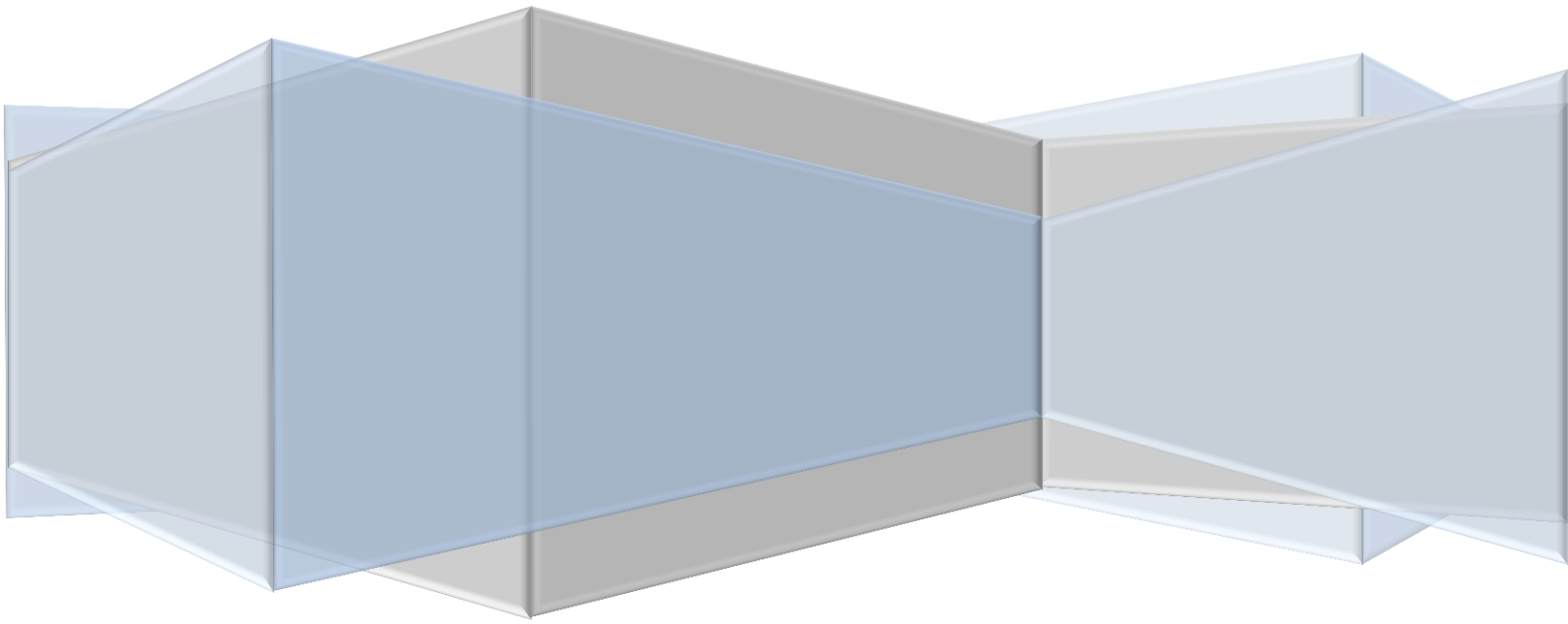
Institute for Advanced Computing and Software Development

Microsoft .Net

PG-DAC



What is the Microsoft.NET?

