sends or receives messages from an MSMQ message queue. It is intended for use with existing applications that use MSMQ message queues (the NetMsmqBinding binding uses MSMQ as a transport between a WCF client and service).

1. **What are the main components of WCF?**

The following are the main components of WCF.

1. Contract definitions

* Data contracts and data member
* Service contracts
* Fault contracts
* Message contracts

2. End points

3. Bindings

* BasicHttpBinding
* WSHttpBinding
* WSDualHttpBinding
* WSFederationHttpBinding
* MsmqIntegrationBinding
* NetMsmqBinding
* NetNamedPipeBinding
* NetPeerTcpBinding
* NetTcpBinding

4. Hosting environments

* Windows application
* Console application
* Windows service
* IIS
* WAS (Windows Activation Service), comes with IIS 7.0
* Windows presentation foundation

1. **What is a service class?**

A WCF Service is composed of three parts —

a Service class that implements the service to be provided, a host environment to host the service, and one or more endpoints to which clients will connect. All communications with the WCF service will happen via the endpoints. The endpoints specify a contract that defines which methods of the Service class will be accessible via the endpoint; each endpoint may expose a different set of methods. The endpoints also define a binding that specifies how a client will communicate with the service and the address where the endpoint is hosted.

1. **What are service contract, operation contract and Data Contract?**

**Service contract**: Describe which operations the client can perform on the service.

**Data contract**: Define which data types are passed to and from the service. WCF defines implicit contracts for built-in types such as int and string, but you can easily define explicit opt-in data contracts for custom types.

**Operation contract**: You must explicitly indicate to WCF which methods to expose as part of the WCF contract using the **OperationContract** attribute. You can apply the **OperationContract** attribute on methods, but not on properties, indexers, or events, which are CLR concepts.

1. **What is the difference WCF and Web services?**

WCF "web services" are part of a much broader spectrum of remote communication enabled through WCF. You will get a much higher degree of flexibility and portability doing things in WCF than through traditional ASMX because WCF is designed, from the ground up, to summarize all of the different distributed programming infrastructures offered by Microsoft. An endpoint in WCF can be communicated with just as easily over SOAP/XML as it can over TCP/binary and to change this medium is simply a configuration file change. In theory this reduces the amount of new code needed when porting or changing business needs, targets, etc.

ASMX is older than WCF, and anything ASMX can do so can WCF (and more). Basically you can see WCF as trying to logically group together all the different ways of getting two apps to communicate in the world of Microsoft; ASMX was just one of these many ways and so is now grouped under the WCF umbrella of capabilities.

1. **What are the message exchange pattern in WCF?**

The Message exchange pattern describes how the client and the WCF service exchange messages. WCF supports following three MEPs:

* Request/Response
* One Way
* Duplex

1. **What is Request-Reply pattern in WCF?**

The Request-Reply pattern is the default message exchange pattern. In this, the client sends a message to the service and waits for the response from the service. During this time, the client stops processing until a response is received from the WCF service. Even if the return type of service operation is void, the client still waits for the service to get completed. The Faults and exception gets reported to the client immediately. Except MSMQ, all bindings support Request-Reply MEP.

The Request-Reply pattern is implemented by either specifying nothing in the operation contract or setting the isOneWay to false.

1. **What is One-way pattern in WCF?**

In the One-way pattern, only one message is exchanged between the client and the service. The client makes a call to the service method, but does not wait for a response message. So, the receiver of the message does not send a reply message nor does the sender of the message expects one. To enable the one way MEP, the IsOneWay property is set to true.

As messages are exchanged only in one-way, Faults does not get reported. So, clients are unaware of the server channel faults until a subsequent call is made.

In One-way message exchange pattern, an exception is thrown if the one-way operations declares output parameters, by-reference parameter or return value.

1. **Are One-way calls same as Asynchronous calls?**

No, they are not. When a one way call is received at the service, and if the service is busy serving other requests, then the call gets queued and the client is unblocked and can continue executing while the service processes the operation in the background. One-way calls can still block the client, if the number of messages waiting to be processed, has exceeded the server queue limit. So, One-way calls are not asynchronous calls, they just appear to be so.

1. **Can you explain duplex message exchange pattern in WCF?**

A **Duplex** MEP is a two way message channel, also known as **Callback** operation. **WCF** supports allowing a service to call back to its client. This is useful in two scenarios. First, when the client triggers a long running process in the Service and subsequently requires a notification back from the service about the completion of the process. And second, when the client needs to be able to receive unsolicited messages from the service.

To implement a **duplex** channel, two contracts are needed: ***Service contract*** and ***Callback Contract***. The **Duplex** channel is not supported by all type of Bindings. Only bi-directional capable bindings support callback operations. Because of its connectionless nature, the Http cannot be used for callbacks. So, **BasicHttpBinding** and **WSHttpBinding** do not support **Duplex**. The commonly used **Duplex** supporting bindings are **NetTcpBinding** and **NetNamedPipeBinding**. Although, WCF provides a **WSDualHttpBinding**, for implementing Duplex over Http, however, it is impractical to use it due to various firewalls and other barriers. Also, the **Duplex** MEP are not scalable due to long running session.

1. **What are Async calls in WCF?**

When a client calls a service, usually the client is blocked while the service executes the call, and control returns to the client only when the operation completes its execution and returns. However, there are quite a few cases in which you will want to call operations asynchronously; that is, you’ll want control to return immediately to the client while the service executes the operation in the background and then somehow let the client know that the method has completed execution and provide the client with the results of the invocation. Such an execution mode is called **asynchronous** operation invocation, and the action is known as an **asynchronous** call. **Asynchronous** calls allow you to improve client responsiveness and availability.

The WCF **Asynchronous** calls are strictly a client-side facility and the service is unaware of it. So, any service can support Asynchronous calls, regardless of any underlying bindings. The WCF **Asynchronous** calls are similar to [delegate based **Asynchronous**](#What_is_Asynchronous_Delegates) calls.

1. **Can you explain transactions in WCF?**

A transaction is a logical unit of work consisting of multiple activities that need to all succeed or all fail. This requires support from both client and server side. To add transaction support to a WCF service, you will take the following actions:

* Add transaction support to the service contract. This is **required**.
* Add transaction support to the code that implements the service contract. This is **required**.
* Configure transactions in the implementation code. This is optional.
* Enable transactions on the binding on Server and Client. This is **required**.
* Use the TransactionScope Class to Start a Transaction in Client. This is **required**.

**Adding transaction support to service contract** (specifically operation contract). This done by adding the TransactionFlowOption attribute to the operation contract.

* **NotAllowed**: The operation cannot participate in a transaction. This is the default value for this attribute.
* **Allowed**: The operation will participate in a transaction if the client creates one.
* **Mandatory**: In order to call this operation, the client must create a transaction.

[ServiceContract]

public interface IService1{

[OperationContract]

[**TransactionFlow(TransactionFlowOption.Allowed)]**

void UpdateData();

}

**Add Transaction Support to the Code that Implements the Service Contract**. This is done using the TransactionScopeRequired property of Operation Behavior attribute.

[OperationBehavior(**TransactionScopeRequired = true**)]

public void UpdateData(){

...

}

**Configure Transactions in the Implementation Code**. This is optional. To enable it, Transaction Isolation level property of Service Behavior attribute is used.

[ServiceBehavior (**TransactionIsolationLevel = System.Transactions.IsolationLevel.Serializable**,

  TransactionTimeout="00:00:30")]

public class OrdersService : IOrdersService { … }

**Enable Transactions on the Binding**. To do this, we need to select a Binding which supports Transactions. All of the WCF supplied bindings support transactions, with the exception of the **BasicHTTPBinding** and **NetPeerTcpBinding** bindings. This has to be done at both Server and Client side.

</ServiceBehavior>

</behaviors>

  <Bindings>

    <WSHttpBinding>

      <binding name="WSHttpBinding"

**transactionFlow ="true"** />

    </WSHttpBinding>

  </Bindings>

</system.serviceModel>

Use the **TransactionScope Class to Start a Transaction in Client**. To do this, we need to add reference to System.Transactions assembly. Then we need to use the TransactionScope object to place the transactional code.

using (TransactionScope ts = new TransactionScope(TransactionScopeOption.RequiresNew))

{

try

{

ServiceReference1.Service1Client obj = new ServiceReference1.Service1Client ();

obj.UpdateData ();

ts.Complete ();

}

catch (Exception ex)

{

ts.Dispose ();

}

}

<https://msdn.microsoft.com/en-us/library/ff384250.aspx>,

<http://www.codeproject.com/Articles/38793/Steps-to-Enable-Transactions-in-WCF>,

<http://wcftutorial.net/How_to_create_WCF_with_Transaction.aspx>

<http://www.c-sharpcorner.com/UploadFile/e70b61/transaction-in-wcf/>

1. **What different transaction isolation levels provided in WCF?**

In general, the more isolated the transaction, the more consistent their results are. The following are the isolation levels supported in WCF.

* **Serializable**: This is the default level of isolation used by the .NET Framework. It is also the most restrictive level. Other transactions cannot change any data read by this transaction until the transaction completes. The transaction can safely reread its data, knowing that other transactions have not changed it. This provides the most protection for the transaction, but can cause performance problems due to excessive locking of database records. This is a good choice for short transactions, but if your transactions are longer, you may want to use another level of isolation.
* **ReadUncommitted**: Other transactions can change data read by the transaction. They can also read data changed by the transaction before it commits. This may cause the other transactions to read incomplete or inaccurate data.
* **ReadCommitted**: Other transactions can change data read by the transaction. However, they cannot read data changed by the transaction until it commits. This is the default isolation level in SQL Server.
* **RepeatableRead**: Other transactions cannot update any data read by this transaction. However, they can insert rows. If the transaction needs to reread rows, there is a risk of additional data being present. This may cause unexpected results in some cases, but will improve performance.
* **Unspecified**: The service does not know ahead of time what level of isolation the transaction will use. This is the default value for the TransactionIsolationLevel property in a WCF service.
* **Chaos**
* **Snapshot**

1. **What are channels in WCF?**

A service can respond to requests from multiple client applications simultaneously. To achieve this feat, the application hosting the service must be able to accept multiple incoming requests and direct service responses back to the appropriate client. Additionally, the host application must ensure that messages being sent between the client and service conform to the security, reliability, and transactional requirements of the binding being used. Fortunately, you don’t have to write this functionality yourself. The WCF runtime environment provides a collection of channel objects that can perform this processing for you.

A channel is responsible for handling one aspect of message processing, as specified by the bindings of a service. For example, a transport channel manages communications by using a specific transport protocol, and a transaction channel controls the transactional integrity of a conversation.

The WCF runtime provides built-in channels for each of the supported transport protocols. The WCF runtime also provides channels that handle the different ways that WCF can encode data, manage security, implement reliability, and perform transactions.

The WCF runtime composes channels into a channel stack. All messages passing between the client and the service go through each channel in the channel stack. Each channel in the channel stack transforms the message in some way, and the output from one channel is passed as input to the next. The channel stack operates in two directions–messages received from clients across the network proceed up the channel stack to the service, and response messages sent back from the service traverse the channel stack in the opposite direction back to the network and then to the client. If a channel cannot process a message, it reports an error, an error message is sent back to the client, and the message is not processed any further.

There is an order to the channels in the channel stack. A transport channel always resides at the bottom of the stack and is the first channel to receive data from the network. On top of the transport channel will be an encoding channel. These two channels are mandatory. The remaining channels in a stack are optional.

1. **How a client request is processed in WCF?**

When you start a service running, the WCF runtime uses the endpoint information specified as part of the service configuration and creates a listener object for each address specified for the service. When an incoming request is received, the WCF runtime constructs a channel stack by using the bindings specified for the address and routes the incoming data from the client through the stack. If a message successfully traverses all the channels in the channel stack, the transformed request is passed to an instance of the service for processing.

The WCF runtime creates an InstanceContext object to control the interaction between the channel stack and the service instance. You can modify the way in which the WCF runtime instantiates the service through the InstanceContext object by specifying the [**ServiceBehavior**] attribute of the class implementing the service contract.

A WCF client application can communicate with a WCF service by using a proxy class. You can generate this proxy class by using Visual Studio or by using the svcutil utility from the command line. This proxy class implements a channel stack on the client side. You configure this channel stack in the same way that you do for a service, by using bindings. All responses received from a service pass through the channels in this stack. To communicate successfully, the client and the service should use an equivalent channel stack containing a compatible set of bindings.

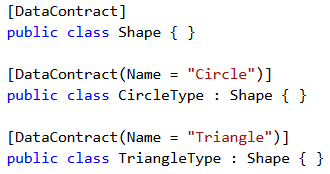
1. **What is KnownType attribute?**

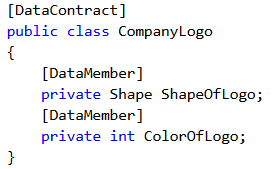
The **KnownTypeAttribute** class allows you to specify, in advance, the types that should be included for consideration during deserialization. Normally, when passing parameters and return values between a client and a service, both endpoints share all of the data contracts of the data to be transmitted. However, this is not the case in the following circumstances:

* The sent data contract is derived from the expected data contract. In that case, the transmitted data does not have the same data contract as expected by the receiving endpoint.
* The declared type for the information to be transmitted is an **interface**, as opposed to a **class**, **structure**, or **enumeration**. Therefore, it cannot be known in advance which type that implements the interface is actually sent and therefore, the receiving endpoint cannot determine in advance the data contract for the transmitted data.
* The declared type for the information to be transmitted is **Object**. Because every type inherits from [**Object**](https://msdn.microsoft.com/en-us/library/system.object%28v=vs.110%29.aspx), and it cannot be known in advance which type is actually sent, the receiving endpoint cannot determine in advance the data contract for the transmitted data. This is a special case of the first item: Every data contract derives from the default, a blank data contract that is generated for [**Object**](https://msdn.microsoft.com/en-us/library/system.object%28v=vs.110%29.aspx).
* Some types, which include .NET Framework types, have members that are in one of the preceding three categories. For example, **Hashtable** uses **Object** to store the actual objects in the hash table. When serializing these types, the receiving side cannot determine in advance the data contract for these members.

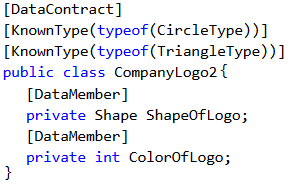
The **KnownType** attribute is necessary when you are serializing non-concrete types such as interfaces or base classes. The WCF serializer must know about all possible implementations of the interface or inherited class. Any implementations that it doesn't know about will cause a serialization exception.

One way to let the deserialization engine know about a type is by using the **KnownTypeAttribute**. The attribute cannot be applied to individual data members, only to whole data contract types. The attribute is applied to an outer type that can be a class or a structure. In its most basic usage, applying the attribute specifies a type as a "known type." This causes the known type to be a part of the set of known types whenever an object of the outer type or any object referred to through its members is being deserialized. More than one **KnownTypeAttribute** attribute can be applied to the same type.





// The Wrong Way



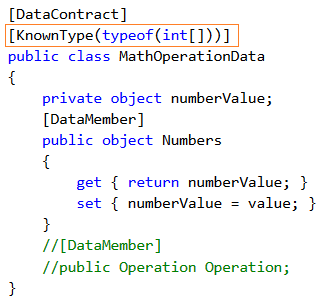
// The Right Way

In the above code, **Shape** is inherited by **CircleType** and **TriangleType**. The class **CompanyLogo** can be serialized, but it cannot be deserialized, if the data-member **ShapeOfLogo** is set to either **CircleType** or **TriangleType** object. The class **companyLogo2** is the correct way. Whenever the outer type **CompanyLogo2** is being deserialized, the deserialization engine knows about **CircleType** and **TriangleType** and, therefore, is able to find matching types for the "Circle" and "Triangle" data contracts.

When the **KnownType** attribute is applied to a Base-type class, then those **KnownTypes** are automatically associated to the Derived classes.

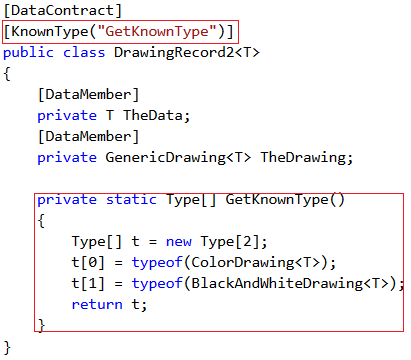
1. **How to make WCF accept a single value, Array and List in same Data-Member?**

Suppose there is a requirement which requires a data member exposed in data contract to accept an array, a list or just a single value. This can be done using the KnownType attribute. The example below will expose the Number data-member to accept an integer, an array of integers or a list of integers.



1. **How does KnownType works for Generic type data-contracts?**

Sometimes it becomes necessary to add a generic type as a known type. However, an open generic class cannot be passed as a parameter to the **KnownType** attribute. In this case, the **KnownType** attribute provides a method parameter. In this, we can pass the name of the method which, when implemented, will return the list of types to add to the known types collection.



<http://www.codeproject.com/Tips/601224/What-is-KnownType-Attribute-and>

<http://msdn.microsoft.com/en-us/library/ms730167(v=vs.110).aspx>

1. **How are sessions maintained in WCF?**

In WCF applications, a session correlates a group of messages into a conversation. The correlation can be anything from the messages based on a shared network connection, to the messages based on a shared tag. WCF sessions are different than ASP.Net ones in terms of behaviors and controlling. WCF sessions have the following main conceptual features:

* They are explicitly initiated and terminated by the calling application (the WCF client). Server initiated in ASP.Net.
* Messages delivered during a session are processed in the order in which they are received. Unordered in ASP.Net.
* Sessions correlate a group of messages into a conversation. Different types of correlation are possible.
* There is no general data store associated with a WCF session. Storage mechanism is available in ASP.Net.

Sessions are supported only by Session-Based bindings. Service contracts specify that they require, permit, or refuse session-based bindings by setting the **ServiceContractAttribute.SessionMode** property on the service contract interface (or class) to one of the **System.ServiceModel.SessionMode** enumeration values. These are:

* **Allowed**. It specifies that the contract supports sessions if the incoming binding supports them. Default.
* **NotAllowed**. It specifies that the contract never supports bindings that initiate sessions. And,
* **Required**. It specifies that the contract requires a sessionful binding. An exception is thrown if the binding is not configured to support session.

<https://msdn.microsoft.com/en-us/library/ms733040%28v=vs.110%29.aspx>

1. **How are sessions initiated and Terminated in WCF?**

If a service contract requires or supports sessions, one or more contract operations can be marked as initiating or terminating a session by setting the **IsInitiating** and **IsTerminating** properties (Demarcation of operations).

Initiating operations are those that must be called as the first operation of a new session. Non-initiating operations can be called only after at least one initiating operation has been called. You can therefore create a kind of session constructor for your service by declaring initiating operations designed to take input from clients appropriate to the beginning of the service instance. (The state is associated with the session, however, and not the service object.)

Terminating operations, conversely, are those that must be called as the last message in an existing session. In the default case, WCF recycles the service object and its context after the session with which the service was associated is closed. You can, therefore, create a kind of destructor by declaring terminating operations designed to perform a function appropriate to the end of the service instance.

Services do not start sessions with clients. In WCF client applications, a direct relationship exists between the lifetime of the session-based channel and the lifetime of the session itself. As such, clients create new sessions by creating new session-based channels and tear down existing sessions by closing session-based channels gracefully. A client starts a session with a service endpoint by calling one of the following:

* **ICommunicationObject.Open** on the channel returned by a call to **ChannelFactory<TChannel>.CreateChannel**.
* **ClientBase<TChannel>.Open** on the WCF client object generated by the ServiceModel Metadata Utility Tool (**Svcutil.exe**).
* An initiating operation on either type of WCF client object (by default, all operations are initiating). When the first operation is called, the WCF client object automatically opens the channel and initiates a session.

Typically a client ends a session with a service endpoint by calling one of the following:

* **ICommunicationObject.Close** on the channel returned by a call to **ChannelFactory<TChannel>.CreateChannel.**
* **ClientBase<TChannel>.Close** on the WCF client object generated by **Svcutil.exe**.
* A terminating operation on either type of WCF client object (by default, no operations are terminating; the contract must explicitly specify a terminating operation).

https://msdn.microsoft.com/en-us/library/ms751429%28v=vs.110%29.aspx

1. **What is Instancing and what are instantiation modes?**

In WCF, a request from client is handled by a service instance. Instancing is a process which determines the Service instantiation. The Instance management of WCF decides by which service instance, a client request will be handled. The determination of the instancing mode is done on the Server side. This keeps the implementation details hidden from the caller. The Instancing mode is configured within the service behavior. So once set, the instancing mode will be applicable to all the endpoints of the service. It can also be applied directly to the service implementation class.

There are three Instantiation mode, which are set using ***InstanceContextMode*** parameter of service behavior attribute. These modes are: ***PerCall,*** ***PerSession*** (Default), and ***Singleton***.

<https://msdn.microsoft.com/en-us/magazine/cc163590.aspx#S6>, <http://stackoverflow.com/questions/8247509/how-does-wcf-instance-work>,

https://www.youtube.com/watch?v=7mp5ylCqAmg&list=PL6n9fhu94yhVxEyaRMaMN\_-qnDdNVGsL1&index=41,

1. **What is PerCall Instantiation mode?**

In **PerCall** instantiation mode, every single request gets its own copy of service implementation object (*service-object*). When the request arrives at the service host, the host creates an instance of Service’s implementation class. This class is then called to process the client request. After the request is complete and the response is returned to the client, the *service-object* is disposed of. It is a Default behavior, so developers do not need to worry about concurrency.

In **PerCall**, the *service-object* is instantiated as soon as it is needed and it is disposed as soon as the request is completed. If the object holds on to a scarce resource, the lifetime of the service object has been reduced to minimize the performance impact of holding that resource.

In **PerCall**, no state can exist between calls. However, if need be, then state can be persisted by the service with the use of database. In subsequent requests, the previously saved state can be restored and used. However, in that case, there has to be a way to distinguish the state of different clients. This can be done by using a business level value like customer id, batch id etc., or a simple meaningless value like Guid. The state can also be passed as parameter to the service to mock session.

The **PerCall** mode is best used when individual operations are short and the operations does not spawn any background threads that continue processing after the request is complete.

1. **What is PerSession Instantiation mode?**

In **PerSession** instantiation mode, WCF maintains a private session between a client and a particular instance of the service’s implementation object.

In **PerSession** mode, each client, upon the first request to the service, gets an instance of the *service-object*. This instance is dedicated to processing the requests that come from that client. Any subsequent calls are considered to be the part of the same session, and calls are processed by the same instance of the implementation object.

The **PerSession** mode has two components, **Contractual** and **Behavioral**. In **Contractual** piece, client is made aware that a session is required. This is necessary because to maintain the session, the client must include an identifier to locate the appropriate *service-object*. The contractual piece is implemented using the **SessionMode** property of **ServiceContract** attribute. For **PerSession** mode, the **SessionMode** property is set to **SessionMode.required**. The **Behavioral** piece is implemented in the **ServiceBehavior** attribute, where the service contract is implemented. In ServiceBehavior attribute, the instance context mode is set to **PerSession**.

In **PerSession** mode, the association is between the Client-Proxy and the Service. When a Proxy is created, an identifier is generated for that Proxy. This identifier is then used by the service host to direct any requests to the appropriate instance. So, if a client creates more than one proxy then there will be separate session for each proxy and they cannot be combined.

In **PerSession** mode, the service instance is disposed automatically after 10 minutes of no activity. This is configurable. This minimizes the performance impact, in case session involves holding up of scarce resources, or the proxy is unable to close the session properly.

1. **What is Singleton Instantiation mode?**

In **Singleton** mode, only one instance of *service-object* is created. This instance handles every request that arrives at the service. The instance lives forever and is disposed of only when the host process is shut down.

In **Singleton**, the sessions are not a compulsory requirement. There is no restriction on the ability of the binding to support transport-level sessions. If the service contract has a session then the client must provide a session. Further, if a session is associated with the request, that session will never expire. The session identifier is maintained within the client proxy until the proxy is destroyed.

One of the feature **Singleton** mode offers is the ability to initialize the implementation instance through the constructor. In other modes, the instance object is created behind the scenes at a time determined by the host process. However for singletons, there is an option to create the singleton instance and pass it into the host process. This is done to perform the initialization process outside the scope of first request. Otherwise the first request will have to pay the price. To achieve this, the **ServiceInstance** is created prior to the creation of Host object and then, the **ServiceInstance** is passed to the Host constructor.

1. **What are concurrency modes?**

The **Concurrency** in WCF service occurs when more than one call is made simultaneously to a given service.

When a request arrives at the service, the service model dispatches the message on its on thread, which is taken from a pool of threads. If multiple requests arrive at the same time, additional threads are grabbed from the same pool and designated to process the request. Each request is associated with a *service-object*, which is called by the Service Model to process the request and build the response. Within **WCF**, the concurrency arise based on how the *service-object* is created and shared between the individual requests.Concurrency is used to improve the Throughput. Throughput is the amount of work done in a specified amount of time.

**WCF** provides three possible sharing modes for the *service-object*:

* **Single**: Each thread that is handling a request can access the *service-object*, but only one thread at a time can access the service object.
* **Multiple**: The *service-object* satisfies the multiple requests, potentially simultaneously.
* **Re-Entrant**: Only one thread can access the *service-object* at a time, however, it is possible for that thread to leave the object and re-enter it at some point in the future.

1. **What is Single Concurrency mode?**

The Single concurrency mode provides a safest environment for concurrency. Before the service begins to process a request, a lock is acquired on the service object. The lock is maintained until the request is complete. The subsequent requests are queued up until the service object becomes available. This eliminates any concurrency issues.

Client request Concurrency of a service depends on 3 factors: Instance Context Mode, Concurrency Mode and Session support by the Binding. For example, if the Instance Context mode is Singleton and Concurrency mode is Single, then the Service cannot handle concurrent calls, irrespective of session support. Below table gives a comparative study:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Deciding Factors** | | | **Results** | |
| **Instance Context Mode** | **Concurrency Mode** | **Binding Supports session** | **Is Concurrent?** | **Throughput Impact** |
| PerCall | Single | No | Yes | Positive |
| PerCall | Single | Yes | No | Negative |
| PerSession | Single | No | Yes | Positive |
| PerSession | Single | Yes | Yes - Between different client requests  No – For the requests from same client | Positive – Between different Clients  Negative – For same client requests |
| Singleton | Single | No | No | Negative – Between clients, and also for the requests from the same client. |
| Singleton | Single | Yes | No |

1. **What is Multiple Concurrency mode?**

The Multiple concurrency mode increases the throughput unlike Single concurrency mode. In Multiple concurrency mode, a lock is no longer acquired on the service object before a request is processed. The Service object (s) can process multiple request simultaneously. However, special care need to be taken to synchronize the shared state and resources.

When a service is set to Multiple Concurrency mode, the instance context mode or underlying binding support session, does not have any effect on the overall throughput of the WCF service. Below table describe the same:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Deciding Factors** | | | **Results** | |
| **Instance Context Mode** | **Concurrency Mode** | **Binding Supports session** | **Is Concurrent?** | **Throughput Impact** |
| PerCall | Multiple | No | Yes | Positive |
| PerCall | Multiple | Yes | Yes | Positive |
| PerSession | Multiple | No | Yes | Positive |
| PerSession | Multiple | Yes | Yes | Positive |
| Singleton | Multiple | No | Yes | Positive |
| Singleton | Multiple | Yes | Yes | Positive |

1. **What is Re-Entrant Concurrency mode?**

The Reentrant concurrency mode is just like Single concurrency mode, in terms of acquiring and locking the Service instance. However, the Reentrant mode solves the deadlock problem that is introduced in Single concurrency mode.

Suppose, a service is configured for Single Concurrency mode. Then a client request gets the lock and acquires the service instance. This prevents any other requests to get processed. Now, if the service makes a call back to the client or any other service, it would expect a request/response back from the destination. However, the request that is coming back from the destination, won’t be entertained by the Service instance because it is locked by a request. And neither the service instance can complete the request because that request is waiting for the call back to be completed. So a deadlock.

The Reentrant mode is one of the two ways to solve the deadlock situation. In this, the concurrency mode of the service is set to Reentrant. The other way to set the concurrency mode to Single and set the operation contract to one-way, instead of two ways (isOneWay = true).

1. **What is Service Throttling?**
2. **What happens when there is a change in DataContract?**

Whenever there is a change in the DataContract.

1. **What is ExtensionDataObject?**

# 

# *Security*

1. **How CAS helps in ASP.Net Security?**

The *ASP.NET* application runs in the *CLR*, so they are constrained by the permission granted to them by *Code Access Security* (*CAS*). This helps in protecting the application and the server which hosts the web application. If there’s a server that hosts several web application by several companies, *CAS* allows each company to have freedom to manage its own application, while ensuring that the code it uploads cannot effect the host server or the applications of different companies running in it. Even if hackers manage to gain control of an application, their actions will be constrained by the permissions assigned to the application by *CAS*, limiting the damage they can do.

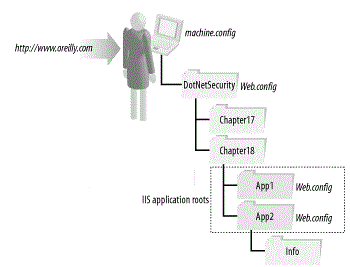
1. **How are configuration files structured in ASP.net?**

ASP.NET uses XML-based configuration files as a simple and convenient mechanism through which to configure elements of application operation. This includes important security settings, such as the user authentication mechanism to use and the code-access security policy to apply to an ASP.NET application. We discuss all of these security settings later this chapter, but first it is important that you understand how ASP.NET configuration files work.

ASP.NET implements a hierarchically configuration file structure. As with all .NET applications, the main source of configuration information is the machine.config file, which is located in the CONFIG folder of the .NET Framework installation directory. The machine.config file provides a central location in which you can implement configuration settings that apply to all ASP.NET applications running on the machine.

Each folder in an ASP.NET application's virtual path can contain a configuration file named Web.config, which provides configuration settings for the application resources located in that folder. Child folders inherit the configuration of their parent folder, meaning that configuration settings cascade down through an application's virtual folder hierarchy. However, child folders can also contain their own Web.config; the settings contained in this file override those configured higher up in the folder hierarchy.

In the corresponding figure, the machine.config file resides at the top of the configuration hierarchy, and its settings apply to all folders in the hierarchy below it unless they are overridden. The top-level DotNetSecurity folder contains a Web.config file, which will apply to all resources in all of the folders below it; this includes the Chapter17 and Chapter18 folders and all of their child folders. Both of the App1 and App2 folders have their own Web.config files. The Info folder contains no Web.config file and is subject to the configuration settings specified in the following configuration files:



* machine.config
* DotNetSecurity/Web.config
* DotNetSecurity/Chapter18/App2/Web.config

1. **What is ASP.Net worker process identity?**

ASP.NET uses a pool of worker processes to handle ASP.NET application requests. By default, each worker process executes in the context of a special Windows account named ASPNET. The ASPNET account is created when the .NET Framework and ASP.NET are installed, and is assigned a limited set of Windows privileges and permissions. Running ASP.NET applications with reduced permissions ensures that the applications cannot access system resources that they should not and reduces the risks posed by an application whose security has been compromised. (We can configure ASP.NET applications to run in the context of the authenticated user making the request. This is achieved using Impersonation.)

You can change the account under which ASP.NET worker processes run by configuring the *userName* and *password* attributes of the <processModel> element, which is located in the machine.config file. This change affects all ASP.NET applications running on the machine; it is not possible to change the process identity for select applications.

ASP.NET provides two preconfigured values that you can assign to the userName attribute to control the worker process identity:

* Machine: This is the default value and runs the ASP.NET application as the ASPNET account.
* System: This value causes the ASP.NET worker process to run as the same account under which IIS is running; by default this will be the built-in Windows System account.

When using either of these settings, the value of the password attribute should be set to AutoGenerate to avoid the necessity of specifying the true password. For example, the following extract from the machine.config file configures the ASP.NET worker process to run as the same account used by IIS:

<configuration>

<processModel userName="System" password="AutoGenerate"/>

</configuration>

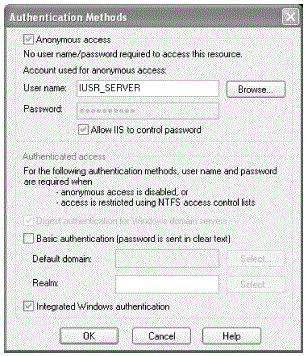
Alternately, you can specify the name of an account under which the worker process should run. The userName attribute specifies the account name, and the password attribute contains the account's password in clear text. The following extract from the machine.config file configures the ASP.NET application process to run as the account "SomeUser," which has the password "secret":

<configuration>

<processModel userName="SomeUser" password="secret"/>

</configuration>

The obvious downside of this approach is the presence of the plaintext username and password in the configuration file. Although by default IIS and ASP.NET are configured not to serve .config files in response to client requests, it is never good practice to list such information in plaintext.



1. **What are IIS Authentication Modes?**

Before a client request reaches your ASP.NET application, it must pass through IIS authentication. The result of IIS authentication provides the input that drives the .NET authentication mechanism. For the Windows authentication of ASP.NET, IIS authentication provides your application with the information about the authenticated Windows user. However, for the None, Forms, and Passport authentication modes, it is most likely that you will want to turn off IIS authentication by enabling anonymous access to IIS.

*Unless you enable* *impersonation, ASP.NET authentication has no effect on the Windows user context under which your ASP.NET application executes*. That means, the application executes in the context of the ASP.NET worker process identity. ASP.NET authentication controls the identity and principal assigned to the ASP.NET application.

IIS 7 supports both ***challenge-based*** and ***login redirection-based*** authentication methods. A challenge-based authentication method, for example, Integrated Windows authentication, requires a client to respond correctly to a server-initiated challenge. A login redirection-based authentication method, for example, Forms authentication, relies on redirection to a login page to determine the identity of the user. You cannot use both a challenge-based authentication method and a login redirection-based authentication method at the same time. More in…http://technet.microsoft.com/en-us/library/cc733010(v=ws.10).aspx

Below are the IIS authentication modes:

* Anonymous Authentication,
* Basic Authentication,
* Digest Authentication,
* ASP.NET Impersonation Authentication
* Forms Authentication,
* Windows Authentication and,
* Client Certificate Mapping Authentication

1. **What is Anonymous Authentication?**

Anonymous authentication allows any user to access any public content without providing a user name and password challenge to the client browser. When you enable Anonymous authentication, you can change the account that IIS uses to access your sites and applications. By default, IIS 7 uses IUSR as the user name for anonymous access. This user name is created when you install IIS 7.

**Pros**

* Offers the best performance because Anonymous authentication imposes no appreciable overhead.
* Does not require management of individual user accounts.
* If IIS does not control the password, can access network resources.

**Cons**

* Does not authenticate clients on an individual basis.
* If IIS does not control the password, account must have local logon ability.

1. **What is Basic Authentication?**

Basic authentication requires that users provide a valid user name and password to access content. This authentication method does not require a specific browser, and all major browsers support it. Basic authentication also works across firewalls and proxy servers. For these reasons, it is a good choice when you want to restrict access to some, but not all, content on a server. However, the disadvantage of Basic authentication is that it transmits unencrypted base64-encoded passwords across the network. You should use Basic authentication only when you know that the connection between the client and the server is secure.

**Pros**

* Because it is part of the HTTP 1.0 specification, Basic is the most widely supported user authentication scheme.
* Can authenticate through a proxy server.
* Makes it possible to track individual users.
* Can access network resources, if user account has local logon rights on the Web server.
* Can be used in conjunction with Kerberos, enabling delegation of security credentials.

**Cons**

* Is inherently insecure unless using SSL/TLS, which impacts performance.
* Requires the creation of individual Windows accounts for each user.

1. **What is Digest Authentication?**

Digest authentication addresses the primary weaknesses of basic authentication: sending passwords in plain text. Digest authentication is a challenge/response mechanism, which sends a digest (also known as a hash) instead of a password over the network. A digest is a fixed-size result obtained by applying a mathematical function, called a hash function or digest algorithm, to an arbitrary amount of data.

When a client attempts to access a resource requiring Digest authentication, IIS send a challenge to the client to create a digest and send it to the server. The client concatenates the password with data known to both the server and the client. The client then applies a digest algorithm (specified by the server) to the combined data. The client sends the resulting digest to the server as the response to the challenge. The server uses the same process as the client to create a digest using a copy of the client's password it obtains from Active Directory, where the password is stored using reversible encryption. If the digest created by the server matches the digest created by the client, IIS authenticates the client. IIS uses a sub-authentication DLL (iissuba.dll) to authenticate the user, resulting in a network logon. By itself, Digest authentication is only a slight improvement over Basic authentication. In the absence of SSL/TLS, an attacker could record communication between the client and server. Using this information, the attacker can then use that information to replay the transaction.

**Pros**

* Sends a digest over the network instead of a password.
* Works with proxy servers and firewalls.
* Does not require SSL/TLS for the sake of password protection.

**Cons**

* Cannot delegate security credentials.
* Is only supported by Internet Explorer 5.0 and later.
* Is subject to replay attacks unless you use SSL/TLS.
* Requires storing of passwords in clear-text using reversible encryption.
* Requires the creation of domain accounts for each user in Active Directory.

1. **What is Client Certificate Mapping Authentication?**

A certificate is a digitally signed statement that contains information about an entity and the entity's public key, thus binding these two pieces of information together. A trusted organization (or entity) called a Certification Authority (CA) issues a certificate after the CA verifies that the entity is who it says it is. Certificates can contain different types of data. For example, an X.509 certificate includes the format of the certificate, the serial number of the certificate, the algorithm used to sign the certificate, the name of the CA that issued the certificate, the name and public key of the entity requesting the certificate, and the CA's signature. X.509 client certificates simplify authentication for larger user bases because they do not rely on a centralized account database. You can verify a certificate simply by examining the certificate.

Operating systems such as Windows still require the notion of a user account. Certificate mapping makes it possible for administrators to associate a single certificate (one-to-one mapping), or multiple certificates (many-to-one), to a user account. Many-to-one mapping uses rules to define certificate criteria for mapping.

IIS uses SSL/TLS to authenticate a server and provide an encrypted HTTP session. IIS can also use SSL/TLS to authenticate the client by requiring the client to provide a certificate. When requesting a client certificate, the server provides the client with a list of CAs that the server trusts. This list is derived from the server's Certificate Trust List (CTL). If the client possesses a certificate issued by a CA from the CTL, it sends a copy of that certificate to the server for verification. If the certificate is valid, IIS authenticates the user that maps to the provided certificate.

**Pros**

* Includes strong authentication scheme.
* Provides two-way authentication of server and client.
* Can access network resources.

**Cons**

* Cannot delegate security credentials.
* Does not work with all browsers.
* Requires SSL/TLS.
* Is cumbersome to configure; however, many-to-one can be easier than one-to-one.

1. **What is Forms Authentication?**

Forms authentication uses client-side redirection to forward unauthenticated users to an HTML form where they can enter their credentials, which are usually a user name and password. After the credentials are validated, users are redirected to the page they originally requested.

1. **What is ASP.NET Impersonation Authentication?**

Use ASP.NET impersonation when you want to run your ASP.NET application under a security context different from the default security context for ASP.NET application.

If you enable impersonation for an ASP.NET application, that application can run in one of two different contexts: either as the user authenticated by IIS 7 or as an arbitrary account that you set up. For example, if you were using Anonymous authentication and chose to run the ASP.NET application as the authenticated user, the application would run under an account set up for anonymous users (typically, IUSR). Likewise, if you chose to run the application under an arbitrary account, it would run under whatever security context was set up for that account.

By default, ASP.NET impersonation is disabled. If you enable impersonation, your ASP.NET application runs under the security context of the user authenticated by IIS 7.

1. **What is File Authorization?**

File authorization is enforced automatically when you enable ASP.NET Windows authentication, and cannot be disabled or configured. After IIS authenticates the user, but before ASP.NET hands the request to your application for processing, ASP.NET checks to see whether the authenticated user has the necessary NTFS permissions to access the requested resource. If she has permission, ASP.NET will hand the request to your application for processing; otherwise, it will refuse the request.

Understand that the NTFS checks performed during File authorization are in addition to the normal enforcement of NTFS file permissions performed during application execution. If your ASP.NET application accesses files during execution, they are accessed in the context of the ASP.NET worker process identity, or the identity of the authenticated Windows user if you have enabled impersonation. If the active identity does not have the necessary NTFS permissions to access the file, an error will occur, which your application must handle.

1. **What is the Scope of WCF security?**

WCF security is divided into three functional areas: transfer security, access control, and auditing. The following sections briefly discuss these areas and provide links for more information. http://msdn.microsoft.com/en-us/library/ms735093(v=vs.110).aspx

* *Transfer* *Security* – Transport and Message Security modes
* *Access* *Control* – Authorization, Impersonation etc. and,
* *Auditing*

1. **What is Transfer Security?**

Transfer security encompasses three major security functions: *integrity*, *confidentiality*, and *authentication*. *Integrity* is the ability to detect whether a message has been tampered with. *Confidentiality* is the ability to keep a message unreadable by anyone other than the intended recipient; this is achieved through cryptography. *Authentication* is the ability to verify a claimed identity. Together, these three functions help to ensure that messages securely arrive from one point to another.

Two main mechanisms are used to implement transfer security in WCF are **transport** **security** **mode** and **message security mode**.

1. **What is Transport Security Mode?**

When using *transport* *security*, the user credentials and claims are passed using the transport layer. In other words, user credentials are transport-dependent, which allows fewer authentication options compared to message security. Each transport protocol (TCP, IPC, MSMQ, or HTTP) has its own mechanism for passing credentials and handling message protection. The most common approach for this is to use *Secure* *Sockets* *Layer* (*SSL*) for encrypting and signing the contents of the packets sent over *Secure* *HTTP* (*HTTPS*). http://msdn.microsoft.com/en-us/library/ff650862.aspx

Transport security is used to provide point-to-point security between the two endpoints (service and client). If there are intermediary systems between the client and the service, each intermediate point must forward the message over a new SSL connection.

Use transport security for the following scenarios:

* You are sending a message directly from your application to a WCF service and the message will not be routed through intermediate systems.
* You have both the service and the client in an intranet.

Using transport security has the following advantages:

* It provides interoperability, meaning that communicating parties do not need to understand the WS-Security specification.
* It may result in better performance.
* Hardware accelerators can be used to further improve performance.

Using transport security has the following disadvantages:

* Because security is applied on a point-to-point basis, there is no provision for multiple hops or routing through intermediate application nodes.
* It supports a limited set of credentials and claims compared to message security.
* It is transport-dependent upon the underlying platform, transport mechanism, and security service provider such as NTLM or Kerberos.

1. **What is Message Security Mode?**

When using message security, the user credentials and claims are encapsulated in every message using the WS-Security specification to secure messages. This option gives the most flexibility from an authentication perspective. You can use any type of security credentials you want, largely independent of transport, as long as both the client and the service agree.

Use message security for the following scenarios:

* You are sending a message to a WCF service, and the message is likely to be forwarded to other WCF services or may be routed through intermediate systems.
* Your WCF clients are accessing the WCF service over the Internet, it is possible that other intermediate systems may be used in between, and security is your top consideration.

Using message security has following advantages:

* It provides end-to-end security. Because message security directly encrypts and signs the message, having intermediaries does not break the security.
* It allows partial or selective message encryption and signing, thus improving overall application performance.
* Message security is transport-independent and can be used with any transport protocol.
* It supports a wide set of credentials and claims, including issue token, which enables federated security.

Using message security has following disadvantages:

* This option may reduce performance compared to transport security because each individual message is encrypted and signed.
* It does not support interoperability with older ASP.NET Web Services (ASMX) clients because it requires both the client and service to support WS-Security specifications.

1. **What are the authentication options available with Transport security?**

The follow authentication options are available when using transport security:

* **None**. When using this option, the WCF service does not authenticate the callers. This is not the recommended option from a security perspective—avoid using this option wherever possible.
* **Basic**. This option is available with the HTTP protocol only. The client is authenticated by using the username and password against the *Active* *Directory* service. The client credentials are transported by using a Base64 encode string, which is very similar to a clear string and therefore not the most secure option. The service is authenticated by the Secure Sockets Layer (SSL) certificate used for secure communication.
* **NTLM**. This option is available with the HTTP protocol only. The client is authenticated by using a challenge-response scheme against Windows accounts. NTLM authentication is well suited for a workgroup environment and is more secure than Basic authentication. The service is authenticated by using an SSL certificate.
* **Windows**. When using this option, the WCF service uses Kerberos authentication when in a domain, or NTLM authentication when deployed in a workgroup environment. This option uses a Windows token presented by the caller to authenticate against the Active Directory. This is the most secure option compared to Basic or NTLM authentication. The service is authenticated by using the Windows credentials of the process identity, or an SSL certificate if you are using the HTTP protocol.
* **Certificate**. When using this option, the caller presents an X.509 client certificate that the WCF service validates by trusting the certificate (peer trust) or trusting the issuer of the certificate (chain trust). This option should be used when Windows authentication is not possible, as in the case of business-to-business (B2B) scenarios. The service is authenticated with the service certificate, or by using an SSL certificate if you are using the HTTP protocol.

1. **What are the authentication options available with Message security?**

The following authentication options are available when using message security:

* **None**. When using this option, the WCF service does not authenticate the callers. This is not the recommended option from a security perspective—avoid using this option wherever possible.
* **Windows**. When using this option, the WCF service uses Kerberos authentication when in a domain, or NTLM authentication when deployed in a workgroup environment. This option uses the Windows token presented by the caller to authenticate against the Active Directory. Service is authenticated by using the Windows credentials of the process identity.
* **Username**. When using this option, the caller provides a username and password to the service. The service can either authenticate against Windows credentials, use a membership provider such as the Microsoft SQL Server® membership provider, or use a custom validator to validate against the custom store. You should choose this option only when Windows authentication is not possible. The service is authenticated by using a service certificate.
* **Certificate**. When using this option, the caller presents an X.509 client certificate. The WCF service looks up the certificate information on the host side and validates it (peer trust), or trusts the issuer of the client certificate (chain trust). This option should be used when Windows authentication is not possible, or in the case of B2B scenarios. The service is authenticated by using a service certificate.
* **Issue** **token**. When using this option, the client and service depend on the Secure Token Service (STS) to issue tokens that the client and service trusts. Microsoft Windows CardSpace™ is a typical example of an STS.

1. **What are the Authorization Options in WCF?**

WCF supports three basic authorization approaches:

* **Role-based**. Access to WCF operations is secured based on the role membership of the caller. Roles are used to partition your application’s user base into sets of users that share the same security privileges within the application; for example, Senior Managers, Managers, and Employees. Users are mapped to roles, and if the user is authorized to perform the requested operation, the application executes the operation.
* **Identity-based**. WCF supports an Identity Model feature, which is an extension of role-based authorization. Identity Model enables you to manage claims and policies in order to authorize clients. With this approach, you can verify claims contained within the authenticated users’ credentials. These claims can be compared with the set of authorization policies for the WCF service. Depending on the claims provided by the client, the service can either grant or deny access to the operation or resources. Identity Model is useful for fine-grained authorization and is most beneficial when using issue token authentication.
* **Resource-based**. With this option, individual resources are secured by using Windows access control lists (ACLs). The WCF service impersonates the caller prior to accessing resources, which allows the operating system to perform standard access checks. All resource access is performed using the original caller’s security context. This authorization approach severely impacts application scalability, because it means that connection pooling cannot be used effectively within the application’s middle tier.

In enterprise-level applications where scalability is essential, a role-based or identity-based approach for authorization is the best choice. For small-scale intranet applications that serve per-user content from resources (such as files) that can be secured with Windows Access Control Lists (ACL), a resource-based approach may be appropriate.

1. **What are the Identities in WCF?**

For an Authorization to work, there are three elements that effect the ability of a service to access resources. These are:

* **Process Identity**: This is the identity of the process hosting the WCF service. When the WCF service is hosted in Internet Information Services (IIS), it typically is NETWORK SERVICE by default. This means that the machine account credentials of the service host are presented to downstream resources. The process identity is important because it identifies what Windows resources and back end the service can access, when the WCF service is not impersonating the original caller.
* **Security Principal**: The principal is a container for the caller’s Identity and the roles that are associated with it. The executing thread includes a security principal that contains the user identity and associated roles. The roles can be Windows roles if the principal is a **Windows Principal**; an ASP.NET role if it is a **role Principal**; or a custom role if it is a **generic Principal**. To be able to authorize, either with the **Roles.IsinRole** call, with **IPrincipals.IsInRole**, or with declarative authorization checks, a security principal must be present in the thread executing the WCF business logic. If a custom authentication is used in WCF, the security principals must be set in a class that derives from **IAuthorizationPolicy**, and this custom authorization policy must be configured in WCF.
* **Service Security Context**: This identity type, available in the WCF run time, contains all of the security-related objects available in the WCF context. These objects are the user identity and authorization context and policies. The service security context is available on both the service and the client side. In the authorization context, you can extract the claim set associated with a security token, whether it is a certificate, issue token, username, or Windows token. To get the service security context on the client side, you need to use the operation context instead.