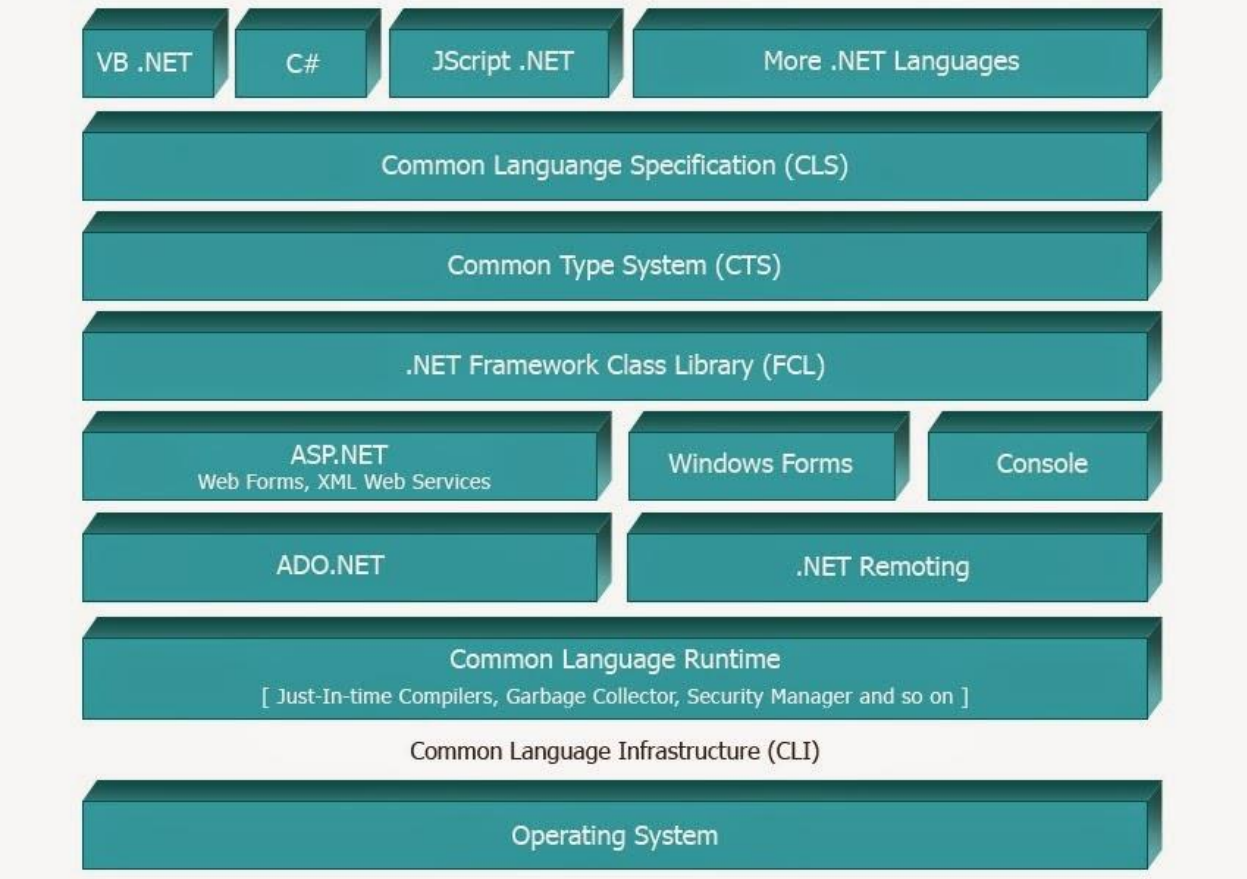
.NET is a set of technologies designed to transform the internet into a full scale distributed platform. It provides new ways of connecting systems, information and devices through a collection of web services. It also provides a language independent, consistent programming model across all tiers of an application.

The goal of the .NET platform is to simplify web development by providing all of the tools and technologies that one needs to build distributed web applications.

What is the .NET Framework?

The .NET Framework is set of technologies that form an integral part of the .NET Platform. It is Microsoft's managed code programming model for building applications that have visually stunning user experiences, seamless and secure communication, and the ability to model a range of business processes.

The .NET Framework has two main components: the common language runtime (CLR) and .NET Framework class library. The CLR is the foundation of the .NET framework and provides a common set of services for projects that act as building blocks to build up applications across all tiers. It simplifies development and provides a robust and simplified environment which provides common services to build and execute an application. The .NET framework class library is a collection of reusable types and exposes features of the runtime. It contains of a set of classes that is used to access common functionality.



What is CLR?

The .NET Framework provides a runtime environment called the Common Language Runtime or CLR. The CLR can be compared to the Java Virtual Machine or JVM in Java. CLR handles the execution of code and provides useful services for the implementation of the program. In addition to executing code, CLR provides services such as memory management, thread management, security management, code verification, compilation, and other system services. It enforces rules that in turn provide a robust and secure execution environment for .NET applications.

What is CTS?

Common Type System (CTS) describes the datatypes that can be used by managed code. CTS defines how these types are declared, used and managed in the runtime. It facilitates cross-language integration, type safety, and high performance code execution. The rules defined in CTS can be used to define your own classes and values.

What is CLS?

Common Language Specification (CLS) defines the rules and standards to which languages must adhere to in order to be compatible with other .NET languages. This enables C# developers to inherit from classes defined in VB.NET or other .NET compatible languages.