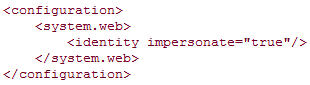
1. **What is Claim-based authorization?**

Claims-based authorization is an approach where the authorization decision to grant or deny access is based on arbitrary logic that uses data available in claims to make the decision. It is a combination of three pieces of information: *the type of claim*, the *right* that is being claimed and the *resource to which the claim* applies. Claims-based security helps decouple the security model from the application domain. A claim can be anything we want to attach to the identity of the user, such as an email, phone number, or flag indicating whether the user is a super user. This gives us the ultimate flexibility on how we want to setup our authorization process. The claims are passed in the security tokens. Once a user is authenticated, the request to the application is made with an associated security token. Windows Identity Foundation (WIF) associates those claims with the principal that represents the user. The application passes the claims to a decision logic mechanism. The decision mechanism calculates the outcome based on the claims. Access is granted based on the outcome.

1. **What is Impersonation?**

An ASP.NET application runs in the context of a specified user account. By default, this is a special account named ASPNET, which has very restricted access to system resources.

Impersonation allows an ASP.NET application to run in the context of the authenticated Windows user that made the web request. This means that all server-side resources accessed by the ASP.NET application occur in the context of the authenticated user. Although this complicates the task of security administration because you must configure access to individual resources for individual users or groups, it gives you fine-grained control over who accesses what resources. To enable impersonation, add an <identity> element to the <system.web> element of your configuration file, as shown in the figure.



You can use Windows authentication with ASP.NET in a number of ways: **http://msdn.microsoft.com/en-us/library/ff647405.aspx**

* *Windows authentication without impersonation*. This is the default setting. ASP.NET performs operations and accesses resources by using your application's process identity, which by default is the Network Service account on Windows Server 2003.
* *Windows authentication with impersonation*. With this approach, you impersonate the authenticated user and use that identity to perform operations and access resources.
* *Windows authentication with fixed-identity impersonation*. With this approach, you impersonate a fixed Windows account to access resources using a specific identity. On Windows Server 2003, you should avoid this impersonation approach; instead, use a custom application pool with a custom service identity.

1. **What is Impersonation and delegation in WCF?**

**Impersonation** is a common technique that WCF services use to assume the original caller's identity in order to authorize access to service resources (such as files or database tables). Service resources can be resources that are either local to the service machine or remotely hosted. Impersonation is used to access resources on the same machine as the service, while **delegation** is used to access resources that are remotely hosted.

By default, **impersonation** is disabled and resources are accessed by using the WCF service's process identity. Impersonation allows you to access local resources and perform other operations using the authenticated user's identity or a specific Windows identity. You can enable **impersonation** either programmatically or by applying appropriate attributes at operation or service levels.

You can **impersonate** *imperatively* or *declaratively*. *Imperative* *impersonation* is performed programmatically at run time and can vary depending on business logic or other conditions. *Declarative* *impersonation* is applied with a static attribute that can be associated with an operation or an entire interface. In general, you should use imperative impersonation when you need the fine granularity made possible by writing the impersonation logic into your code. If you do not need such fine granularity, you can use declarative impersonation.

**Delegation** allows you to use an impersonation token to access network resources. Your ability to use delegation depends on the authentication mechanism in use and appropriate account configuration.

How to Impersonate : <http://blogs.msdn.com/b/saurabs/archive/2012/07/16/wcf-learning-impersonation.aspx>

1. **What is Access Control?**

Microsoft defines the process as follows:

* The client requests a resource on the server.
* The IP address of the client is checked against any IP address restrictions in IIS. If the IP address is denied access, then the request fails and a “403 Access Forbidden” message is returned to the user.
* The server, if configured to require it, requests authentication information from the client. The browser either prompts the user for a user name and password or offers this information automatically.
* IIS checks whether the user has a valid Windows user account. If the user does not, then the request fails and a “401 Access Is Denied” message is returned to the user.
* IIS checks whether the user has Web permissions for the requested resource. If the user does not, then the request fails and a “403 Access Forbidden” message is returned to the user.
* Any security modules, such as Microsoft ASP.NET impersonation, are added.
* IIS checks the NTFS permissions on static files, Active Server Pages (ASP), and Common Gateway Interface (CGI) files for the resource. If the user does not have NTFS permissions for the resource, then the request fails and a “401 Access Is Denied” message is returned to the user.
* If the user has NTFS permissions, the request is fulfilled.

Authenticating users isn’t enough. You also want to filter what your users are able to do. For example, a guest may be authenticated as trusted against your domain but you don’t want him to be able to format your hard disks! That is where the concept of access control comes in. Windows Server 2003 gives you the tools to make sure that you can secure your computer by granting access rights to users or groups of users to the various objects on your system. Objects can be files, folders, or shares. These access rights are called permissions.

The primary technology for providing authentication and access control in Windows Server 2003 is Active Directory.

1. **How will you secure a Web Service?**

A secure system is one in which the following attributes are properly addressed:

* Authentication
* Access control
* Encryption
* Certification and certificates
* Auditing and logging

Authentication is the process of making sure that the person or process that is hitting your server is who or what they claim to be. Access control is about making sure that the users can only hit what they are entitled to hit. Encryption obfuscates the information being passed between your clients and your web server to make sure that someone sniffing the wire can’t steal data, such as passwords or site contents, which they aren’t entitled to. Certification is a logical extension to authentication. Certificates are units of digital identity that are difficult to fake, and as a result provide an additional means of authenticating a user. If the user presents a certificate that is trusted, then the server is sure that the user is who they say they are. Auditing and logging are about tracking user activity to discover naughty behavior or the denial of authorized behavior. These five aspects of security are like jigsaw pieces; when they’re configured to properly fit together, you will have a nice secure web server for your Web services to run on.

1. **What is IIS Access Control?**

You control what your authenticated or anonymous users can do using the principles of access control. Authentication gets people into your site; access control controls what they can do when they are there. Using access control technology, you can restrict them to read and write only what they are allowed to read and write.

The following methods are available for you to secure access to the contents of your web server:

* NTFS security access control
* Web site permissions access control
* Configuring worker process identities
* Securing sites by restricting IP addresses

1. **What is WS-Security Specifications?**

The specification defines SOAP extensions for client authentication, message integrity, and message confidentiality. It is important to note that these are on the message level, not on the wire level like SSL. WS-Security isn’t a new technique for any of these, but rather a method of implementing them using SOAP. It specifies rules for authentication, signatures, and encryption mechanisms.

* Client authentication is the process of establishing and proving the identity of the user. Proper use of authentication schemes can help prevent masquerade attacks, where unauthorized users can gain access by pretending to be someone else, or replay attacks, where stolen authentication information could be reused.
* Message integrity ensures that the message that was sent from the client is the message that is received by your Web service. It ensures that a malicious user cannot intercept messages on the wire and alter them to their needs. WS-Security uses the XML-Signature specification to cryptographically sign SOAP documents. The signatures are defined inside a <Signature> element as part of the security headers. This signature is derived from the SOAP message content as well as a security token, therefore should the message be changed; the signature would no longer be valid. It is effectively a checksum, albeit an advanced one that is computationally infeasible to derive.
* Message confidentiality is the process by which the user is made sure that the data cannot be read during transit. One method of doing this on the wire is SSL, but on the message level the XML Encryption specification is the basis on which portions of a SOAP message may be encrypted. Any part of an XML document may be encrypted, so it is up to the developer to choose whether to encrypt the entire SOAP document or only selected portions that need to be kept private. The latter is probably recommended, as the disadvantages of SSL discussed earlier in this chapter would then be avoided.

The core ingredient of WS-Security is a SOAP header element called <Security>. It contains all the security-related data and information that is needed to implement the supporting functionality, such as signatures or encryption. This header may target different receivers using what are called roles. The role is specified using an attribute of the Security header, and therefore if you want to define the behavior for different targets you can do it by specifying multiple <Security> headers, each with a different role. If you don’t specify the role, any receiver of the message can consume it.

For Example:

<wsse:Security soap:mustUnderstand="1">

<wsu:Timestamp wsu:Id="Timestamp-ad2dbc05-0283-4b70-bde6-fb617d545a3f">

<wsu:Created>2004-07-27T00:34:37Z</wsu:Created>

<wsu:Expires>2004-07-27T01:24:37Z</wsu:Expires>

</wsu:Timestamp>

<wsse:UsernameToken xmlns:wsu="..."

wsu:Id="SecurityToken-b133a645-d389-4bd1-81d7-4bce94cebb0d">

<wsse:Username>jsheridan</wsse:Username>

<wsse:Password Type="...">Delenn12</wsse:Password>

<wsse:Nonce>TdoX/CVGmQ8ueLm1QrYhQQ==</wsse:Nonce>

<wsu:Created>2004-07-27T00:34:37Z</wsu:Created>

</wsse:UsernameToken>

</wsse:Security>

1. **What is TLS/SSL? (\*)**

**Transport Layer Security** (**TLS**) and its predecessor, **Secure Sockets Layer** (**SSL**), both frequently referred to as "SSL", are [*cryptographic protocols*](https://en.wikipedia.org/wiki/Cryptographic_protocol) that provide [communications security](https://en.wikipedia.org/wiki/Communications_security) over a [computer network](https://en.wikipedia.org/wiki/Computer_network). Several versions of the protocols find widespread use in applications such as [web browsing](https://en.wikipedia.org/wiki/Web_browsing), [email](https://en.wikipedia.org/wiki/E-mail), [Internet faxing](https://en.wikipedia.org/wiki/Internet_fax), [instant messaging](https://en.wikipedia.org/wiki/Instant_messaging), and [voice-over-IP](https://en.wikipedia.org/wiki/Voice_over_Internet_Protocol) (VoIP). [Websites](https://en.wikipedia.org/wiki/Website) use TLS to secure all communications between their [servers](https://en.wikipedia.org/wiki/Server_(computing)) and [web browsers](https://en.wikipedia.org/wiki/Web_browser).

The Transport Layer Security protocol aims primarily to provide [privacy](https://en.wikipedia.org/wiki/Privacy) and [data integrity](https://en.wikipedia.org/wiki/Data_integrity) between two communicating computer applications. When secured by TLS, connections between a client (e.g., a web browser) and a server (e.g., wikipedia.org) have one or more of the following properties:

* The connection is private (or secure) because [symmetric cryptography](https://en.wikipedia.org/wiki/Symmetric_cryptography) is used to encrypt the data transmitted. The [keys](https://en.wikipedia.org/wiki/Key_(cryptography)) for this symmetric encryption are generated uniquely for each connection and are based on a [shared secret](https://en.wikipedia.org/wiki/Shared_secret) negotiated at the start of the session (see [TLS handshake protocol](https://en.wikipedia.org/wiki/Transport_Layer_Security#TLS_handshake)). The server and client negotiate the details of which encryption algorithm and cryptographic keys to use before the first byte of data is transmitted (see [Algorithm](https://en.wikipedia.org/wiki/Transport_Layer_Security#Algorithm) below). The negotiation of a shared secret is both secure (the negotiated secret is unavailable to eavesdroppers and cannot be obtained, even by an attacker who places themselves in the middle of the connection) and reliable (no attacker can modify the communications during the negotiation without being detected).
* The identity of the communicating parties can be authenticated using [public-key cryptography](https://en.wikipedia.org/wiki/Public-key_cryptography). This authentication can be made optional, but is generally required for at least one of the parties (typically the server).
* The connection ensures integrity because each message transmitted includes a message integrity check using a [message authentication code](https://en.wikipedia.org/wiki/Message_authentication_code) to prevent undetected loss or alteration of the data during transmission.

1. **What is HTTPS? (\*)**

**HTTPS** (also called ***HTTP over***[***Transport Layer Security***](https://en.wikipedia.org/wiki/Transport_Layer_Security) (TLS), ***HTTP over SSL***, and ***HTTP Secure***) is a [communications protocol](https://en.wikipedia.org/wiki/Communications_protocol) for [secure communication](https://en.wikipedia.org/wiki/Secure_communication) over a [computer network](https://en.wikipedia.org/wiki/Computer_network) which is widely used on the [Internet](https://en.wikipedia.org/wiki/Internet). HTTPS consists of communication over [Hypertext Transfer Protocol](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) (HTTP) within a connection encrypted by *Transport Layer Security*, or its predecessor, *Secure Sockets Layer*. The main motivation for HTTPS is [authentication](https://en.wikipedia.org/wiki/Authentication) of the visited [website](https://en.wikipedia.org/wiki/Website) and protection of the [privacy](https://en.wikipedia.org/wiki/Information_privacy) and [integrity](https://en.wikipedia.org/wiki/Data_integrity) of the exchanged data.

In its popular deployment on the internet, HTTPS provides authentication of the website and associated [web server](https://en.wikipedia.org/wiki/Web_server) with which one is communicating, which protects against [man-in-the-middle attacks](https://en.wikipedia.org/wiki/Man-in-the-middle_attack). Additionally, it provides bidirectional [encryption](https://en.wikipedia.org/wiki/Encryption) of communications between a client and server, which protects against [eavesdropping](https://en.wikipedia.org/wiki/Eavesdropping) and [tampering](https://en.wikipedia.org/wiki/Tamper-evident#Tampering) with or forging the contents of the communication. In practice, this provides a reasonable guarantee that one is communicating with precisely the website that one intended to communicate with (as opposed to an impostor), as well as ensuring that the contents of communications between the user and site cannot be read or forged by any third party.

# 

# *Memory Management*

1. **Some Memory Fundamentals**

The following list summarizes important CLR memory concepts.

* Each process has its own, separate virtual address space. All processes on the same computer share the same physical memory, and share the page file if there is one.
* As an application developer, you work only with virtual address space and never manipulate physical memory directly. The garbage collector allocates and frees virtual memory for you on the managed heap.
* Virtual memory can be in three states:
  + Free. The block of memory has no references to it and is available for allocation.
  + Reserved. The block of memory is available for your use and cannot be used for any other allocation request. However, you cannot store data to this memory block until it is committed.
  + Committed. The block of memory is assigned to physical storage.
* Virtual address space can get fragmented. This means that there are free blocks, also known as holes, in the address space. When a virtual memory allocation is requested, the virtual memory manager has to find a single free block that is large enough to satisfy that allocation request. Even if you have 2 GB of free space, the allocation that requires 2 GB will be unsuccessful unless all of that space is in a single address block.
* You can run out of memory if you run out of virtual address space to reserve or physical space to commit.

Your page file is used even if physical memory pressure (that is, demand for physical memory) is low. The first time your physical memory pressure is high, the operating system must make room in physical memory to store data, and it backs up some of the data that is in physical memory to the page file. That data is not paged until it is needed, so it is possible to encounter paging in situations where the physical memory pressure is very low.

1. **What are the conditions for garbage collection**

Garbage collection occurs when one of the following conditions is true:

* The system has low physical memory.
* The memory that is used by allocated objects on the managed heap surpasses an acceptable threshold. This means that a threshold of acceptable memory usage has been exceeded on the managed heap. This threshold is continuously adjusted as the process runs.
* The [GC.Collect](http://msdn.microsoft.com/en-us/library/system.gc.collect.aspx) method is called. In almost all cases, you do not have to call this method, because the garbage collector runs continuously. This method is primarily used for unique situations and testing.

1. **What is managed heap?**

* After the garbage collector is initialized by the CLR, it allocates a segment of memory to store and manage objects. This memory is called the managed heap, as opposed to a native heap in the operating system.
* There is a managed heap for each managed process. All threads in the process allocate objects on the same heap.
* When a garbage collection is triggered, the garbage collector reclaims the memory that is occupied by dead objects. The reclaiming process compacts live objects so that they are moved together, and the dead space is removed, thereby making the heap smaller. This ensures that objects that are allocated together stay together on the managed heap, to preserve their locality.

1. **What are Generations?**

The heap is organized into generations so it can handle long-lived and short-lived objects. Garbage collection primarily occurs with the reclamation of short-lived objects that typically occupy only a small part of the heap. There are three generations of objects on the heap:

* ***Generation 0***. This is the youngest generation and contains short-lived objects. An example of a short-lived object is a temporary variable. Garbage collection occurs most frequently in this generation.

Newly allocated objects form a new generation of objects and are implicitly generation-0 collections, unless they are large objects, in which case they go on the large object heap in a generation-2 collection.

Most objects are reclaimed for garbage collection in generation 0 and do not survive to the next generation.

* ***Generation 1***. This generation contains short-lived objects and serves as a buffer between short-lived objects and long-lived objects.
* ***Generation 2***. This generation contains long-lived objects. An example of a long-lived object is an object in a server application that contains static data that is live for the duration of the process.

***Garbage collections*** occur on specific generations as conditions warrant. Collecting a generation means collecting objects in that generation and all its younger generations. A generation 2 garbage collection is also known as a full garbage collection, because it reclaims all objects in all generations (that is, all objects in the managed heap).

Objects that are not reclaimed in a garbage collection are known as survivors, and are promoted to the next generation. Objects that survive a generation-0 garbage collection are promoted to generation-1; objects that survive a generation-1 garbage collection are promoted to generation-2; and objects that survive a generation-2 garbage collection remain in generation 2.

1. **What happens during a Garbage Collection?**

A garbage collection has the following phases:

* A marking phase that finds and creates a list of all live objects.
* A relocating phase that updates the references to the objects that will be compacted.
* A compacting phase that reclaims the space occupied by the dead objects and compacts the surviving objects. The compacting phase moves objects that have survived a garbage collection toward the older end of the segment.

Because generation 2 collections can occupy multiple segments, objects that are promoted into generation 2 can be moved into an older segment. Both generation 1 and generation 2 survivors can be moved to a different segment, because they are promoted to generation 2.

The large object heap is not compacted, because this would increase memory usage over an unacceptable length of time.

The garbage collector uses the following information to determine whether objects are live:

* ***Stack roots***. Stack variables provided by the just-in-time (JIT) compiler and stack walker.
* ***Garbage collection handles***. Handles that point to managed objects and that can be allocated by user code or by the common language runtime.
* ***Static data***. Static objects in application domains that could be referencing other objects. Each application domain keeps track of its static objects.

Before a garbage collection starts, all managed threads are suspended except for the thread that triggered the garbage collection.

1. **What is garbage collection?**

The .NET Framework memory management relies on a sophisticated process known as ***garbage collection***, or GC. When an application tries to allocate memory for a new object and the heap has insufficient free memory, the .NET Framework starts the garbage collection process.

The garbage collector visits all the objects in the heap and marks those objects that are pointed to by any variable in the application. (These variables are known as roots because they’re at the top of an object graph.) This process is sophisticated in that it also recognizes objects referenced indirectly from other objects, such as when you have a Person object that references another Person object through its Spouse property. After marking all the objects that can be reached from the application’s code, the garbage collector can safely release the remaining (unmarked) objects because they’re guaranteed to be unreachable by the application. Next the garbage collector compacts the heap and makes the resulting block of free memory available to new objects. Interestingly, this mechanism indirectly resolves the circular reference problem because the garbage collector doesn’t mark unreachable objects and therefore correctly releases memory associated with objects pointed to by other objects in a circular reference fashion but not used by the main program.

1. **What’s the significance of Finalize method in .NET?**

If your managed objects reference unmanaged objects by using their native file handles, you have to explicitly free the managed objects, because the garbage collector tracks memory only on the managed heap.

Users of your managed object may not dispose the native resources used by the object. To perform the cleanup, you can make your managed object ***finalizable***. Finalization consists of cleanup actions that you execute when the object is no longer in use. When your managed object dies, it performs cleanup actions that are specified in its finalizer method. Since Finalize method does nothing, by default, this method must be overridden if explicit cleanup is required.

When a finalizable object is discovered to be dead, its finalizer is put in a queue, known as finalizing Queue, so that its cleanup actions are executed, but the object itself is promoted to the next generation. Therefore, you have to wait until the next garbage collection that occurs on that generation (which is not necessarily the next garbage collection) to determine whether the object has been reclaimed.

You should note the following points should when implementing Finalizers:

* ***Finalizers*** should always be protected, not public or private so that the method cannot be called from the application's code directly and at the same time, it can make a call to the ***base.Finalize*** method.
* ***Finalizers*** should release unmanaged resources only.
* The framework does not guarantee that a ***finalizer*** will execute at all on any given instance.
* Never allocate memory in ***Finalizers*** or call virtual methods from ***Finalizers***.
* Avoid synchronization and raising unhandled exceptions in the ***Finalizers***.
* The execution order of ***Finalizers*** is non-deterministic—in other words, you can't rely on another object still being available within your finalizer.
* Do not define ***Finalizers*** on value types.
* Don't create empty destructors. In other words, you should never explicitly define a destructor unless your class needs to clean up unmanaged resources—and if you do define one, it should do some work. If, later, you no longer need to clean up unmanaged resources in the destructor, remove it altogether.

1. **What is a Finalizer Queue?**

The ***Finalizer Queue*** holds all objects that have a ***finalizer*** method defined. When the garbage collector collects garbage, it moves any objects with a finalizer into the finalizer queue. At some point later-- depending on memory pressure or GC heuristics, when the garbage collector decides to collect these objects, it walks down the queue and runs the ***Finalizers***.

1. **How can we force garbage collection and how can we suppress a finalize method?**

***System.GC.Collect()*** forces garbage collector to run. If you run a garbage collection frequently, you’re missing one of the most promising performance optimizations that the new .NET Framework offers. ***GC.SuppressFinalize*** is used to suppress a Finalize method.

1. **What is IDisposable, Dispose and Close method?**

Some objects require explicit tear-down code to release resources such as open files, locks, operating system handles, and unmanaged objects. In .NET parlance, this is called disposal, and it is supported through the ***IDisposable*** interface. This interface exposes only one method, ***Dispose***.

***Disposal*** differs from ***Garbage-Collection*** in that disposal is usually explicitly instigated; garbage collection is totally automatic. In other words, the programmer takes care of such things as releasing file handles, locks, and operating system resources while the CLR takes care of releasing memory using Garbage-Collection.

Some types define a method called ***Close*** in addition to ***Dispose***. The Framework is not completely consistent on the semantics of a Close method, although in nearly all cases it’s either:

* Functionally identical to ***Dispose***
* A functional subset of ***Dispose***

An example of the latter is IDbConnection: a ***Closed*** connection can be re-Opened; a Disposed connection cannot. Another example is a Windows Form activated with ***ShowDialog***: ***Close*** hides it; ***Dispose*** releases its resources.

1. **How to force the Dispose method to be called automatically during finalization?**

There are two alternatives for freeing the unmanaged resources used by the classes:

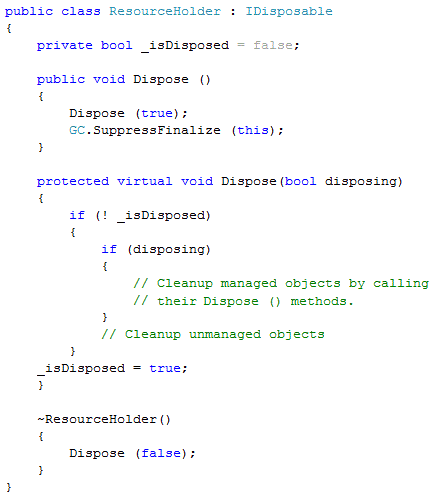
* The execution of a destructor is enforced by the runtime but is nondeterministic and places an unacceptable overhead on the runtime because of the way garbage collection works.
* The IDisposable interface provides a mechanism that allows users of a class to control when resources are freed, but requires discipline to ensure that Dispose () is called.

In general, the best approach is to implement both mechanisms in order to gain the benefits of both while overcoming their limitations. You implement IDisposable on the assumption that most programmers will call Dispose () correctly, but implement a destructor as a safety mechanism in case Dispose () is not called. Here is an example of a dual implementation:

In the code there is a second protected overload of Dispose (), which takes one bool parameter—and this is the method that does all cleaning up. Dispose (bool) is called by both the destructor and by ***IDisposable.Dispose***(). The point of this approach is to ensure that all cleanup code is in one place.

The parameter passed to Dispose (bool) indicates whether Dispose (bool) has been invoked by the destructor or by ***IDisposable.Dispose*** ()—Dispose (bool) should not be invoked from anywhere else in your code.

The idea is this:



* If a consumer calls ***IDisposable.Dispose*** (), then that consumer is indicating that all managed and unmanaged resources associated with that object should be cleaned up.
* If a destructor has been invoked, then all resources still need to be cleaned up. However, in this case, we know that the destructor must have been called by the garbage collector and we should not attempt to access other managed objects because we can no longer be certain of their state. In this situation, the best we can do is clean up the known unmanaged resources, and hope that any referenced managed objects also have destructors that will perform their own cleaning up.

1. **What are Value types and Reference types?**

In C#, all the type can be categorized in four type: **Value type**, **Reference type**, **Generic type** and **Pointers**.

***Value types*** comprise most built-in types (specifically, all numeric types, the char type, and the bool type) as well as custom struct and enum types.

***Reference types*** comprise all class, array, delegate, string and interface types. It is has two parts: an ***Object*** and the ***Reference*** to that object (aka. Pointer).

Both the types are handled differently in memory. This is governed by the following rule:

* A ***Reference*** ***Type*** (***Objects***) always goes on Heap and,
* ***Value*** ***Type*** and ***Pointer*** (***Reference***) always goes where they are declared.

Whenever a method is executed, all the variables, including the parameters, are brought to the stack memory in LIFO pattern. The ***Value*** ***types***, declared inside the method, holds the value directly, whereas the ***Reference*** holds the address of the object. The object properties, if they are value types, will be treated like value types thus holding the actual value.

Whenever a ***value*** ***type*** is assigned to another ***value*** ***type***, the actual value is copied. However, if a ***reference*** ***type*** is assigned to another ***reference*** ***type***, the actual object remains the same and only the address is copied.

1. **Where are Methods stored in the memory? (\*)**

Methods are stored somewhere else in the memory, not in stack or heap. Methods are per-class, not per-instance. So typically, the number of methods doesn't change over the run-time of a program (there are exceptions). In traditional models, the place where the methods live is called the "code segment". In .net, it's more difficult: the methods originally live in the assembly, and get mapped into the process memory. There, the just-in-time compiler creates a second copy of some methods in native code; this copy gets executed. The JIT code may get created and deleted several times during the runtime, so it is practical to view it also as living "in Heap".

During the runtime, if the method is not called for, say, an hour, then the JIT may discard it and then may recompile it two hours later and so on. To support simultaneous (or subsequent) activations of a method, "call stacks" are used, consisting of "stack frames". Every invocation of a method creates a new stack frame (in a memory region called the "stack"), consisting of all parameters to the method, plus any local variables.

# *Threading*

1. **Why is Threading used?**

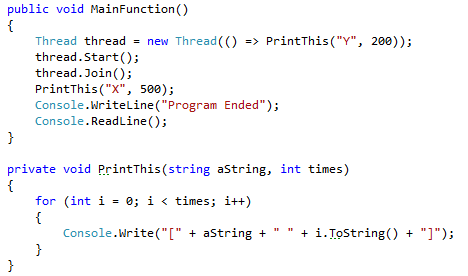
A thread is defined as the execution path of a program. Each thread defines a unique flow of control. It allows to execute code in parallel through multithreading. Multithreading has many uses:

* Maintaining a responsive user interface
* Making efficient use of an otherwise blocked CPU
* Parallel programming, by making use of multicore or multiprocessor computers.
* Speculative execution, by doing task ahead of time.
* Asynchronously processing the requests. Although, .Net does is automatically for WCF, ASP.Net apps.

1. **What is Join and Sleep?**

**Join** blocks the calling thread until the thread it is called upon, terminates. **Join** is a *synchronization* method that blocks the *calling**thread* until the thread whose *Join* method is called has completed. A **calling thread** is the one that calls the method. This method is used to ensure that a thread has been terminated. The caller thread will be blocked indefinitely if the thread does not terminate. Join method takes a timeout parameter, either in milliseconds or timespan. If the thread has ended naturally, then Join returns true otherwise false, if timed out.

The following program prints Y and X. In this program, the *Calling thread* is *MainFunction* and *Caller thread* is the *new* *thread* that is created inside the *MainFunction*. When the Caller thread invokes Join, then the program first prints the Y for 200 times and then, the X is printed for 500 times. Thus it is synchronized, Printing Y followed by X. The *MainFunction* is blocked. if the Join method is removed then X and Y will be printed asynchronously, mixing up the result.

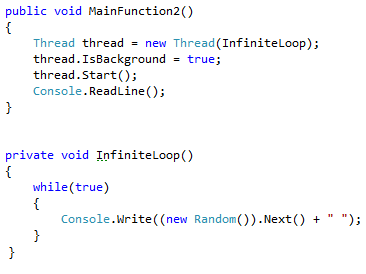


**Sleep** is a static method of Thread class, unlike **Join**, which is an instance method. **Sleep** method pauses the current thread for a specified period of time. The thread will not be scheduled for execution by the operating system for the amount of time specified and the state of the thread changes to **WaitSleepJoin**. Like Join, it is also used for synchronization, however not recommended to do so.

1. **What is Foreground and Background thread?**

Threads can be categorized in two types: Foreground and Background. Foreground threads are the ones which continue to execute until the last Foreground thread is terminated. When all the Foreground threads are closed, the application closes. Background threads are the ones which are terminated when all the Foreground threads are closed. The application does not wait for the Background thread to be completed.

In the side example, the program executes infinitely, writing Random numbers to the console. If the thread is Foreground then the program cannot be stopped. It has to be terminated, manually. If the thread is Background, then the program can be stopped by pressing enter.



1. **What is a thread pool?**

Whenever you start a thread, a few hundred microseconds are spent organizing such things as a fresh private local variable stack. Each thread also consumes (by default) around 1 MB of memory. The thread pool cuts these overheads by sharing and recycling threads, allowing multithreading to be applied at a very granular level without a performance penalty. This is useful when leveraging multicore processors to execute computationally intensive code in parallel in “divide-and-conquer” style.

The thread pool also keeps a lid on the total number of worker threads it will run simultaneously. Too many active threads throttle the operating system with administrative burden and render CPU caches ineffective. Once a limit is reached, jobs queue up and start only when another finishes. This makes arbitrarily concurrent applications possible, such as a web server.

1. **How a thread pool is accessed?**

There are a number of ways to enter the thread pool:

* Via the Task Parallel Library or PLINQ (from Framework 4.0)
* By calling ThreadPool.QueueUserWorkItem
* Via asynchronous delegates
* Via BackgroundWorker

1. **What is Background Worker?**

The **BackgroundWorker** class allows to run an operation on a separate, dedicated thread. Time-consuming operations, such as downloads and database transactions, can cause the user interface to stop responding. When you want a responsive user interface and you must perform time-consuming operations, the **BackgroundWorker** class provides a convenient solution.

**BackgroundWorker** is a helper class in the System.ComponentModel namespace for managing a worker thread. It can be considered a general-purpose implementation of the **EAP**(Event-based Asynchronous Pattern), and 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
* A protocol for reporting progress
* An implementation of IComponent allowing it to be sited in Visual Studio’s designer

BackgroundWorker uses the thread pool, which means you should never call Abort on a BackgroundWorker thread.

1. **How a BackgroundWorker is used and how to cancel the BackgroundWorker thread?**

There are minimum two steps to use a **BackgroundWorker**

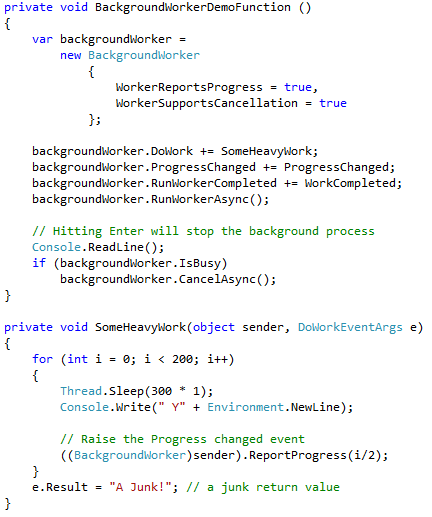
1. Instantiate BackgroundWorker and handle the **DoWork** event.
2. Call **RunWorkerAsync**, optionally with an object argument.

This then sets the **BackgroundWorker** in motion. Any argument passed to **RunWorkerAsync** will be forwarded to **DoWork’s** event handler, via the event argument’s Argument property.

To add support for cancellation:

1. Set the **WorkerSupportsCancellation** property to true.
2. Periodically check the **CancellationPending** property from within the **DoWork** event handler. If it’s true, set the event argument’s **Cancel** property to true, and return. (The worker can also set Cancel and exit without **CancellationPending** being true if it decides that the job is too difficult and it can’t go on.)
3. Call **CancelAsync** to request cancellation.

Here’s an example.

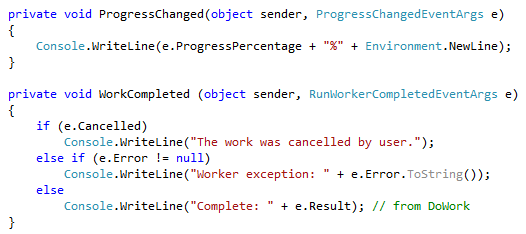


A **BackgroundWorker** is defined with **WorkerReportsProgress** and **WorkerSupportsCancellation** set to true. The three events, **DoWork**, **ProgressChanged** and **RunWorkerCompleted** are assigned their respective handlers.

The **RunWorkerAsync** method invokes the actual method to do some heavy work.

Inside the **SomeHeavyWork**, the sender is type casted to the **BackgroundWorker** and then **ReportProgress** method is invoked. This method actually raises the **ProgressChanged** event of the **BackgroundWorker**.

The **ProgressChanged** event handler and **WorkCompleted** are pretty much straight forward.



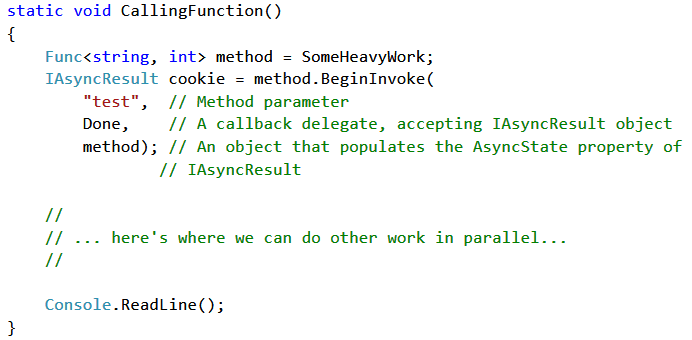
1. **What is Asynchronous Delegates?**

**Delegates** enable you to call a synchronous method in an asynchronous manner. When you call a delegate synchronously, the Invoke method calls the target method directly on the current thread. If the **BeginInvoke** method is called, the common language runtime (**CLR**) queues the request and returns immediately to the caller. The target method is called asynchronously on a thread from the thread pool. The original thread, which submitted the request, is free to continue executing in parallel with the target method. If a callback method has been specified in the call to the **BeginInvoke** method, the callback method is called when the target method ends. In the callback method, the **EndInvoke** method obtains the return value and any input/output or output-only parameters. If no callback method is specified when calling **BeginInvoke**, **EndInvoke** can be called from the thread that called **BeginInvoke**.

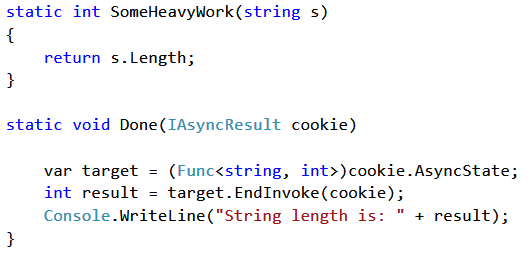
The **BeginInvoke** method initiates the asynchronous call. It has the same parameters as the method that you want to execute asynchronously, plus two additional optional parameters. The first parameter is an AsyncCallback delegate that references a method to be called when the asynchronous call completes. The second parameter is a user-defined object that passes information into the callback method. **BeginInvoke** returns immediately and does not wait for the asynchronous call to complete. **BeginInvoke** returns an **IAsyncResult**, which can be used to monitor the progress of the asynchronous call.

The **EndInvoke** method retrieves the results of the asynchronous call. It can be called any time after **BeginInvoke**. If the asynchronous call has not completed, **EndInvoke** blocks the calling thread until it completes. The parameters of **EndInvoke** include the **out** and **ref** parameters of the method that you want to execute asynchronously, plus the **IAsyncResult** returned by **BeginInvoke**.

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. The **BeginInvoke** method has a callback delegate - a method accepting an **IAsyncResult** object that’s automatically called upon completion. This allows the instigating thread to “forget” about the asynchronous delegate.



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.

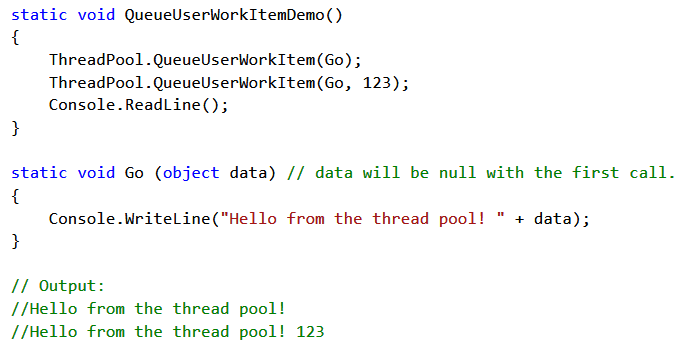


**EndInvoke** does three things. First, it waits for the asynchronous delegate to finish executing, if it hasn’t already. Second, it receives the return value (as well as any ref or out parameters). Third, it throws any unhandled worker exception back to the calling thread.

1. **What is QueueUserWorkItem and how it is different than Async Delegates?**

The **Thread**.**QueueUserWorkItem** is one of the ways to enter the thread pool, without **Task Parallel Library (TPL)**. To use **QueueUserWorkItem**, simply call this method with a delegate that you want to run on a pooled thread:

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.

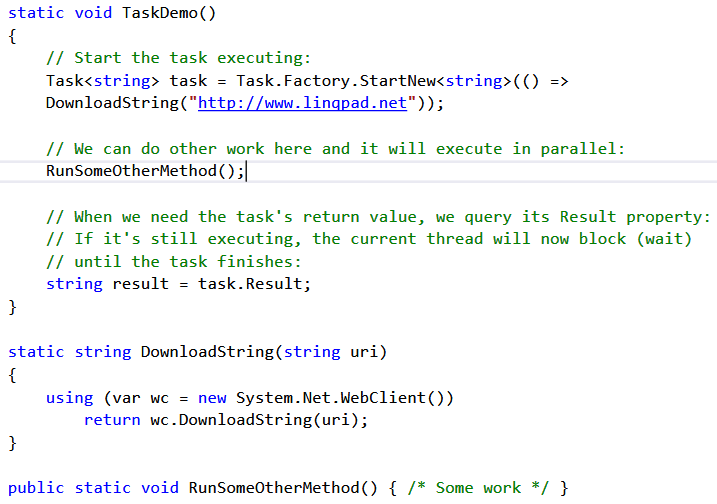


The difference between the **Asynchronous Delegates** and **QueueUserWorkItem** is that asynchronous delegates let you return data from the thread. **Asynchronous delegates** also marshal any exception back to the caller. **ThreadPool.QueueUserWorkItem** doesn’t provide an easy mechanism for getting 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 re-thrown on the original thread (or more accurately, the thread that calls EndInvoke), and so they don’t need explicit handling.

1. **What is Task?**

The **Task** class in **Task Parallel Library (TPL)** allows to enter the thread pool. This feature is present in Framework 4.0. The non-generic Task class is a replacement for **Thread.QueueUserWorkItem**. The generic **Task<TResult>** is a replacement for **Asynchronous Delegates**. It lets you get a return value back from the task after it finishes executing. This is done using Result property of task.

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.



1. **What is Synchronization and how it is implemented?**

Coordinating the action of threads for a predictable outcome is known as Synchronization. Synchronization is particularly important when threads access the same data. Synchronization constructs can be divided into four categories:

**Simple blocking methods**: These wait for another thread to finish or for a period of time to elapse. **Sleep**, **Join**, and **Task.Wait** are simple blocking methods.

**Locking constructs**: These limit the number of threads that can perform some activity or execute a section of code at a time. Exclusive locking constructs are most common—these allow just one thread in at a time, and allow competing threads to access common data without interfering with each other. The standard exclusive locking constructs are lock (**Monitor**.**Enter/Monitor.Exit**), **Mutex**, and **SpinLock**. The nonexclusive locking constructs are **Semaphore**, **SemaphoreSlim**, and **ReaderWriterLockSlim**.

**Signaling constructs**: These allow a thread to pause until receiving a notification from another, avoiding the need for inefficient polling. There are two commonly used signaling devices: event wait handles and **Monitor’s** **Wait/Pulse** methods. Framework 4.0 introduces the **CountdownEvent** and **Barrier** classes.

**Nonblocking synchronization constructs**: These protect access to a common field by calling upon processor primitives. The CLR and C# provide the following nonblocking constructs: **Thread.MemoryBarrier, Thread.VolatileRead, Thread.VolatileWrite**, the **volatile** keyword, and the **Interlocked** class.

1. **What is Thread Blocking?**

A thread is deemed blocked when its execution is paused for some reason, such as when Sleeping or waiting for another to end via Join or EndInvoke. A blocked thread immediately yields its processor time slice, and from then on consumes no processor time until its blocking condition is satisfied. You can test for a thread being blocked via its ThreadState property:

bool blocked = (someThread.ThreadState & ThreadState.WaitSleepJoin) != 0;

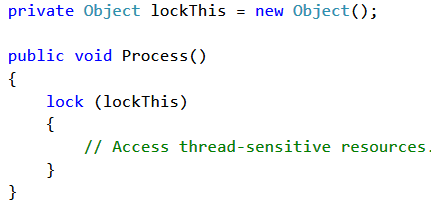
1. **What is Locking?**

Locking is essential in threaded programs. It restricts code from being executed by more than one thread at the same time. This makes threaded programs reliable. Exclusive locking is used to ensure that only one thread can enter particular sections of code at a time. The two main exclusive locking constructs are **Lock** and **Mutex**. Of the two, the **lock** construct is faster and more convenient. **Mutex**, though, has a niche in that its lock can span applications in different processes on the computer.

1. **How Lock is used and how it is different from Monitors?**

The **lock** statements can be used to ensure that a block of code runs to completion without interruption by other threads. This is accomplished by obtaining a mutual-exclusion lock for a given object for the duration of the code block. A **lock** statement is given an object as an argument, and is followed by a code block that is to be executed by only one thread at a time.

The argument provided to the **lock** keyword must be an **object** based on a reference type, and is used to define the scope of the **lock**. In the example above, the lock scope is limited to this function because no references to the object *lockThis* exist outside the function. If such a reference did exist, lock scope would extend to that object. Strictly speaking, the object provided is used solely to uniquely identify the resource being shared among multiple threads, so it can be an arbitrary class instance. In practice, however, this object usually represents the resource for which thread synchronization is necessary. For example, if a container object is to be used by multiple threads, then the container can be passed to lock, and the synchronized code block following the lock would access the container. As long as other threads locks on the same contain before accessing it, then access to the object is safely synchronized.



The **Monitor** class also does the same thing like **Lock**, by using **Monitor.Enter** and **Monitor.Exit** methods, along with Try and Finally blocks. However, **Lock** is more concise and it ensures that the underlying **Monitor** is released even if the protected code throws an exception.

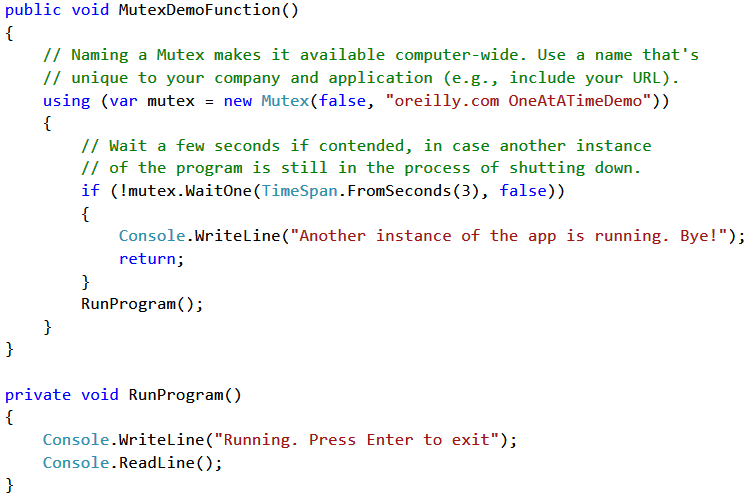
1. **What is Mutex and how it is used?**

When two or more threads need to access a shared resource at the same time, the system needs a synchronization mechanism to ensure that only one thread at a time uses the resource. Mutex is a synchronization primitive that grants exclusive access to the shared resource to only one thread. If a thread acquires a mutex, the second thread that wants to acquire that mutex is suspended until the first thread releases the mutex.

Unlike monitors, however, **a mutex can be used to synchronize threads across processes.** When used for inter-process synchronization, a mutex is called a **named mutex** because it is to be used in another application, and therefore it cannot be shared by means of a global or static variable. It must be given a name so that both applications can access the same mutex object.

With a **Mutex** class, you call the **WaitOne** method to lock and **ReleaseMutex** to unlock. Closing or disposing a Mutex automatically releases it. Just as with the **lock** statement, a **Mutex** can be released only from the same thread that obtained it.

A common use for a cross-process Mutex is to ensure that only one instance of a program can run at a time. Here’s how it’s done:

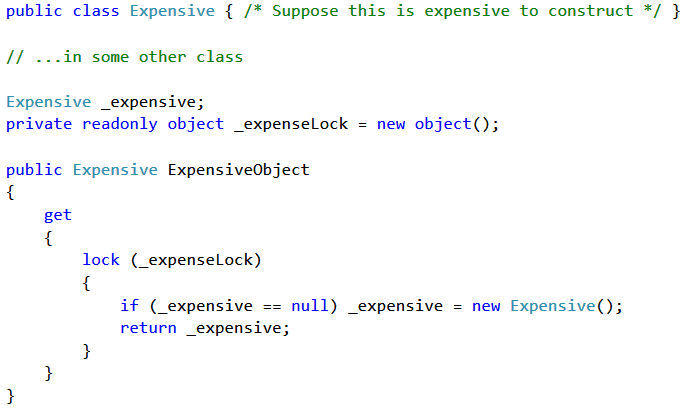


1. **What is Thread Safety?**

Thread safety is a computer programming concept applicable in the context of multi-threaded programs. A piece of code is thread-safe if it only manipulates shared data structures in a manner that guarantees safe execution by multiple threads at the same time. There are various ways to ensure thread safety like *Mutual Exclusion* where just one thread reads or writes the shared data. *Thread-local storage* can also be used to ensure the thread safety. In this the variables are localized so that each thread has its own private copy. Using *immutable* objects can also result in thread safety.

1. **What is Lazy Initialization?**

*Lazy initialization* of an object means that its creation is deferred until it is first used. Lazy initialization is primarily used to improve performance, avoid wasteful computation, and reduce program memory requirements. A thread safe version of lazy initialization can be done as side example.



Framework 4.0 provides a new class called **Lazy<T>** to help with lazy initialization. If instantiated with an argument of true, it implements the thread-safe initialization pattern just described. **Lazy<T>** actually implements a slightly more efficient version of this pattern, called double-checked locking. Double-checked locking performs an additional volatile read to avoid the cost of obtaining a lock if the object is already initialized. To use **Lazy<T>**, instantiate the class with a value factory delegate that tells it how to initialize a new value, and the argument true. Then access its value via the Value property:

Lazy<Expensive> \_expensive = new Lazy<Expensive> (() => new Expensive (), true);

public static Expensive Expensive {get {return \_expensive.Value;} }

If you pass false into **Lazy<T>**’s constructor, it implements the thread-unsafe lazy initialization pattern that we described at the start of this section—this makes sense when you want to use **Lazy<T>** in a single-threaded context.

1. **What are Asynchronous methods?**

To avoid performance bottlenecks and enhance the overall responsiveness of an application, asynchronous programming is used. However, traditional techniques for writing asynchronous applications can be complicated, making them difficult to write, debug, and maintain. Framework4.5 introduces a simplified approach, **async** programming, that leverages asynchronous support in the .NET Framework 4.5 and the Windows Runtime. The compiler does the difficult work that the developer used to do, and the application retains a logical structure that resembles synchronous code. As a result, you get all the advantages of asynchronous programming with a fraction of the effort. The **async** and **await** keywords in C# are the heart of async programming. By using those two keywords, you can use resources in the .NET Framework or the Windows Runtime to create an asynchronous method almost as easily as you create a synchronous method. Asynchronous methods that you define by using **async** and **await** are referred to as async methods.

The following example shows an async method. Almost everything in the code should look completely familiar to you. The comments call out the features that you add to create the asynchrony.

// Three things to note in the signature:

// - The method has an async modifier.

// - The return type is Task or Task<T>. (See "Return Types" section.)

// Here, it is Task<int> because the return statement returns an integer.

// - The method name ends in "Async."

async Task<int> AccessTheWebAsync ()

{

// You need to add a reference to System.Net.Http to declare client.

HttpClient client = new HttpClient ();

// GetStringAsync returns a Task<string>. That means that when you await the

// task you'll get a string (urlContents).

Task<string> getStringTask = client.GetStringAsync ("http://msdn.microsoft.com");

// You can do work here that doesn't rely on the string from GetStringAsync.

DoIndependentWork ();

// The await operator suspends AccessTheWebAsync.

// - AccessTheWebAsync can't continue until getStringTask is complete.

// - Meanwhile, control returns to the caller of AccessTheWebAsync.

// - Control resumes here when getStringTask is complete.

// - The await operator then retrieves the string result from getStringTask.

string urlContents = await getStringTask;

// The return statement specifies an integer result.

// Any methods that are awaiting AccessTheWebAsync retrieve the length value.

return urlContents.Length;

}

1. **What is Thread Affinity?**

All WPF applications start out with two important threads, one for rendering and one for managing the user interface. The rendering thread is a hidden thread that runs in the background, so the only thread that you ordinarily deal with is the UI thread. WPF requires that most of its objects be tied to the UI thread. This is known as **Thread** **Affinity**, meaning that only the thread that instantiates the WPF objects, can subsequently access their members. Using it on other threads will cause a runtime exception to be thrown or an unpredictable behavior.

On the positive side, this means we do not have to lock around accessing a UI object. On the negative side, if you want to call a member on object X created on another thread Y, you must marshal the request to thread Y. You can do this explicitly by calling **Invoke (synchronous)** or **BeginInvoke (asynchronous)** on the element’s **Dispatcher** object.

1. **What is Thread Local Storage?**

In single threaded applications, global variables are used to store data that are required by the other parts of the program. These are the data which are not passed from one method to another, but are kept at a single place so that they can be reused and updated. However, in case of multi-threaded application, the global variable exposes couple of drawbacks. Firstly, it is not thread-safe. A thread can modify the global variable, overriding the old value which might be crucial for the other threads. Secondly, there can be race condition among threads to get hold of global variable.

The Thread local storage (TLS) acts like a global variable for a multi-threaded application, where each thread has its own copy of global variable. TLS is a variable that looks like global but exists once per thread.

It is useful in storing the execution path’s infra-structure, like messaging, transaction and security tokens. Passing such data around in methods is extremely clumsy and alienates all but your own methods. There are three ways to implement thread local storage:

1. Using [ThreadStatic] attribute. E.g.: [ThreadStatic] static int \_x;
2. Using ThreadLocal<T>. E.g.: static ThreadLocal<int> \_x = new ThreadLocal<int> (() => 3);
3. Using GetData and SetData. E.g.: object data = Thread.GetData (\_secSlot); Thread.SetData (\_secSlot, value);
4. **What is Signaling method of Synchronization?**

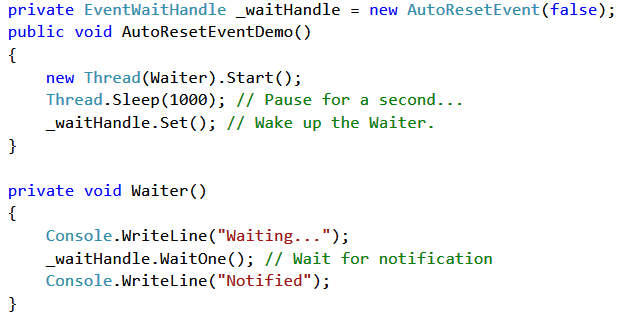
Synchronization is a way to coordinate the actions of the threads for a predictable outcome. It becomes important when multiple thread tries to access the same data. To implement Synchronization, Signals are one of the various ways. These allow a thread to pause until receiving a notification from other thread. There are two mechanism to use Signal

1. Event Wait handles (**AutoResetEvent**, **ManualResetEvent** and **CountDownEvent**) and
2. Monitor’s Wait/Pulse method
3. **What is AutoResetEvent and ManualResetEvent?**

The **AutoResetEvent** is like a Tollbooth where inserting a ticket will allow only one Car to pass through. The Auto in the class’s name refers to the fact that an open Tollbooth automatically closes or resets when a car moves through, thus blocking the cars behind.

**AutoResetEvent** allows threads to communicate with each other by signaling. Typically, you use this class when threads need exclusive access to a resource.

A thread waits, or blocks, at the Tollbooth by calling **WaitOne** (wait at this “one” Tollbooth until it opens), and a ticket is inserted by calling the Set method. If a number of threads call **WaitOne**, a queue builds up behind the Tollbooth. A ticket can come from any thread; in other words, any (unblocked) thread with access to the **AutoResetEvent** object can call **Set** on it to release one blocked thread.



A thread waits for a signal by calling **WaitOne** on the **AutoResetEvent**. If the **AutoResetEvent** is in the non-signaled state, the thread blocks, waiting for the thread that currently controls the resource to signal that the resource is available by calling **Set**.

Calling **Set** signals **AutoResetEvent** to release a waiting thread. **AutoResetEvent** remains signaled until a single waiting thread is released, and then automatically returns to the on-signaled state. If no threads are waiting, the state remains signaled indefinitely.

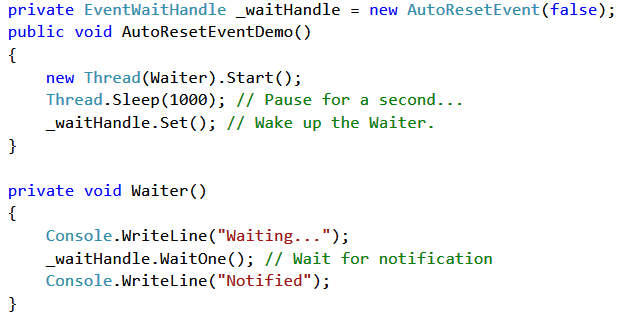
If a thread calls **WaitOne** while the **AutoResetEvent** is in the signaled state, the thread does not block. The **AutoResetEvent** releases the thread immediately and returns to the non-signaled state.

A **ManualResetEvent** functions like an ordinary gate. Calling Set opens the gate, allowing any number of threads calling **WaitOne** to be let through. Calling **Reset** closes the gate. Threads that call **WaitOne** on a closed gate will block; when the gate is next opened, they will be released all at once. Apart from these differences, a **ManualResetEvent** functions like an **AutoResetEvent**.

1. **What is CountDownEvent?**

The **CountDownEvent** is a synchronization primitive that unblocks its waiting threads after it has been signaled a certain number of times. It lets you wait on more than one thread. **CountDownEvent** is designed for scenarios in which you would otherwise have to use a **ManualResetEvent** or **ManualResetEventSlim** and manually decrement a variable before signaling the event. For example, in a fork/join scenario, you can just create a **CountDownEvent** that has a signal count of 5, and then start five work items on the thread pool and have each work item call Signal when it completes. Each call to Signal decrements the signal count by 1. On the main thread, the call to Wait will block until the signal count is zero.

In the side example, the **CountDownEvent** class is instantiated with count of 3. Whenever a Signal is called, the count is decremented. The Wait method blocks the threads until the count goes down to zero. So, even though the threads 1 and 2 gets completed, its only when the thread 3 gets completed and makes a Signal, then only all the threads are unblocked.



# 

# *WPF*

1. **What is Dispatcher?**

A ***dispatcher*** manages the work that takes place in a WPF application. The **dispatcher** owns the application thread and manages a queue of work items. As your application runs, the dispatcher accepts new work requests and executes one at a time.

Technically, a **dispatcher** is created the first time you instantiate a class that derives from **DispatcherObject** on a new thread. If you create separate threads and use them to show separate windows, you’ll wind up with more than one dispatcher. However, most applications keep things simple and stick to one user interface thread and one dispatcher. They then use multithreading to manage data operations and other background tasks.

Every thread have their dispatcher object. Rather than storing the dispatcher on Thread Local Storage (TLS), they are stored in a static list that contains all the available dispatcher objects. Whenever the dispatcher of an object is required, they go over the list, comparing the dispatcher’s thread property with the current thread, until they find the correct dispatcher object for this thread.

1. **What is DispatcherObject?**

Almost every WPF element has **thread affinity**. This means that access to such an element should be made only from the thread that created the element. In order to do so, every element, that requires thread affinity, is derived from DispatcherObject class. This class provides a property named Dispatcher that returns the Dispatcher object associated with the WPF element.

All the WPF controls inherits from the DispatcherObject class. It exposes just three members:

* **Dispatcher**: Returns the dispatcher that’s managing this object,
* **CheckAccess**(): Return true if the code is on the right thread to use the object; returns false otherwise and
* **VerifyAccess**(): Does nothing if the code is on the right thread to use the object; throws an *InvalidOperationException* otherwise.

1. **What is Dependency property?**

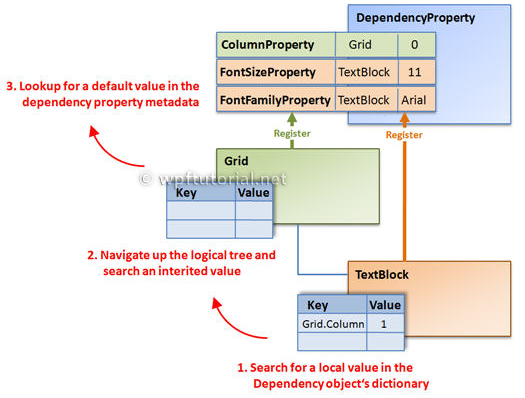
Windows Presentation Foundation (WPF) provides a set of services that can be used to extend the functionality of a common language runtime (CLR) property. Collectively, these services are typically referred to as the WPF property system. A property that is backed by the WPF property system is known as a **Dependency** **property**.

In WPF, properties are typically exposed as common language runtime (CLR) properties, also known as normal properties. However, internally these properties are implemented as a dependency property. The main difference is, that the value of a normal .NET property is read directly from a private member in your class, whereas the value of a DependencyProperty is resolved dynamically when calling the GetValue() method that is inherited from DependencyObject.

When you set a value of a dependency property it is not stored in a field of your object, but in a dictionary of keys and values provided by the base class DependencyObject. The key of an entry is the name of the property and the value is the value you want to set.

1. **How does Dependency property works?**

Each WPF control *registers a set of DependencyProperties to the static Dependency Property class*. Each of them consists of a key - that must be unique per type - and a metadata that contain callbacks and a default value.



All types that want to use DependencyProperties must derive from DependencyObject. This base-class defines a key, value dictionary that contains local values of dependency properties. The key of an entry is the key defined with the dependency property.

When you access a dependency property over its .NET property wrapper, it internally calls the GetValue (of DependencyProperty) to access the value. This method resolves the value by using a value resolution strategy that is explained in detail below. If a local value is available, it reads it directly from the dictionary. If no value is set, it goes up the logical tree and searches for an inherited value. If no value is found it takes the default value defined in the property metadata. This sequence is a bit simplified, but it shows the main concept.

1. **What are the advantages of Dependency property?**

The advantages of dependency properties are

* **Reduced memory footprint**: It's a huge dissipation to store a field for each property when you think that over 90% of the properties of a UI control typically stay at its initial values. Dependency properties solve these problems by only store modified properties in the instance. The default values are stored once within the dependency property.
* **Value inheritance**: When you access a dependency property the value is resolved by using a value resolution strategy. If no local value is set, the dependency property navigates up the logical tree until it finds a value. When you set the FontSize on the root element it applies to all textblocks below except you override the value.
* **Change notification**: Dependency properties have a built-in change notification mechanism. By registering a callback in the property metadata you get notified, when the value of the property has been changed. This is also used by the databinding.

The purpose of dependency properties is to provide a way to compute the value of a property based on the value of other inputs. These other inputs might include:

* System properties such as themes and user preference,
* Just-in-time property determination mechanisms such as data binding and animations/storyboards,
* Multiple-use templates such as resources and styles, or
* Values known through parent-child relationships with other elements in the element tree.

In addition, a dependency property can be implemented to provide *self-contained validation*, *default values*, *callbacks* that monitor changes to other properties, and a system that can *coerce* property values based on potentially runtime information. Derived classes can also change some specific characteristics of an existing property by overriding dependency property metadata, rather than overriding the actual implementation of existing properties or creating new properties.

1. **What are attached properties?**

An **Attached Property** is a concept defined by XAML. An attached property is intended to be used as a type of global property that is settable on any object. In WPF, attached properties are typically defined as a specialized form of dependency property that does not have the conventional property "wrapper".

Attached properties applies to several controls but they are defined in a different class. It is analogous to extension methods. The extension methods are defined outside the class boundaries but they are applicable to the object of the class. For example, the Grid.Row and Grid.Column properties are defined as attached properties in Grid class but these properties are applicable to the WPF controls declared inside the any Grid, like Button or textbox etc. In fact, these properties can be set on an element even if that element is not in a Grid, or even if there is not a single Grid object in the element tree.

1. **What are the type of controls available in WPF?**

There are three basic type of controls in WPF, Content Control, Item Control and Layout controls.

**Content Control** are the controls which are derived from the ***ContentControl*** Class and can contain a single nested element. This nested element can be of any type and can be set or retrieved in the code through the content property. The **Content** property is of type **Object**, so it can accept any object as content. Rendering of a content depends on the type of the object. For items that do not derive from **UIElement** class, the ToString method is called and the resulting string is rendered as the control content. The items that are derived from **UIElement** are displayed as contained within the content control. Examples of Content controls are **Button, Label, Checkbox,** and **RadioButton**.

There are some specialized controls which do not have Content property. They are specialized in terms of the content they display. For example **TextBox, TextBlock, Progress bar, Image, Slider**.

**Item controls**, also known as list-based controls, are designed to contain multiple child elements. Item controls are a familiar part of any user interface. Data is displayed frequently in item controls, and lists are used to allow the user to choose from a series of options. Item controls in WPF take the idea of lists one step further. Like content controls, item controls do not have restrictions on the kind of content they can present. Thus, an item control could present a list of strings or something more complex, such as a list of check box controls, or even a list that included various kinds of controls. Examples of Item controls are ListBox, ComboBox, TreeView, Menus, ToolBar, and StatusBar.

**Layout controls**, are controls which contain multiple nested controls of any type, provide built-in logic for the visual layout of those controls. Examples include **Grid, StackPanel, WrapPanel, DockPanel, and Canvas**.

1. **What are Binary Resources?**

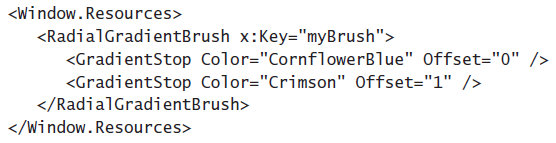
Resources are the non-code piece of an application or component. These are Fonts, Bitmaps, Audio/Video file, string tables etc. **WPF** supports two types of resources: **Binary** and **Logical** resources.

The **Binary resources** are the large binary files like Bitmaps, Audio/Video files etc. These are the traditional resources used in other .Net components. Binary resources can be packaged in three different ways: Embedded inside an assembly, as loose files that are known to the application at compile time and as loose files that might not be known to the application at compile time.

1. **What are Logical Resources?**

**Logical resources** are WPF feature which integrates closely with XAML. **Logical resources** enables to define objects in XAML that are not part of the **logical tree** but are available for use by WPF elements in the user interface. Elements in the UI can access these resources as needed. These logical resources are arbitrary .NET objects stored (and named) in an element’s Resources property, typically meant to be shared by multiple child elements. There are few advantages in defining the objects in the Resources section rather than defining the object every-time it is being used. First of them is the Re-usability because the object is defined only once rather than multiple times. Secondly, it offers flexibility by separating the objects used by the interface from the user interface itself. We can refactor the parts of the user interface without having to redesign it completely.

Any type of object can be defined as a resource. Every WPF element defines a **Resources collection**, which you can use to define objects available to that element and the elements in its visual tree. Although it is most common to define resources in the Resources collection of the window, you can define a resource in any element’s Resources collection and access it so long as the accessing element is part of the defining element’s visual tree.



A logical resource can be declared as given in the figure. Every object declared as a resource must set the [*x: key*] property. This is the name other child WPF elements will use to access the resource. The [x: Key] property does not have to be unique in the application, but it must be unique in the Resources collection in which it is defined.

1. **What are the differences between Static and Dynamic resources?**

The difference between the **DynamicResource** and **StaticResource** syntax lies in how the referencing elements retrieve the resources. Resources referenced by the **StaticResource** syntax are retrieved once by the referencing element and used for the lifetime of the resource. Resources referenced with the **DynamicResource** syntax are acquired every time the referenced object is used.

It might seem intuitive to think that, if you use **StaticResource** syntax, the referencing object does not reflect changes to the underlying resource, but this is not necessarily the case. *WPF objects that implement dependency properties automatically incorporate change notification, and changes made to the properties of the resource are picked up by any objects using that resource*. Take the following example:

This example renders the grid in the window with a blue background. If the Color property of the **SolidColorBrush** defined in the **Window.Resources** collection was changed in code to red, for instance, the background of the grid would render as red because change notification would notify all objects using that resource that the property had changed. The difference between static and dynamic resources comes when the underlying object changes. If Brush defined in the **Window.Resources** collection were accessed in code and set to a different object instance, the grid in the previous example would not detect this change.



However, if the grid used the following markup, the change of the object would be detected, and the grid would render the background with the new brush.



The downside of using **dynamic resources** is that they tend to decrease application performance because they are retrieved every time they are used, thus reducing the efficiency of an application. The best practice is to use static resources unless there is a specific reason for using a dynamic resource. Examples of instances in which you would want to use a dynamic resource include when you use the *SystemBrushes*, *SystemFonts*, and *SystemParameters* classes as resources or any other time when you expect the underlying object of the resource to change.

1. **What are Freezable object?**

A Freezable is a special type of object that has two states: unfrozen and frozen. When unfrozen, a Freezable appears to behave like any other object. When frozen, a Freezable can no longer be modified.

A Freezable provides a Changed event to notify observers of any modifications to the object. Freezing a Freezable can improve its performance, because it no longer needs to spend resources on change notifications. *A frozen Freezable can also be shared across threads, while an unfrozen Freezable cannot*.

Although the Freezable class has many applications, most Freezable objects in Windows Presentation Foundation (WPF) are related to the graphics sub-system. The Freezable class makes it easier to use certain graphics system objects and can help improve application performance. Most of the freezable classes contain un-managed resources. Whenever there is a change in the original object, the un-managed resources, contained in the object, are notified about the changes. This resource monitoring takes up sometime and thus degrades the performance. A freezable’s Freeze method enables to disable the Self-updating ability, thus improving the performance.

When a Freezable is freezed, all the feezable objects inside it freezes. A Freezable **can't** be frozen if any of the following are true:

* It has animated or data bound properties.
* It has properties set by a dynamic resource.
* It contains Freezable sub-objects that can't be frozen.

1. **What are resource dictionary?**

A resource dictionary is a collection of resources that reside in a separate XAML file and can be imported into the application. They can be useful for organizing the resources in a single place or for sharing resources between multiple projects in a single solution. A resource diction is simply a XAML document that does nothing but store the resources we want to use.

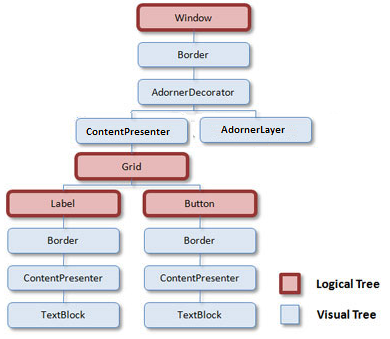
1. **What is Visual Tree and Logical tree?**

The **LogicalTree** represents the essential structure of an application UI. It closely matches the elements that are declared in XAML, and excludes most visual elements created internally to help render the elements that are declared. WPF uses the logical tree to determine several things including dependency property value inheritance, resource resolution, and more. A developer spends most of the time building the logical tree and then backing it up with event handling code.

A **VisualTree** is an expanded version of the logical tree. It breaks down elements into smaller pieces. In other words, instead of seeing a carefully encapsulated black box such as the Button control, you see the visual components of that button—the border that gives buttons their signature shaded background (represented by the ButtonChrome class), the container inside (a ContentPresenter), and the block that holds the button text (represented by the familiar TextBlock).

Elements of a WPF user interface are hierarchically related. This relation is called the **LogicalTree**. The template of one element consists of multiple visual elements. This tree is called the **VisualTree**. WPF differs between those two trees, because for some problems you only need the logical elements and for other problems you want all elements.

A WPF control consists of multiple, more primitive controls. A button - for example - consists of a border, a rectangle and a content presenter. These controls are visual children of the button.



When WPF renders the button, the element itself has no appearance, but it iterates through the visual tree and renders the visual children of it. This hierarchical relation can also be used to do hit-testing, layout etc.

But sometimes you are not interested in the borders and rectangles of a controls' template. Particularly because the template can be replaced, and so you should not relate on the visual tree structure! Because of that you want a more robust tree that only contains the "real" controls - and not all the template parts. And that is the eligibility for the logical tree.

The **logical** **tree** describes the relations between elements of the user interface. The logical tree is responsible for:

* Inherit *DependencyProperty* values
* Resolving *DynamicResource* references
* Looking up element names for bindings
* Forwarding *RoutedEvents*

The **visual tree** contains all logical elements including all visual elements of the template of each element. The visual tree is responsible for:

* Rendering visual elements
* Propagate element opacity
* Propagate Layout- and Render Transforms
* Propagate the *IsEnabled* property.
* Do Hit-Testing
* RelativeSource (find ancestor)

A visual tree has two usefulness:

* You can alter one of the elements in the visual tree using styles. You can select the specific element you want to modify using the *Style.TargetType* property. You can even use triggers to make changes automatically when control properties change. However, certain details are difficult or impossible to modify.
* You can create a new template for your control, called **ControlTemplate**. In this case, your control template will be used to build the visual tree exactly the way you want it.

1. **Are Content Elements part of VisualTree?**

Content elements (derived classes of *ContentElement*, like *ListBoxItem*) are not part of the visual tree; they do not inherit from Visual and have no visual representation. In order to appear in a UI at all, a ContentElement must be hosted within a content host that is a Visual, usually a FrameworkElement. You can conceptualize that the content host is somewhat like a "browser" for the content and chooses how to display that content within the screen region the host controls. Once the content is hosted, the content can be made a participant in certain tree processes that are normally associated with the visual tree.

Generally the FrameworkElement host class includes implementation code that adds any hosted *ContentElement* to the event route through sub-nodes of the content logical tree, even though the hosted content is not part of the true visual tree. This is necessary so that a *ContentElement* can source a routed event that routes to any element other than itself.

1. **What are Routed Events? Explain Bubbling and Tunneling.**

An **event** is a message sent by an object, such as a control or another part of the user interface, that the program responds to (or handles) by executing code. Although the traditional .NET event architecture is still present in WPF programming, WPF builds upon the event concept by introducing **routed** **events**.

A key concept to remember in event routing is the control containment hierarchy. In WPF user interfaces, controls frequently contain other controls. For example, a typical user interface might consist of a top-level Window object, which contains a *Grid* object, which itself might contain several controls, one of which could be a *ToolBar* control, which in turn contains several *Button* controls. *The routed event architecture allows for an event that originates in one control to be raised by another control in the containment hierarchy*. Thus, if the user clicks one of the Button controls on the toolbar, that event can be raised (handled) by the *button*, the *toolbar*, the *grid*, or the *window*.

Event routing gives you the flexibility to write tight, well-organized code that handles events in the most convenient place. It’s also a necessity for working with the WPF content model, which allows you to build simple elements (such as a button) out of dozens of distinct ingredients, each of which has its own independent set of events.

There are three types of routed events: direct, bubbling, and tunneling.

* **Direct Events**

Direct events are most similar to standard .NET events. Like a standard .NET event, a direct event is raised only by the control in which it originates. Because other controls in the control containment hierarchy do not raise these events, there is no opportunity for any other control to provide handlers for these events. An example of a direct event is the *MouseLeave* event.

* **Bubbling Events**

Bubbling events are raised first in the control where they originate and then by each control in that control’s control containment hierarchy, also known as a visual tree. The *MouseDown* event is an example of a bubbling event. Suppose you have a Label control contained inside a *FlowPanel* control contained inside a window. When the mouse button is pressed over the label, the first control to raise the *MouseDown* event would be Label. Then *FlowPanel* would raise the MouseDown event and then, finally, the window itself. You could provide an event handler at any or all stages of the event process.

* **Tunneling Events**

Tunneling events are the opposite of bubbling events. *A tunneling event is raised first by the topmost container in the visual tree and then down through each successive container until it is finally raised by the element in which it originates*. An example of a tunneling event is the *PreviewMouseDown* event. In the previous example, although the event originates with the Label control, the first control to raise the *PreviewMouseDown* event is Window, then FlowPanel, and then, finally, Label. Tunneling events enable you to intercept and handle events in the window or container before the event is raised by the specific control so you can filter input, such as keystrokes, at varying levels.

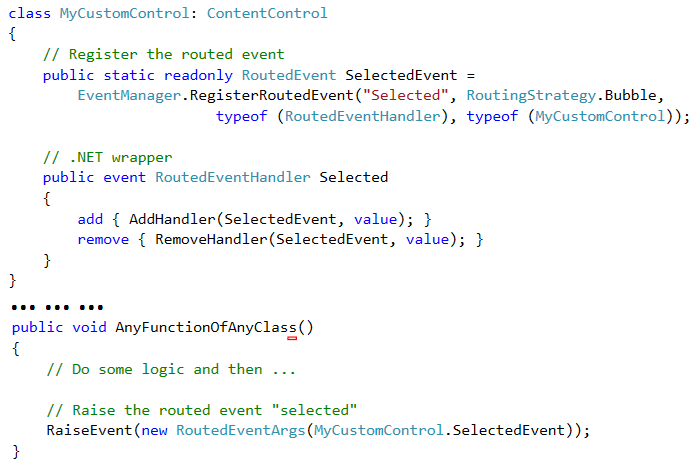
In the .NET Framework, all tunneling events begin with the word Preview, such as *PreviewKeyDown*, *PreviewMouseDown*, and so on, and are typically defined in pairs with a complementary bubbling event. For example, the tunneling event *PreviewKeyDown* is paired with the *KeyDown* bubbling event. The tunneling event always is raised before its corresponding bubbling event, thus allowing higher-level controls in the visual tree to handle the event. *Each tunneling event shares its instance of event arguments with its paired bubbling event*.

All routed events include an instance of **RoutedEventArgs**. The **Handled** property of **RoutedEventArgs** indicates whether the Event has been handled. By setting the **Handled** property to True, we can halt the further event **Bubbling** and **Tunneling**.

1. **What is EventManager class?**

The **EventManager** is a static class that manages the registration of all WPF routed events. While dependency properties are registered with the **DependencyProperty.Register** method, routed events are registered with the **EventManager.RegisterRoutedEvent** method.

Below code creates a Custom control. The custom control has a **RoutedEvent**, which is first registered, using **EventManager** and then exposed as a normal .Net Property. Later on, the event is raised in any function of any class after some logic.



1. **What are Styles? Explain its few properties.**

Styles can be thought of as analogous to cascading style sheets as used in Hypertext Markup Language (HTML) pages. Styles basically tell the presentation layer to substitute a new visual appearance for the standard one. They enable you to make changes to the user interface as a whole easily and to provide a consistent appearance and behavior for your application in a variety of situations. Styles enable you to set properties and hook up events on UI elements through the application of those styles. Further, you can create visual elements that respond dynamically to property changes through the application of triggers, which listen for a property change and then apply style changes in response.

Style has following important properties

* **BasedOn** - Indicates another style that this style is based on. This property is useful for creating inherited styles.
* **Resources** - Contains a collection of local resources the style uses.
* **Setters** – Contains a collection of Setter or EventSetter. These are used to set properties or events on an element as a part of a style.
* **TargetType** - Identifies the intended element type for the style
* **Triggers** - Contains a collection of Trigger objects and related objects that enable you to designate a change in the user interface in response to changes in properties.

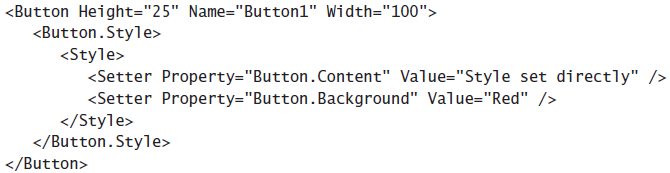
1. **How a Style is created and applied?**

Style can be applied to elements in various ways. Below are few examples that show how to apply styles.

Setting the Style property directly -

The most straightforward way to apply a style to an element is to set the Style property directly in XAML. The following example demonstrates directly setting the Style property of a Button element:

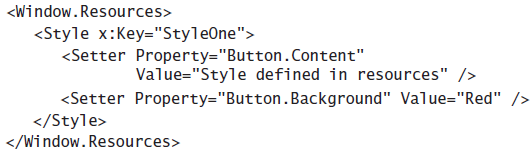
Although setting the style directly in an element might be the most straightforward, it is seldom the best method. When setting the style directly, you must set it for each element you want to be affected. In most cases, it is simpler to set the properties of the element directly at design time.



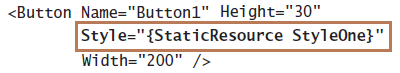
One scenario in which you might want to set the style directly in an element is to provide a set of triggers for that element. Because triggers must be set in a style (except for **EventTrigger**), you could conceivably set the style directly to set triggers for an element.

Setting the Style in a resource collection -

The most common method for setting styles is to create the style as a member of a Resources collection and then apply the style to elements in your user interface by referencing the resource. The side example demonstrates creating a style as part of the **Window.Resources** collection.



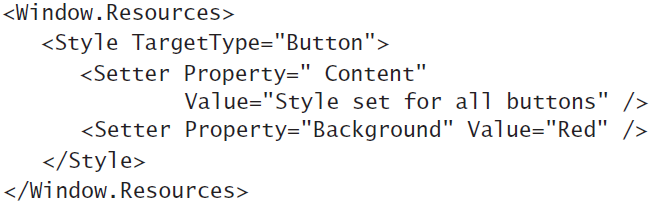
Under most circumstances, you must supply a key value for a style that you define in the Resources collection. Then you can apply that style to an element by referencing the resource, as shown in bold, in side example.



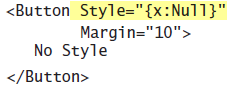
The advantage of defining a style in the resources section is that you can then apply that style to multiple elements by simply referencing the resource.

Applying style to all controls of specific type -

You can use the TargetType property to specify a type of element to be associated with the style. When you set the TargetType property on a style, that style is applied to all elements of that type automatically. Furthermore, you do not need to specify the qualifying type name in the ***Property*** of any setters you use; you can just refer to the property name. When you specify TargetType for a style you have defined in a Resources collection, you do not need to provide a key value for that style. The side example demonstrates the use of the TargetType property.



If you want an individual element to opt out of the style, you can set the style on that element explicitly, as seen in side example. This example explicitly sets the style to Null, which causes the button to revert to its default look. You also can set the style to another style directly, as seen earlier in this lesson.



1. **What are Triggers?**

**Triggers** enable you to implement property changes declaratively in response to other property changes that would have required event-handling code in Windows Forms programming. Using **Triggers**, you can automate simple style changes that would ordinarily require boilerplate event-handling logic. Triggers are linked to styles through the **Style.Triggers** collection. Every style can have an unlimited number of triggers, and each trigger is an instance of a class that derives from **System.Windows.TriggerBase**. A Trigger can be applied directly to the elements, without needing to create a style, by using the **FrameworkElement.Trigger** collection. However, there is a constraint that this trigger collection only supports event triggers.

There are five types of trigger. These are:

* **Trigger-** This is the simplest form of trigger. It watches for a change in a dependency property and then uses a setter to change the style.
* **MultiTrigger-** This is similar to trigger but combines multiple conditions. All the conditions must be met before the trigger springs into action.
* **DataTrigger-** This trigger works with data binding. It’s similar to Triggers, except it watches for a change in any bound data. Monitors a bound property and activates when the value of the bound property matches the Value property
* **MultiDataTrigger-** Monitors multiple bound properties and activates only when all the monitored bound properties match their corresponding Value properties
* **EventTrigger-** This is the most sophisticated trigger. It applies an animation when an event occurs**.**

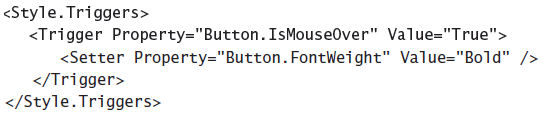
1. **What are Property Trigger and MultiTrigger?**

A **Trigger** or **Property Trigger** monitors the value of a DependencyProperty specified by the Property. When the value of the specified property equals the Value property, the trigger is activated.

Few important properties of Property trigger are:

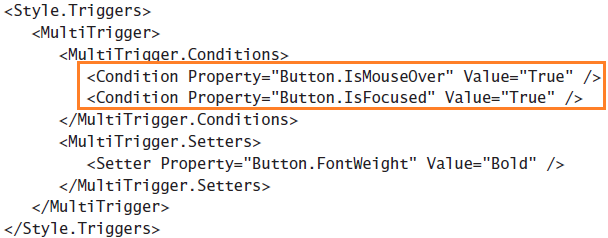
* **Property**: Indicates the property that is monitored for change
* **Setters**: Contains a collection of Setter objects that are applied when the trigger becomes active
* **Value**: Indicates the value that is compared to the property referenced by the Property.

Triggers listen to the property indicated by the **Property** and compare that property to the **Value** property. When the referenced property and the Value property are equal, the trigger is activated. Any Setter objects in the Setters collection of the trigger are applied to the style. When the referenced property no longer matches the Value property, the trigger is inactivated. All Setter objects in the Setters collection of the trigger are inactivated.



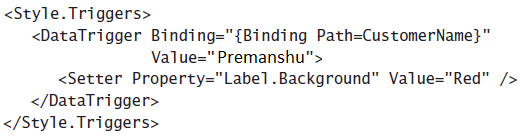
The side example, demonstrates a simple Trigger object that changes the *FontWeight* value of a Button element to Bold when the mouse enters the button.

**MultiTrigger** are similar to **Property Trigger** in that they monitor the value of properties and activate when those properties meet a specified value. The difference is that **MultiTrigger** are capable of monitoring several properties at a time, and they activate only when all monitored properties equal their corresponding Value properties. The properties that are monitored, and their corresponding Value properties, are defined by a collection of Condition objects. The side example demonstrates a MultiTrigger property that sets the Button. *FontWeight* property to Bold only when the button is focused and the mouse has entered the control.



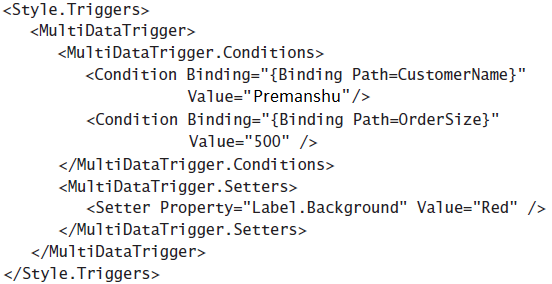
1. **What are Data Trigger and Multi-Data Trigger?**

Data triggers are similar to property triggers in that they monitor a property and activate when the property meets a specified value, but they differ in that the property they monitor is a bound property. Instead of a *Property* property, data triggers expose a Binding property that indicates the bound property to listen to. The following shows a data trigger that changes the Background property of a label to red when the bound property *CustomerName* equals “Premanshu”.



Data triggers are used for properties that are not necessarily dependency properties. They work by creating a binding to a regular property, which is then monitored for changes. This also opens up for binding your trigger to a property on a different control.

Multi-data triggers are to data triggers as multi-triggers are to property triggers. They contain a collection of Condition objects, each of which specifies a bound property through its Binding property, and a value to compare to that bound property. When all the conditions are satisfied, the multi-data trigger activates. The following example demonstrates a multi-data trigger that sets the Label.Background property to red when CustomerName equals “Premanshu” and OrderSize equals 500.



1. **What are Data Templates?**

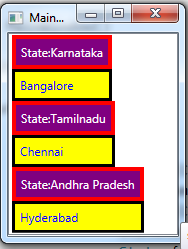
A **data** template is a bit of XAML that describes how bound data is displayed. A **data template** can contain elements that are each bound to a data property, along with additional markup that describes layout, color, and other aspects of appearance.

When binding a property or list directly to a control, you are limited to binding a single property. With data templates, however, you can bind more than one property in each item, thereby displaying multiple bits of related data together.

The following example demonstrates this concept. There are two labels, one for the State’s Name, and another for its Capital. Additional properties are set to provide style, and both labels are placed within a StackPanel class to facilitate layout.

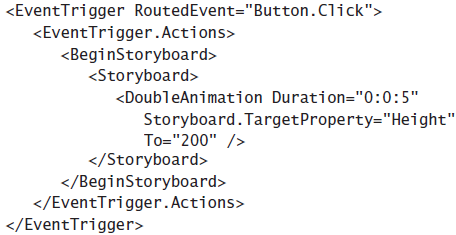






1. **What is an Event Trigger?**

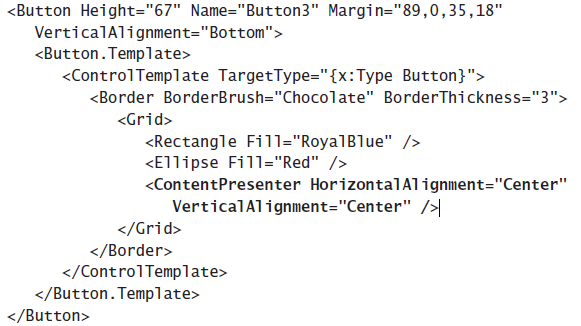
While an ordinary trigger waits for a property change to occur, an event trigger waits for a specific event to be fired. Event triggers are different from the other trigger types. Whereas other trigger types monitor the value of a property and compare it to an indicated value, event triggers specify an event and activate when that event is raised. In addition, event triggers do not have a Setters collection; rather, they have an Actions collection. Most actions deal with animations. The side example demonstrates a simple animation that causes the button to grow in height by 200 units when clicked.



1. **What are Control Template?**

WPF controls are designed to be *lookless*, which means that an element’s functionality is completely separate from the element’s appearance, and its appearance can be completely redefined. Consequently, it is easy to provide a new UI appearance for a WPF element by creating a new **control template**. A **control template** is an XAML document that describes how a control will appear in the presentation layer. The template represents the visual tree of the control; it defines any of the parts that make up a control as well as the appearance and behavior of those parts.

Because WPF elements are designed to be *lookless*, the visual layer is distinct from the logical layer, and you can change the appearance of any WPF element radically by creating a new template for that element.

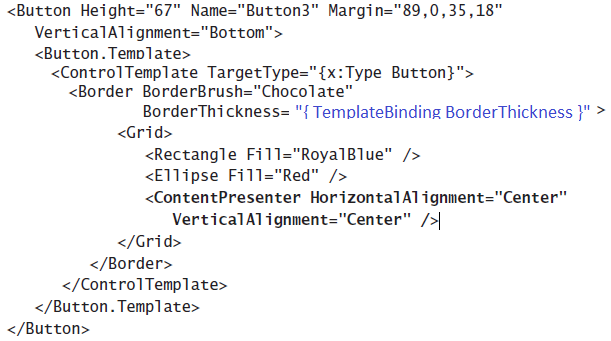


For example in the accompanied XAML code, the Button has a Template section which conatins a **Control Template**. It defines a *Border* element. The **Border** element is a content control so it can contain only one child element. To overcome this contraint, a **Grid** element is added to the **Border** element to include more elements inside the Border. When the control is rendered, a Chocolate bordered Button with RoyalBlue rectangle and Red Ellipse is displayed.

The **ContentPresenter** is essentially a placeholder for the Content property of the element. Whatever the value of the Content property, it is inserted into the space occupied by the ContentPresenter. For List controls, a similar class—**ItemsPresenter**—presents the objects in the Items property. If **ContentPresenter** is not included, then there will be no way to display the value of **Content** property. In order for the ContentPresenter to work, the tag “**x: Type Button**” is added to the **ControlTemplate** markup., otherwise it can be omitted.

1. **What is TemplateBinding?**

The **TemplateBinding** markup is essentially a data-binding expression that binds a property within the template to a named property of the templated parent, which is a fancy way of saying the control for which you are designing the template. The **TemplateBinding** expression takes the following form:



{TemplateBinding <PropertyName>}

where <PropertyName> is the name of the property on the templated parent to which you want to bind.

1. **What are the difference between Style and Control Template?**

Templates and styles define the pieces that make up a control and the default behavior of the control, respectively. You create templates and styles by making copies of the default system styles and templates for a control.

By using **styles**, you can modify the default values of properties that are set on the control to which the style is applied. For example, you can specify default colors for the background, border, and foreground of a control such as a button. Using **templates**, you can modify the structure of the control to which the template is applied. You can modify a control template to rearrange, add, or delete the elements (or parts) in the control. For example, you can add a background image or design to a control such as a button.

1. **What is Data Binding?**

**Data binding** is the mechanism by which information contained in an object is associated with elements in the user interface. The most common scenario is seen when binding a control in the user interface to a record in a database. When a field in the database is bound to a control, that field’s value is displayed in the control. Changes made to the value in the control can then be reflected back to the original record in the database if the programmer so desires. In WPF any property can be bound to any other object or source. WPF makes it easy to bind almost any property on any element to any other object, property, collection, or data source.

The term **data binding** describes the process of creating a dependence for the value of one property, called the ***target*** property, on the value of another property, called the ***source*** property. The *target* property takes on the value of the *source* property. Many variations on the style of binding are possible. In some cases, changes to the source property are transmitted to the target immediately, but in other cases, they are not. In some cases, changes the user makes to the target property also are transmitted back to the value of the source property, although in other cases, they are not. The class that makes all this possible is the **Binding** class.

1. **Explain the Binding class**

The Binding class establishes and describes a relationship between the target and source properties. You can use a Binding object to create a relationship between a target property and another object, a list, an ADO.NET data object, or any other object.

Below are the few important properties of Binding class.

* **ElementName-** Get or sets the name of the element to use as the Binding source object. When binding to a   
  WPF element, this property is used in instead of the Source property.
* **FallbackValue**- Get or set the value to use when a binding is unable to return a value.
* **Mode**- Gets or sets a value that determines the direction of the data flow in the binding.
* **NotifyOnSourceUpdated**- Gets or sets a value that indicates whether to raise the SourceUpdated event when a value is transferred from the target to the source.
* **NotifyOnTargetUpdated**- Gets or sets a value that indicates whether to raise the SourceUpdated event when a value is transferred from the source to the target.
* **Path**- Gets or sets the path to the source property of the binding source object.
* **RelativeSource**- Gets or sets the binding source by specifying its location relative to the position of the binding target. When you want to specify an element at a relative position in the visual tree, this property is used instead of the ElementName property or the Source property to specify the source object.
* **Source**- Gets or sets the object to use as the binding source. When not binding to a WPF element, this property is used instead of the *ElementName* property.
* **TargetNullValue**- Gets or sets the value used in the target when the value of the source is null.
* **XPath**- Gets or sets an XPath query that returns the value on the *Extensible Markup Language* (*XML*) binding source to use.

At the core of a data binding are two properties: the source property and the target property. Source Property is made up of two components:

* ElementName or Source or RelativeSource Property- specifies the source object to which the binding object is bound, and
* Path Property- Specifies to which property of the object the binding is bound.

The target property is the property to which the binding is set. The binding automatically transmits the value retrieved from the source property to the target property.

1. **What is DataContext?**

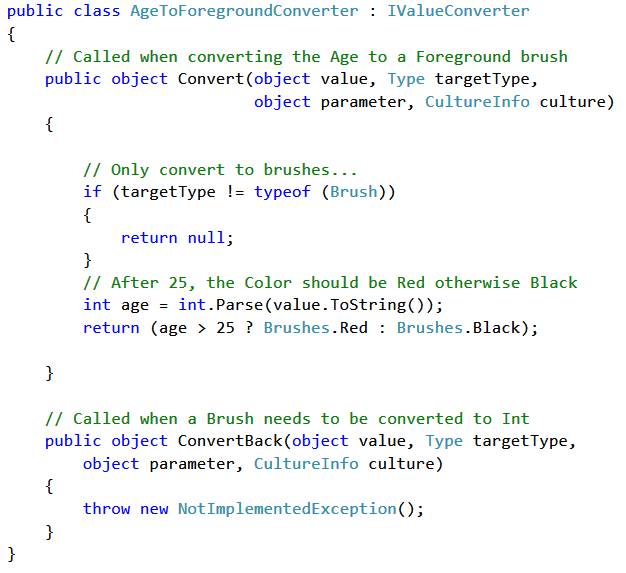
If you don’t specify a source using the Source or RelativeSource property, WPF searches up the element tree starting at the current element. It examines the DataContext property of each element and uses the first one that isn’t null. The DataContext property is extremely useful if you need to bind several properties of the same object to different elements, because you can set the DataContext property of a higher-level container object rather than directly on the target element.

In [DataTemplate example](#What_is_DataTemplate), the DataContext property of the MainWindow class is set to the MainWindow object it self. This way, the entire visual tree, that come under the MainWindow, can access the public properties of MainWindow, similar to the way the ItemSource property of ListBox is binded to one of the public property of MainWindow.

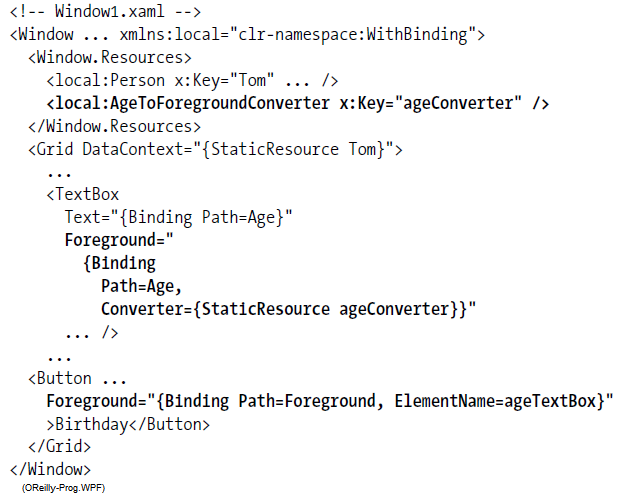
1. **What is Converter?**

In order to bind two incompatible types of properties, a **ValueConverter** or Converter is needed that converts the value from the source to target type and back. A **ValueConverter** is a class, that implements the simple interface **ValueConverter** with the two methods object **Convert**(object value) and object **ConvertBack**(object value).

The side example shows how to declare a converter. The class needs to implement the **IValueConverter** interface. The interface exposes two methods, **Convert** and **ConvertBack**, which does the job of converting from one type to another. In this example, the Integer value is converted to a Brush object, depending on some business logic. The **ConvertBack** can be implemented if there is a business requirement to convert a brush object to integer (rare).



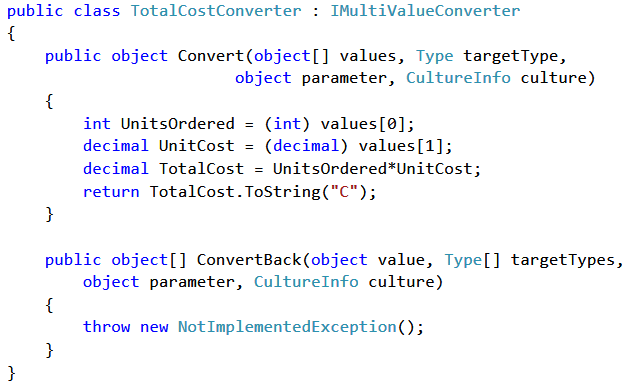
To use the **Converter**, the instance of the Converter is created in the **XAML** and assigned a name. This establishes it as a converter between *Age* and *Foreground* brush. This is done by setting the **Converter** property of the Binding object. Here, the Foreground property of **TextBox** is Binded to the *Age* property of “Tom” and it is Converted to a ***Brush*** (Red or Black) depending on the Age value.



1. **What is Multi-Converter?**

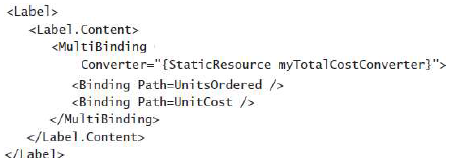
**Multi**-**value** **converters** enable you to return a converted value that results from multiple fields. For example, you might perform a calculation, such as multiplying a UnitsPurchased field by a UnitPrice field, to return a value representing total cost. You can also use a multivalue converter to provide complicated conditional formatting that takes multiple fields into account.

To create a **multi-value converter**, you must implement the **IMultiValueConverter** interface. This interface is very similar to the **IValueConverter** interface.



The only differences are that the Convert method takes an Object array instead of a single Object as the first parameter, and the ConvertBack method returns an Object array and takes a Type array as one of the parameters.

To bind a property to a value returned by a multi-value converter, you must create a reference to the assembly that contains the **multi-value converter** and then add an instance to an available resource collection. To actually create the binding, however, you must use a class called **MultiBinding**. The **MultiBinding** class is essentially a collection of bindings that specifies a converter that takes the value of those bindings into account. You can set a property to a MultiBinding object in the same manner that you would set it to a **Binding** object. The following example demonstrates setting the Content property of a Label element to the value returned by an instance of the *TotalCostConverter* class used in the previous example. This example assumes that you have created an instance of this converter in the Window.Resources collection with a key value of *myTotalCostConverter* and that the *DataContext* property of Window has been set to an object with properties named Units-Ordered and UnitCost.



1. **What is difference between DataTrigger and Converter?**

**Converters** are for converting one data type into another, not for determining the conditional values, whereas DataTriggers are strictly conditional. They can set different properties of different controls when a particular binded property meets a certain value. There are some cases where **DataTrigger** and **Converter** can be replaced with each other, however there are performance overhead in using the Converter.

Converter supports Test Driven Development (TDD). However, if there is no complex business logic or custome logic then Triggers should be used.

One limitation of Triggers is that Setters in the DataTriggers can only change properties of the UI elements. The ViewModel properties cannot be updated with DataTriggers. Converters has ConvertBack method which performs the twoway actions.

1. **What is Command?**

**Commands** are a means by which shared actions can be grouped and invoked in several different ways within an application. In an application, functionality is divided into higher-level tasks. These tasks may be triggered by variety of different actions and through a variety of different user interface elements like menus, context menus, keyboard shortcuts toolbars etc. WPF allows to define these tasks, also known as Commands, and connect controls to them, so that the event handler code is not repeated. The advantages of Commands are:

* No code redundancy. The command handler is written at just once place.
* Support for enabling/disabling all the command linked UI controls, based on the status of the command, and
* A central place to store the text caption for the command

One of the purpose of commands is to indicate whether an action is available. A command can indicate whether an action is possible by implementing the **CanExecute** method. A button can subscribe to the **CanExecuteChanged** event and be disabled if **CanExecute** returns false or be enabled if **CanExecute** returns true.

1. **What is WPF Command Model?**

The WPF command model consist of four key ingredients.

* **Commands**. It is the action to be executed. A command represents an application task and keeps track of whether it can be executed. However, commands don’t actually contain the code that performs the application task.
* **Command** **bindings**. It is the object which maps the command logic to the command. Each command binding links a command to the related application logic, for a particular area of your user interface. This factored design is important, because a single command might be used in several places in your application and have a different significance in each place. To handle this, you use the same command with different command bindings.
* **Command** **sources**. It is the object that invokes the command. A command source triggers a command. For example, a MenuItem and a Button can both be command sources. Clicking them executes the bound command.
* **Command** **targets**. It is the object that the command is being executed on. A command target is the element on which the command is being performed. For example, a Paste command might insert text into a **TextBox** and an OpenFile command might pop a document into a *DocumentViewer*. The target mayor may not be important, depending on the nature of the command.

For example, in a typical windows application, the Paste command is a Command, the MenuItem is a Command Source, the TextBox is a command target and the command binding would be supplied by the TextBox control. It is worth noting that it is not always the case that the command binding is supplied by the control that is the command target class. Quite often, the command binding must be created by the application developer or the command binding might be attached to an ancestor of the command target.

1. **What is ICommandSource?**

Commands can be hooked only the controls that implements the **ICommandSource** interface. These controls are the ones which derive from **ButtonBase** (**Button**, **CheckBox**, **RadioButton** and so on), Individual **ListBoxItem** objects, the **Hyperlink** and the **MenuItem**.

The **ICommandSource** interface defines three properties:

* **Command**: Points to the linked command. This is the only required detail.
* **CommandParameter**: Supplies any other data you want to send with the command.
* **CommandTarget**: Identifies the element on which the command is being performed.

1. **What is UI Element and Framework Element?**
2. **What is Property Element and Object Element?**
3. **What is DelegateCommand?**
4. **What is Command Object?**

(https://msdn.microsoft.com/en-us/library/ms788723(v=vs.110).aspx)

<http://kentb.blogspot.in/2009/04/mvvm-infrastructure-delegatecommand.html>

<https://msdn.microsoft.com/en-us/library/ff921126.aspx>

<https://msdn.microsoft.com/en-us/magazine/dd419663.aspx#id0090030>

<http://stackoverflow.com/questions/14180688/difference-between-delegatecommand-relaycommand-and-routedcommand>

# 

# *MVC*

1. **What is Worker Process?**

A ***worker process*** (**w3wp**) is a program that executes the ASP.Net Application in IIS. All the ASP.Net functionality runs under the scope of ***worker process***.  When a request comes to the server from a client, the ***worker process*** is responsible to generate the request and response. The worker process is the heart of ASP.NET Web Application which runs on IIS.

1. **What is Application Pool?**

A web server can have many web sites that runs together. Each of the web sites is known as app domain. Each app domain or web site is assigned to one Application pool (AppPool). An AppPool can be assigned to multiple web sites. In that case, the web sites share the same resources and settings. A change in the AppPool affects as the web sites assigned to it. Application pools enables a better security, reliability, and availability for any web application.

An AppPool is also a container to one or more worker process. An AppPool with multiple worker process is known as Web Garden. The Worker process serves as a process boundary that separates the AppPools. So, in case of crash, the other AppPools remain un-affected. The Worker Processes do not communicate directly with each other, they do so by means of files or database.

In IIS 7, there are two modes of application pool

* **Integrated application pool mode**: When an application pool is in Integrated mode, you can take advantage of the integrated request-processing architecture of IIS and ASP.NET. When a worker process in an application pool receives a request, the request passes through an ordered list of events. Each event calls the necessary native and managed modules to process portions of the request and to generate the response.

There are several benefits to running application pools in Integrated mode. First the request-processing models of IIS and ASP.NET are integrated into a unified process model. This model eliminates steps that were previously duplicated in IIS and ASP.NET, such as authentication. Additionally, Integrated mode enables the availability of managed features to all content types.

* **Classic application pool mode**: When an application pool is in Classic mode, IIS 7 and above handles requests in the same way as in IIS 6.0 worker process isolation mode. ASP.NET requests first go through native processing steps in IIS and are then routed to Aspnet\_isapi.dll for processing of managed code in the managed runtime. Finally, the request is routed back through IIS to send the response.

This separation of the IIS and ASP.NET request-processing models results in duplication of some processing steps, such as authentication and authorization. Additionally, managed code features, such as Forms authentication, are only available to ASP.NET applications or applications for which you have script mapped all requests to be handled by aspnet\_isapi.dll.

1. **What is WWW Service?**

The *World Wide Web publishing service* (**W3SVC**) or *WWW* Service manages the HTTP protocol and HTTP performance counters. The *W3SVC* and *Windows process Activation Service* (**WAS**) run as local system in the same Svchost.exe process and share same binaries.

In IIS, the WWW service no longer manages worker processes. Instead, the WWW Service is the listener adapter for the HTTP listener, HTTP.sys. As the listener adapter, the WWW Service is primarily responsible for configuring HTTP.sys, updating HTTP.sys when configuration changes, and notifying WAS when a request enters the request queue.

The WWW service manages the following areas:

* HTTP administration and configuration: The WWW Service reads configuration information from the IIS metabase and uses that information to configure and update the HTTP listener, HTTP.sys. In addition, WWW service starts, stops, monitors, and manages worker processes that process HTTP requests.
* Process management: The WWW Service manages application pools and worker processes, such as starting, stopping, and recycling worker processes. Additionally, the WWW Service monitors the health of the worker processes, and invokes rapid fail detection to stop new processes from starting when several worker processes fail in a configurable amount of time.
* Performance monitoring: The WWW Service monitors performance and provides performance counters for Web sites and for the IIS cache.

1. **What is Windows Process Activation Service (WAS)?**

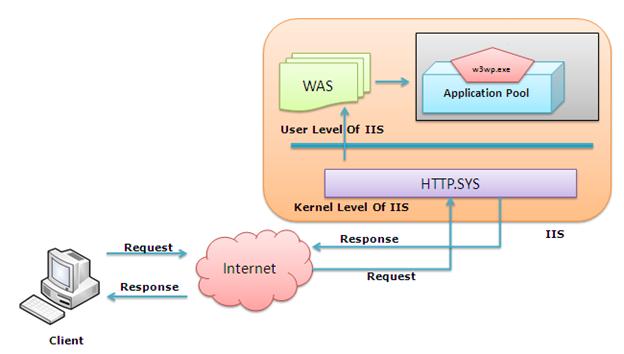
The Windows Process Activation Service (**WAS**) manages application pool configuration and the creation and lifetime of worker processes for HTTP and other protocols. The World Wide Web Publishing Service (W3SVC) and other services depend on WAS. WAS can run without WWW Service, if HTTP functionality is not required.

On startup, **WAS** reads certain information from the ApplicationHost.config file, and passes that information to listener adapters on the server. Listener adapters are components that establish communication between WAS and protocol listeners, such as **HTTP.sys**. Once listener adapters receive configuration information, they configure their related protocol listeners and prepare the listeners to listen for requests.

The following list describes the type of information that WAS reads from configuration:

* Global configuration information
* Protocol configuration information for both HTTP and non-HTTP protocols
* Application pool configuration, such as the process account information
* Site configuration, such as bindings and applications
* Application configuration, such as the enabled protocols and the application pools to which the applications belong

**WAS** manages application pools and worker processes for both HTTP and non-HTTP requests. When a protocol listener (**Http.Sys**) picks up a client request, **WAS** determines if a worker process is running or not. If an application pool already has a worker process that is servicing requests, the listener adapter passes the request onto the worker process for processing. If there is no worker process in the application pool, WAS will start a worker process so that the listener adapter can pass the request to it for processing.



The **WAS** takes the request from **Http**.**Sys** and pass it to respective application pool. When Application pool receives the request, it passes the request to worker process (**w3wp**.**exe**). The worker process “**w3wp**.**exe**” looks up the URL of the request in order to load the correct ISAPI extension. ISAPI extensions are the IIS way to handle requests for different resources. Once ASP.NET is installed, it installs its own ISAPI extension (**aspnet\_isapi.dll**) and adds the mapping into IIS.

1. **What is Http.Sys?**

The IIS architecture has two layers: **Kernel Mode** and **User Mode**. Kernel mode contains the Http listener called **Http** **protocol** **stack** or **Http.Sys**. The **HTTP.sys** listens for HTTP requests from the network, passes the requests onto IIS for processing, and then returns processed responses to client browsers. Whenever a request comes from Client to Server, it hits **Http.Sys** First.

The **Http.Sys** provides the following benefits:

* Kernel-mode caching. Requests for cached responses are served without switching to user mode.
* Kernel-mode request queuing. Requests cause less overhead in context switching because the kernel forwards requests directly to the correct worker process. If no worker process is available to accept a request, the kernel-mode request queue holds the request until a worker process picks it up.
* Request pre-processing and security filtering.

1. **What are HTTP Modules?**

**HTTP Modules** are classes that have access to the incoming request. We can also create our own HTTP Module if we need to handle anything during upcoming request and response. The **Http Modules** are the part of Native Modules that are available with a full installation of IIS 7. The other Native modules includes Security modules, Content Modules, Logging and Diagnostic modules etc.

**Http Modules** perform tasks specific to **hypertext transfer protocol (Http)** in the request processing pipeline. **HTTP modules** include modules to respond to information and inquiries sent in client headers, to return HTTP errors, to redirect requests, and more.

1. **What are HTTP Handlers?**

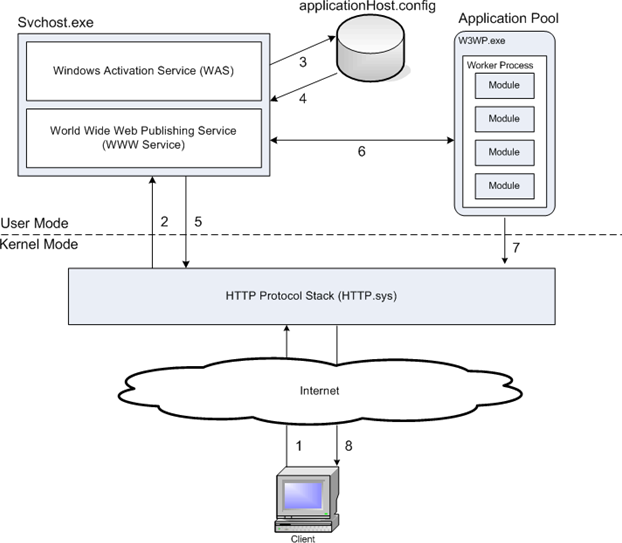
An ASP.NET **HTTP handler** is the process (frequently referred to as the "endpoint") that runs in response to a request made to an ASP.NET Web application. The most common handler is an ASP.NET page handler that processes .aspx files. When users request an .aspx file, the request is processed by the page through the page handler. You can create your own **Http** **handlers** that render custom output to the browser.

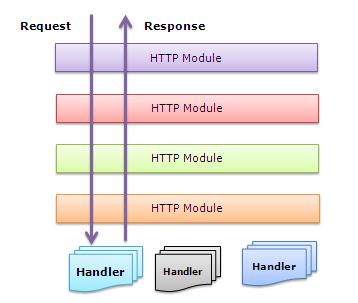
An **Http Handler** is a class that implements the System.Web.IHttpHandler interface. The ASP.NET **Http Handlers** are responsible for intercepting requests made to your ASP.NET web application server. They run as processes in response to a request made to the ASP.NET Site.

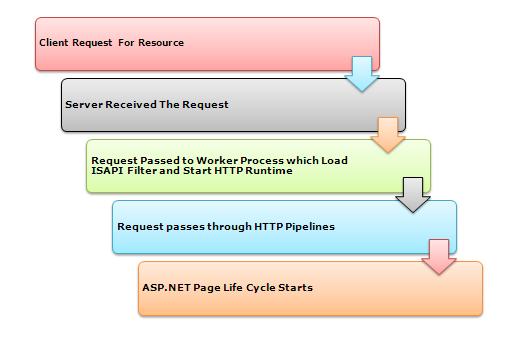
1. **How IIS processes an HTTP request?**

The following are the steps involved when an Http request is made:

1. When a client browser initiates an HTTP request for a resource on the Web server, **HTTP.sys** intercepts the request.
2. **HTTP.sys** contacts WAS to obtain information from the configuration store.
3. WAS requests configuration information from the configuration store, ApplicationHost.config.
4. The WWW Service receives configuration information, such as application pool and site configuration.
5. The WWW Service uses the configuration information to configure HTTP.sys.
6. WAS starts a worker process, **w3wp.exe**, for the application pool to which the request was made.
7. The worker process, *w3wp.exe*, looks up the URL of the request in order to load the correct ISAPI extension. Generally it is aspnet\_isapi.dll.
8. When Worker process loads the aspnet\_isapi.dll, it starts an HttpRuntime, which is the entry point of an application. HttpRuntime is a class which calls the *ProcessRequest* method to start Processing.
9. When this methods called, a new instance of HTTPContext is created, which is accessible using HTTPContext.Current properties. This object still remains alive during life time of object request.  Using HTTPContext.Current we can access some other objects like Request, Response, Session etc.
10. After that HttpRuntime loads an HttpApplication object with the help of HttpApplicationFactory class. Each and every request should pass through a set of HttpModules, called HttpPipeline, to reach to HttpHandler. This list of module are configured by the HttpApplication.
11. The HttpHandler generates the output for the requested resource. So, when we requesting for any aspx web pages,   it returns the corresponding HTML output.
12. The client receives a response.







1. **How is the ASP.Net routing different from MVC routing?**

In normal ASP.Net application, when a page request is received, there is a page on the disk that corresponds to the requested page. If the page exists then the response is given back to the browser otherwise it throws an error “Page not found”. This is because every ASP.Net page implements the IHttpHandler interface. The interface method ProcessRequest () gets called when the page is requested. The ProcessRequest () method is responsible for generating the content that gets sent back to the browser.

In ASP.Net MVC, there is no page on the disk that corresponds to the request. Instead, the request is routed to a special class called Controller. The controller is responsible for generating the content that gets sent back to the browser. Unlike ASP.Net, where there is always a one to one mapping between the request and the pages

1. **What is URL Routing Module?**

The **UrlRoutingModule** class matches an HTTP request to a route in an ASP.NET application. The module iterates through all the routes in the *RouteCollection* property and searches for a route that has a URL pattern that matches the format of the HTTP request. When the module finds a matching route, it retrieves the **IRouteHandler** object for that route. From the route handler, the module gets an **IHttpHandler** object and uses that as the HTTP handler for the current request.

1. **What is MVC page life cycle?**

Below are the detailed steps that occurs when the request arrives to IIS.

1. IIS determines that request needs to be processed by ASP.Net.
2. ***URLRouting*** Module gets a chance to act on the request as any standard HTTP Module.
3. ***URLRouting*** module checks if the request path matches the routes configured with the application.
4. ***URLRouting*** Module gets the ***IHttpHandler*** from the corresponding route’s ***IRouteHandler***. ***MVCRouteHandler*** is the default route handler for ASP.Net MVC.
5. In response to ***GetHTTPHandler***, MVCRouteHandler returns ***MVCHandler*** which implements ***IHttpHandler***.
6. ***IRouteHandler*** executes ***ProcessRequest*** method of MVCHandler passing the current ***HTTPContext***.
7. MVCHandler uses ***IControllerFactory*** to obtain an instance of IController using the controller name in the request path. ***DefaultControllerFactory*** is the default ***IControllerFactory*** in ASP.Net MVC.
8. MVCHandler invokes ***execute*** method on the controller.
9. Controller executes the method corresponding to the action specified in the request path.
10. Action method can optionally add data to the ***viewdata*** dictionary and returns the instance of type ***ActionResult***.
11. ***ExecuteResult*** method of ***ActionResult*** is executed, passing the ***ControllerContext***.
12. ***ViewResult*** locates the corresponding view using configured view engine. ***WebFormViewEngine*** is the default view engine for ASP.Net MVC.
13. ***ViewResult*** invokes Render method on IView.

**The shorter version**:

Any web application has two main execution steps first understanding the request and depending on the type of the request sending out appropriate response. MVC application life cycle is not different it has two main phases first creating the request object and second sending our response to the browser.

**Creating the request object: -**The request object creation has four major steps. Below is the detail explanation of the same.

**Step 1 Fill route:** MVC requests are mapped to route tables which in turn specify which controller and action to be invoked. So if the request is the first request, the first thing is to fill the route table with routes collection. This filling of route table happens in the ***global.asax*** file.

**Step 2 Fetch route:** Depending on the URL sent ***UrlRoutingModule*** searches the route table to create ***RouteData*** object which has the details of which controller and action to invoke.

**Step 3 Request context created:** The ***RouteData*** object is used to create the ***RequestContext*** object.

**Step 4 Controller instance created:** This request object is sent to ***MvcHandler*** instance to create the controller class instance. Once the controller class object is created it calls the ***Execute*** method of the controller class.

**Creating Response object: -** This phase has two steps executing the action and finally sending the response as a result to the view.

1. **What are Routes? What is Route Table?**

A **route** is a URL pattern that is mapped to a handler. In a file-based application such as a Web Forms application, the handler is an .aspx file. In an **MVC** application, the handler is an action method in a controller class. Each route in an MVC application is represented by a Route object (the Route class is defined in the **System.Web.Routing** namespace). **ASP.NET MVC** applications have a collection of such Route objects, known as the **Routing Table**. The creation and configuration of the routing table happens when the application first starts. There is an event handler in the **Global.asax.cs** file called **Application\_Start**. This event handler calls the *RegisterRoutes* () method in the static class RouteConfig located in the **App\_Start** folder.

The **RouteTable** is a class that stores the URL routes for your application whereas a **RouteCollection** provides a collection of route information to be used when mapping a URI to a controller action.

The **RouteTable** contains a property called Routes that will return a **RouteCollection**. The RouteTable uses a RouteCollection in order to store all the URL routing information it needs to accurately direct URI's to the correct controller action.

1. **How are Routes Configured?**

ASP.NET MVC routes are responsible for determining which controller method (also called controller action) to execute for a given URL. They consist of the following properties:

**Unique name**: A name may be used as a specific reference to a given route

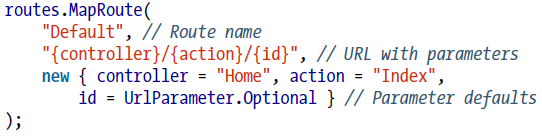
**URL pattern**: A simple pattern syntax that parses matching URLs into meaningful segments

**Defaults**: An optional set of default values for the segments defined in the URL pattern

**Constraints**: A set of constraints to apply against the URL pattern to more narrowly define the URLs that it matches

The default ASP.NET MVC project templates add a generic route that uses the following URL convention to break the URL for a given request into three named segments, wrapped with brackets ({}): “controller”, “action”, and “id”:

{Controller}/{action}/{id}



This route pattern is registered via a call to the ***MapRoute*** () extension method that runs during application startup (located in ***App\_Start***/***RouteConfig***.cs):

1. **What is Caching and what are its types?**

The concept of storing and reusing generated data is called caching, and it’s one of the most effective ways to improve the web application’s performance. The advantages of caching are:

**Reduce hosting server round-trips**: When the content is cached at the client or proxies, it cause minimum request to the server.

**Reduce database server round-trips**: When the content is cached at the web-server, it can eliminate the database request.

**Reduce network traffic**: When content is cached at the client side, it also reduce the network traffic.

**Avoid time consumption for regenerating reusable content**: When reusable content is cached, it avoid the time consumption for regenerating reusable content.

**Improve performance**: Since cached content reduce round-trips, network traffic and avoid time consumption for regenerating reusable content which cause a boost in the performance

Caching techniques generally categorized in to two: Server caching and Client caching.

1. **What is Server side caching and Client side caching?**

**Server-side** caching techniques focus on optimizing the way that the server retrieves, generates, or otherwise manipulates content. The main goal in server-side caching is to limit the amount of work involved in processing a request, be it by avoiding calls to retrieve data from a database or even reducing the number of CPU cycles it takes to generate HTML.

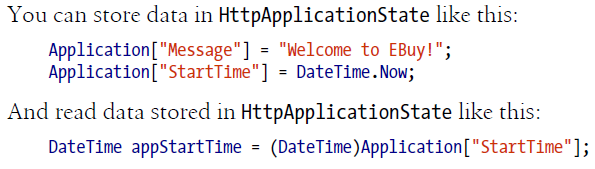
Limiting the work involved in processing requests not only lowers the time it takes to complete each request, but it also makes more server resources available to handle even more requests at the same time.

**Client side** caching is done by the browsers. Modern browsers offer several caching mechanisms of their own. Client-side techniques open up whole new opportunities for improving application performance, from intelligently avoiding duplicate requests all the way to storing content directly to a user’s local environment.

Whereas the main goal of **server-side** caching is to handle requests as quickly and efficiently as possible, the primary goal of **client-side** caching techniques is to avoid making any requests at all. Not only does avoiding unnecessary requests improve the experience for the users who would have made them, it also helps to lower the overall load on the server, which improves the experience of all users of the site at the same time.

1. **What is Application-scoped caching?**

ASP.NET offers an **HttpApplicationState** class to store application-wide data, exposed by the **HttpContext.Application** property. **HttpContext.Application** is a key/value-based collection similar to **HttpContext.Items** and **HttpContext.Session**, except that it lives at the application-level scope so the data that is added to it is able to span users, sessions, and requests.



Data stored in **HttpApplicationState** lives for the lifetime of the **Internet Information Services** worker process that hosts the application instance. Because it’s IIS—not ASP.NET—that manages the lifetime of the worker threads, the **HttpApplicationState** may not be a reliable way to store and retrieve persistent values.

Because of this, **HttpApplicationState** should only be used when the data is guaranteed to be the same across all worker processes. For example, if you are reading the contents of a file on disk or fetching values from a database and those values rarely change, you can use **HttpApplicationState** as a caching layer to avoid making the expensive calls to retrieve those values.

1. **What is ASP.Net cache (application cache)?**

Application data caching is a mechanism for storing the Data objects on cache. It has nothing to do with the page caching and HttpApplicationState caching. ASP.NET allows us to store the object in a Key-Value based cache.

The **System.Web.Cache** object exposed by the **HttpContext.Cache** property provides a better alternative to storing application level data in HttpApplicationState.

System.Web.Cache is a key/value store that acts just like **HttpContext.Items** and **HttpSessionState**; however, the data that it stores is not limited to individual requests or user sessions. In fact, the **Cache** is much more similar to **HttpApplicationState**, except that it is able to cross worker process boundaries and so eliminates most of the issues inherent to **HttpApplicationState**, which generally makes it a better choice.

ASP.NET automatically manages removal of cached items, and notifies the application when such removals happen so that you can repopulate the data. ASP.NET removes cached items when any one of the following occurs:

* The cached item expires.
* The cached item’s dependency changes, invalidating the item.
* The server runs on low resources and must reclaim memory.

1. **How is cached items Expired**

When you add items to the Cache, you can indicate how long it should keep the data around before it expires and should no longer be used. This time span can be expressed in one of two ways:

***Sliding expiration***: Specifies that an item should expire a certain amount of time after it was last accessed. For example, if you cache an item with a sliding expiration of 20 minutes and the application continuously accesses the item every few minutes, the item should stay cached indefinitely (assuming the cached item has no dependencies and the server does not run low on memory). The moment the application stops accessing the item for at least 20 minutes, the item will expire.

***Absolute expiration***: Specifies that an item expires at a specific moment in time, regardless of how often it is accessed. For example, if you cache an item with an absolute expiration of 10:20:00 PM, the item will no longer be available beginning at 10:20:01 PM.

Only one type of expiration—sliding or absolute—may be specified for each item. You cannot use both expiration types on the same cached item. You can use different types of expiration for different cached items, however.

1. **What are Cache dependencies?**

You can also configure an item’s lifetime in the cache to be dependent on other application elements, such as files or databases. When the element that a cache item depends on changes, ASP.NET removes the item from the cache.

For example, if your website displays a report that the application creates from an XML file, you can place the report in the cache and configure it to have a dependency on the XML file. When the XML file changes, ASP.NET removes the report from the cache. The next time your code requests the report, the code first determines whether the report is in the cache and, if not, re-creates it. This ensures that an up-to-date version of the report is always available.

File dependency is not the only dependency that is available in ASP.NET—ASP.NET offers all of the types of dependencies listed below out of the box, along with the ability to create your own dependency policies.

* **Aggregate**: This type combines multiple dependencies (via the System.Web.Caching.AggregateCache Dependency class). The cached item is removed when any of the dependencies in the aggregate change.
* **Custom**: The cached item depends on a custom class that derives from System.Web.Caching.Cache Dependency. For example, you can create a custom web service cache dependency that removes data from the cache when a call to a web service results in a particular value.
* **File**: The cached item depends on an external file and is removed when the file is modified or deleted.
* **Key**: The cached item depends on another item in the application cache (referred to by its cache key). The cached item is removed when the target item is removed from the cache.
* **SQL**: The cached item depends on changes in a table in a Microsoft SQL Server database. The cached item is removed when the table is updated.

1. **What is Cache Scavenging?**

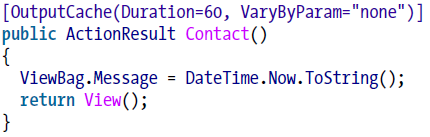
***Scavenging*** is the process of deleting items from the cache when memory is scarce. The items that are removed are typically those that have not been accessed in some time, or those that were marked as low priority when they were added to the cache. ASP.NET uses the CacheItemPriority object to determine which items to scavenge first. In all cases, ASP.NET provides CacheItemRemovedCallback to notify the application that an item is being removed.

1. **What is Output Cache?**

The **OutputCache** filter allow you to cache the data that is output of an action method. ASP.Net has the ability to cache the HTML that is generated as a result of a request. This technique is called ***output caching***.

An output cache is enabled for the action method as a whole. This is done by adding an action filter, **OutputCacheAttribute**, to the controller action method. By doing this the entire output of the Controller Action is cached automatically. By default the cache will be kept for 60 seconds.

For example, in the side example, when you execute this action after the output caching is in place, you’ll see that the value of ViewBag.Message only changes every 60 seconds. For further proof, try adding a breakpoint to the controller method. You’ll see that the breakpoint only gets hit the first time the page is executed (when the cached version does not exist), and any time after the cached version expires.



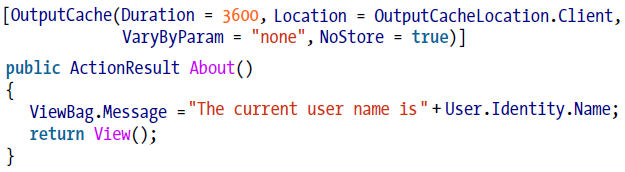
1. **How do you configure the Cache location?**

The OutputCacheAttribute contains several parameters that give you complete control over how and where the page’s content is cached.

By default, the **Location** parameter is set to *Any*, which means content is cached in three locations: the web server, any proxy servers, and the user’s web browser. You can change the Location parameter to any of the following values: **Any**, **Client**, **Downstream**, **Server**, **None**, or **ServerAndClient**.

The default ***Any*** setting is appropriate for most scenarios, but there are times when you need more fine-grained control over where data is cached. For example, say you want to cache a page that displays the current user’s name. If you use the default ***Any*** setting, the name of the first person to request the page will incorrectly be displayed to all users.

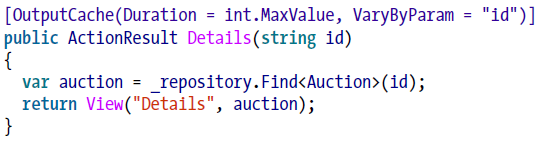
To avoid this, configure the output cache with the Location property set to Output **CacheLocation.Client** and **NoStore** set to *true* so that the data is stored only in the user’s local web browser:



1. **How can you vary the output cache?**

We can vary the output cache based on the request parameters. One of the most powerful aspects of output caching is being able to cache multiple versions of the same controller action based on the request parameters used to call the action.

For example, say you have a controller action named Details that displays the details of an auction. If you use the default output caching setup, the same product details will be displayed for each request. To resolve this issue, you can set the **VaryByParam** property to create different cached versions of the same content based on a form parameter or query string parameter:



The **VaryByParam** property offers quite a few options to help specify when a new version cache will be created. If you specify "none", you will always get the first cached version of the page. If you use "\*", a different cached version will be created whenever any of the form or query string values vary. You can define the list of form or query string parameter caching rules by separating the entries using a query string.

Following is the complete list of properties available on the OutputCacheAttribute.

* **CacheProfile:** The name of the output cache policy to use.
* **Duration:** The amount of time in seconds to cache the content.
* **Enabled:** Enables/disables output cache for the current content.
* **Location:** The location of where to cache the content.
* **NoStore:** Enables/disables HTTP Cache-Control
* **SqlDependency:** The database and table name pairs that the cache entry depends on
* **VaryByContentEncoding:** A comma-delimited list of character sets (content encodings) that the output cache uses to vary the cache entries.
* **VaryByCustom**: A list of custom strings that the output cache uses to vary the cache entries
* **VaryByHeader**: A comma-delimited list of HTTP header names used to vary the cache entries
* **VaryByParam**: A semicolon-delimited list of form POST or query string parameters that the output cache uses to vary the cache entry

1. **What are DataAnnotations?**

The Data Annotations are the attributes that are applied over Model classes in MVC. They perform some routine task on the class that they are applied, like specifying Validation rules, specify how data is displayed, setting the relationships between classes etc. By applying these attributes on the data class or member, you centralize the data definition and do not have to re-apply the same rules in multiple places. Some examples are **DataType** Attribute, **Range** Attribute, **Required** Attribute, **StringLength** Attribute, **DisplayFormat** Attribute, **Display** Attribute, **Association** Attribute etc.

1. **What is CORS?**

The **Cross-origin resource sharing** (**CORS**) is a mechanism that allows many resources (e.g. fonts, JavaScript, etc.) on a **web** **page** to be requested from another **domain** outside the domain from which the resource originated.

**CORS** are HTTP requests for resources from a **different domain** than the domain of the resource making the request.  For instance, a resource loaded from Domain A (http://domaina.example) such as an HTML web page, makes a request for a resource on Domain B (http://domainb.foo), such as an image, using the ***img*** element (http://**domainb**.foo/image.jpg).  This occurs very commonly on the web today — pages load a number of resources in a cross-site manner, including ***CSS style sheets***, ***images*** and ***scripts***, and other resources.

1. **What is REST and RESTful WebApi?**

**Representational State Transfer (REST)** is a software architecture style consisting of guidelines and best practices for creating scalable web services. **REST** is a coordinated set of 6 constraints applied to the design of components in a distributed hypermedia system that can lead to a more performant and maintainable architecture.

The six constraints are:

* **Uniform Interface**: it is fundamental to the design of any REST service. The uniform interface simplifies and decouples the architecture, which enables each part to evolve independently.
* **Client-Server:** A uniform interface separates clients from servers. This separation of concerns means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the ***portability*** of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more ***scalable***. Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.
* **Stateless:** The client–server communication is further constrained by no client context being stored on the server between requests. Each request from any client contains all the information necessary to service the request, and session state is held in the client. The session state can be transferred by the server to another service such as a database to maintain a persistent state for a period and allow authentication. The client begins sending requests when it is ready to make the transition to a new state. While one or more requests are outstanding, the client is considered to be in transition. The representation of each application state contains links that may be used the next time the client chooses to initiate a new state-transition.
* **Cacheable:** As on the World Wide Web, clients can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable, or not, to prevent clients from reusing stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.
* **Layered system:** A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load balancing and by providing shared caches. They may also enforce security policies.
* **Code on demand:** Servers can temporarily extend or customize the functionality of a client by the transfer of executable code. Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript. "Code on demand" is the only optional constraint of the REST architecture.

Web service APIs that adhere to the [REST architectural constraints](http://en.wikipedia.org/wiki/Representational_state_transfer#Architectural_constraints) are called [RESTful APIs](http://en.wikipedia.org/w/index.php?title=RESTful_APIs&action=edit&redlink=1). HTTP based [RESTful APIs](http://en.wikipedia.org/w/index.php?title=RESTful_APIs&action=edit&redlink=1) are defined with these aspects:

* base [URI](http://en.wikipedia.org/wiki/URI), such as *http://example.com/resources/*
* An Internet media type for the data. This is often [***JSON***](http://en.wikipedia.org/wiki/JSON) but can be any other valid Internet media type (e.g. XML, Atom, micro-formats, images, etc.)
* standard HTTP methods (e.g., ***GET, PUT, POST,*** or ***DELETE***)
* hypertext links to reference state
* hypertext links to reference related resources

1. **What is jQuery?**

jQuery is a cross-platform [JavaScript library](http://en.wikipedia.org/wiki/JavaScript_library) designed to simplify the client-side scripting of [HTML](http://en.wikipedia.org/wiki/HTML). jQuery's syntax is designed to make it easier to navigate a document, select DOM elements, create animations, handle events, and develop Ajax applications. jQuery also provides capabilities for developers to create plug-ins on top of the JavaScript library. This enables developers to create abstractions for low-level interaction and animation, advanced effects and high-level, theme-able widgets. The modular approach to the jQuery library allows the creation of powerful dynamic web pages and web applications.

jQuery syntax:

Basic syntax is: **$(selector).action ()**

* A **$** sign to define/access jQuery
* A **(selector)** to "query (or find)" HTML elements
* A jQuery **action()** to be performed on the element(s)

Examples:

**$(this).hide ()** - hides the current element.

**$("p").hide ()** - hides all <p> elements.

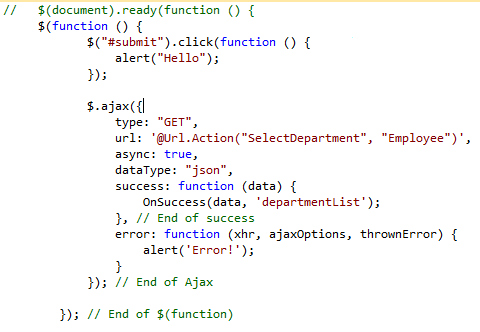
**$(".test").hide ()** - hides all elements with class="test".

**$("#test").hide ()** - hides the element with id="test".

**$("h1, div, p").hide () –** hides all <h1>, <div> and <p> elements.

**$("div > p").hide () -** All <p> elements that are a direct child of a <div> element.

A working example of Ajax:



1. **What is NuGet?**

# 

# *ADO.Net*

1. **What is the namespace in which .NET have the data functionality classes?**

Following are the namespaces provided by .NET for data management:-

* + **System.data**: This contains the basic objects used for accessing and storing relational data, such as Dataset, Debatable, and DataRelation. Each of these is independent of the type of data source and the way we connect to it.
  + **System.Data.OleDB**: Contains the objects that we use to connect to a data source via an OLE-DB provider, such as OleDbConnection, OleDbCommand, etc. These objects inherit from the common base classes, and so have the same properties, methods, and events as the SqlClient equivalents.
  + **System.Data.SqlClient**: This contains the objects that we use to connect to a data source via the Tabular Data Stream (TDS) interface of Microsoft SQL Server (only). This can generally provide better performance as it removes some of the intermediate layers required by an OLE-DB connection.
  + **System.XML:** This Contains the basic objects required to create, read, store, write, and manipulate XML documents according to W3C recommendations.

1. **Can you give an overview of ADO.NET architecture?**

The most important section in ADO.NET architecture is “Data Provider”. Data Provider provides access to data-source (SQL SERVER, ACCESS, and ORACLE).

In short it provides object to achieve functionalities like opening and closing connection, retrieve data and update data. In the below figure you can see the four main sections of a data provider:-

* + **Connection**
  + **Command object** (This is the responsible object to use stored procedures)
  + **Data Adapter** (This object acts as a bridge between data-store and dataset).
  + **DataReader** (This object reads data from data store in forward only mode).
  + **Dataset** object represents disconnected and cached data.
  + **DataView** object is used to sort and filter data in Datatable.

1. **What is difference between dataset and DataReader?**

Following are some major differences between dataset and DataReader:-

* + DataReader provides forward-only and read-only access to data , while the DataSet object can hold more than one table (in other words more than one row set) from the same data source as well as the relationships between them.
  + Dataset is a disconnected architecture while DataReader is connected architecture.
  + Dataset can persist contents while DataReader cannot persist contents, they are forward only.

1. **What is the use of command objects and what are the methods provided by the command object?**

They are used to connect connection object to DataReader or dataset. Following are the methods provided by command object:-

* + **ExecuteNonQuery**: - Executes the command defined in the CommandText property against the connection defined in the Connection property for a query that does not return any rows (an UPDATE, DELETE or INSERT). Returns an Integer indicating the number of rows affected by the query.
  + **ExecuteReader**: - Executes the command defined in the CommandText property against the connection defined in the Connection property. Returns a "reader" object that is connected to the resulting rowset within the database, allowing the rows to be retrieved.
  + **ExecuteScalar**: - Executes the command defined in the CommandText property against the connection defined in the Connection property. Returns only a single value (effectively the first column of the first row of the resulting rowset). Any other returned columns and rows are discarded. Fast and efficient when only a "singleton" value is required

1. **What are basic methods of DataAdapter?**

There are three most commonly used methods of DataAdapter:-

* + **Fill**: - Executes the SelectCommand to fill the DataSet object with data from the data source. It can also be used to update (refresh) an existing table in a DataSet with changes made to the data in the original data Source if there is a primary key in the table in the DataSet.
  + **FillSchema**: - Uses the SelectCommand to extract just the schema for a table from the data source, and creates an empty table in the DataSet object with all the corresponding constraints.
  + **Update**: - Calls the respective *InsertCommand*, *UpdateCommand*, or *DeleteCommand* for each inserted, updated or deleted row in the DataSet so as to update the original data source with the changes made to the content of the DataSet. This is a little like the UpdateBatch method provided by the ADO Recordset object, but in the DataSet it can be used to update more than one table.

1. **What is Dataset object?**

The DataSet provides the basis for disconnected storage and manipulation of relational data. We fill it from a data store, work with it while disconnected from that data store, then reconnect and flush changes back to the data store if required.

1. **What are the various objects in Dataset?**

**Dataset** has a collection of Datatable object within the Tables collection. Each Datatable object contains a collection of **DataRow** objects and a collection of **DataColumn** objects. There are also collections for the primary keys, constraints, and default values used in this table which is called as constraint collection, and the parent and child relationships between the tables. Finally, there is a **DefaultView** object for each table. This is used to create a **DataView** object based on the table, so that the data can be searched, filtered or otherwise manipulated while displaying the data.

1. **How can we force the connection object to close after my DataReader is closed?**

Command method ExecuteReader takes a parameter called as CommandBehavior wherein we can specify saying close connection automatically after the DataReader is close.

pobjDataReader = pobjCommand.ExecuteReader (CommandBehavior.CloseConnection)

**I want to force the DataReader to return only schema of the datastore rather than data?**

pobjDataReader = pobjCommand.ExecuteReader (CommandBehavior.SchemaOnly)

1. **How can we fine tune the command object when we are expecting a single row or a single value?**

Again CommandBehavior enumeration provides two values SingleResult and SingleRow. If you are expecting a single value then pass “CommandBehaviour.SingleResult” and the query is optimized accordingly, if you are expecting single row then pass “CommandBehaviour.SingleRow” and query is optimized according to single row.

1. **Which is the best place to store ConnectionString in .NET projects?**

Config files are the best place to store connection strings. If it’s a web-based application “Web.config” file will be used and if it’s a windows application “App.config” files will be used.

1. **What are the various methods provided by the dataset object to generate XML?**

* ReadXML Reads a XML document in to Dataset.
* GetXML is a function which returns a string containing XML document.
* WriteXML is a function which writes a XML data to disk.

1. **How can we save all data from dataset?**

Dataset has “AcceptChanges” method which commits all the changes since last time “AcceptChanges” has been executed.

1. **How can we check that some changes have been made to dataset since it was loaded?**

*Twist: - How can we cancel all changes done in dataset? , How do we get values which are changed in a dataset?*

For tracking down changes Dataset has two methods which come as rescue “GetChanges “and “HasChanges”.

**GetChanges** Return’s dataset which are changed since it was loaded or since Acceptchanges was executed.

**HasChanges** This property indicates has any change’s been made since the dataset was loaded or AcceptChanges method was executed. If we want to revert or abandon all change’s since the dataset was loaded use “RejectChanges”.

*Note: - One of the most misunderstood things about these properties is that it tracks the changes of actual database. That’s a fundamental mistake; actually, the changes are related only to the changes with dataset, and have nothing to with changes happening in actual database. As dataset are disconnected and do not know anything about the changes happening in actual database.*

1. **How can we add/remove row’s in “DataTable” object of “DataSet”?**

Datatable provides “NewRow” method to add new row to Datatable. Datatable has “DataRowCollection” object which has all rows in a Datatable object.

Following are the methods provided by “DataRowCollection” object:-

**Add** Adds a new row in Datatable

**Remove** Remove’s a “DataRow” object from “Datatable”

**RemoveAt** Remove’s a “DataRow” object from “Datatable” depending on index position of the Datatable.

1. **What’s basic use of “DataView”?**

DataView represent’s a complete table or can be small section of row’s depending on some criteria.It’s best used for sorting and finding data with in “Datatable”.

DataView has the following methods:-

**Find** Take’s an array of value’s and return’s the index of the row.

**FindRow** This also takes array of values but returns a collection of “DataRow”. If we want to manipulate data of “Datatable” object create “DataView” (Using the “DefaultView” we can create “DataView” object) of the “DataTable” object. and use the following functionalities :-

**AddNew** Adds a new row to the “DataView” object.

**Delete** Deletes the specified row from “DataView” object.

1. **How can we add relation’s between tables in a DataSet?**

Dim objRelation As DataRelation objRelation=New DataRelation ("CustomerAddresses", objDataSet.Tables ("Customer").Columns("Custid") ,objDataSet.Tables("Addresses").Columns("Custid\_fk")) objDataSet.Relations.Add(objRelation)

Relation’s can be added between “DataTable” object’s using the “DataRelation” object. Above sample code is trying to build a relationship between “Customer” and “Addresses” “Datatable” using “CustomerAddresses” “DataRelation” object.

1. **What’s the use of CommandBuilder?**

CommandBuilder builds “Parameter” objects automatically. Below is a simple code which uses CommandBuilder to load its parameter object’s.

Dim pobjCommandBuilder As New OleDbCommandBuilder(pobjDataAdapter) pobjCommandBuilder.DeriveParameters(pobjCommand)

Be careful while using “DeriveParameters” method as it needs an extra trip to the Datastore which can be very inefficient.

1. **What’s difference between “Optimistic” and “Pessimistic” locking?**

In pessimistic locking when user wants to update data it locks the record and till then no one can update data Other user’s can only view the data when there is pessimistic locking. In optimistic locking multiple user’s can open the same record for updating, thus increase maximum concurrency. Record is only locked when updating the record. This is the most preferred way of locking practically. Now a days browser based application is very common and having pessimistic locking is not a practical solution.

1. **How many ways are there to implement locking in ADO.NET?**

Following are the ways to implement locking using ADO.NET:-

* When we call “Update” method of DataAdapter it handles locking internally. If the Dataset values are not matching with current data in Database it raises Concurrency exception error. We can easily trap this error using Try...Catch block and raise appropriate error message to the user.
* Define a Datetime stamp field in the table. When actually you are firing the UPDATE SQL statements compare the current timestamp with one existing in the database. Below is a sample SQL which checks for timestamp before updating and any mismatch in timestamp it will not update the records. This is the best practice used by industries for locking.

Update table1 set field1=@test where LastTimeStamp=@Current Timestamp

* Check for original values stored in SQL SERVER and actual changed values. In stored procedure check before updating that the old data is same as the current. Example in the below shown SQL before updating field1 we check that is the old field1 value same. If not then some one else has updated and necessary action has to be taken.

Update table1 set field1=@test where field1 = @oldfield1value

* Locking can be handled at ADO.NET side or at SQL SERVER side i.e. in stored procedures. for more details of how to implementing locking in SQL SERVER read “What are different locks in SQL SERVER ?” in SQL SERVER chapter.

1. **How can we perform transactions in .NET?**

The most common sequence of steps that would be performed while developing a transactional application is as follows:

* + Open a database connection using the Open method of the connection object.
  + Begin a transaction using the Begin Transaction method of the connection object. This method provides us with a transaction object that we will use later to commit or rollback the transaction. Note that changes caused by any queries executed before calling the Begin Transaction method will be committed to the database immediately after they execute. Set the Transaction property of the command object to the above mentioned transaction object.
  + Execute the SQL commands using the command object. We may use one or more command objects for this purpose, as long as the Transaction property of all the objects is set to a valid transaction object.
  + Commit or roll back the transaction using the Commit or Rollback method of the transaction object.
  + Close the database connection.

1. **What are the Isolation Levels for a transaction?**

The Isolation Level property returns an enumerated value that specifies the level of the current transaction and that’s equal to the value passed to the BeginTransaction method, as you can see in the preceding code example. Here’s a brief description of the isolation levels that ADO.NET supports:

■ **Chaos** the pending changes from the more highly isolated transactions can’t be overridden. SQL Server doesn’t support this isolation level.

■ **ReadUncommitted** No shared (read) locks are issued, and no exclusive (write) locks are honored, which means that an application can read data that has been written from inside a transaction but not committed yet. If the transaction is then rolled back, the data that was read doesn’t correspond to the data now in the database, a phenomenon known as *dirty reads*.

■ **ReadCommitted (default)** Shared (read) locks are issued, and exclusive (write) locks are honored; this isolation level avoids dirty reads, but an application isn’t guaranteed to retrieve a given row if the same query is reexecuted (a problem known as *nonrepeatable reads*). Moreover, a reexecuted query might find additional rows because in the meantime the code running in another transaction has inserted one or more records (*phantom rows*).

■ **RepeatableRead** Exclusive locks are placed on all the rows being read so that code running in a transaction can’t even read the data being read from inside another transaction. This isolation level degrades the scalability of the application but prevents the nonrepeatable reads problem. Phantom rows are still possible, however.

■ **Serializable** This level is similar to the RepeatableRead level, but an exclusive lock is issued on the entire range, and therefore code running in another transaction can’t even add a new record in the same range. This isolation level is the least efficient one, but it also solves the phantom row problem: each transaction truly runs in complete isolation.