CLR and CTS advantages

Language interoperability
Code reusability
Faster development.

What is managed code?

The .NET Framework provides a run-time environment called the Common Language Runtime, which manages the execution of code and provides services that make the development process easier. Compilers and tools expose the runtime's functionality and enable you to write code that benefits from this managed execution environment. The code that runs within the common language runtime is called managed code.

What is MSIL?

When the code is compiled, the compiler translates your code into Microsoft intermediate language (MSIL). MSIL is understood by any environment where .Net framework is installed i.e. write once run anywhere. It is platform independent. MSIL is a part of assembly. The common language runtime includes a JIT compiler for converting this MSIL then to native code.

MSIL contains metadata that is the key to cross language interoperability. Since this metadata is standardized across all .NET languages, a program written in one language can understand the metadata and execute code, written in a different language. MSIL includes instructions for loading, storing, initializing, and calling methods on objects, as well as instructions for arithmetic and logical operations, control flow, direct memory access, exception handling, and other operations. It is also known as CIL(common Intermediate Language).

What is JIT?

JIT(Just in time) is a compiler that converts MSIL to native code JIT is a part of CLR. The native code consists of hardware specific instructions that can be executed by the CPU.

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

What is portable executable (PE)?

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

What is an application domain?

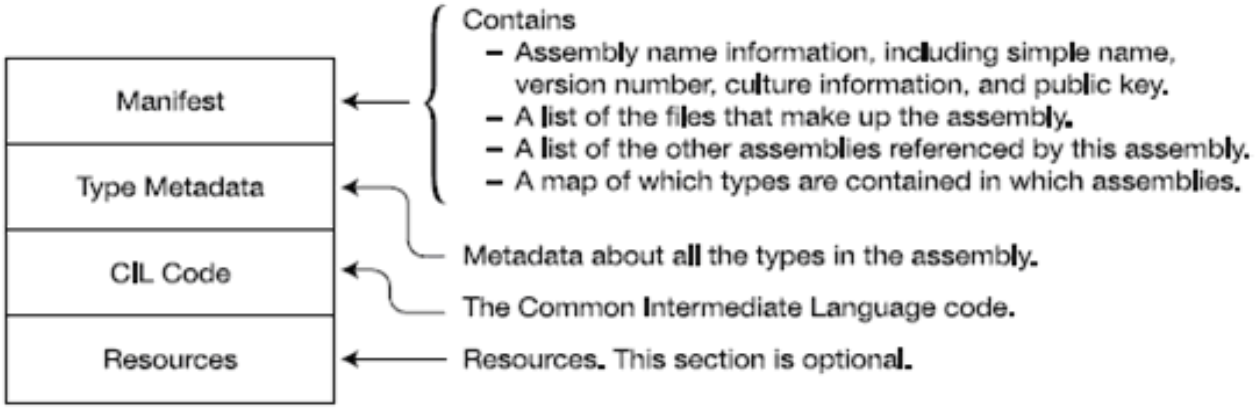
Application domain is the boundary within which an application runs. A process can contain multiple application domains. Application domains provide an isolated environment to applications that is similar to the isolation provided by processes. An application running inside one application domain cannot directly access the code running inside another application domain. To access the code running in another application domain, an application needs to use a proxy.

How does an AppDomain get created?

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

What is an assembly?

An assembly is a collection of one or more .exe or dll's. An assembly is the fundamental unit for application development and deployment in the .NET Framework. An assembly contains a collection of types and resources that are built to work together and form a logical unit of functionality. An assembly provides the CLR with the information it needs to be aware of type implementations.



What are the contents of assembly?

A static assembly can consist of four elements:

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

What are the different types of assembly?

Assemblies can also be private or shared. A private assembly is installed in the installation directory of an application and is accessible to that application only. On the other hand, a shared assembly is shared by multiple applications. A shared assembly has a strong name and is installed in the GAC.

We also have satellite assemblies that are often used to deploy language-specific resources for an application.

What is a dynamic assembly?

A dynamic assembly is created dynamically at run time when an application requires the types within these assemblies.

What is a strong name?

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

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

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

Steps

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

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

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

- drag & drop it to assembly folder (C:\WINDOWS\assembly OR C:\WINNT\assembly) (shfusion.dll tool)
- gacutil -i abc.dll

What is a garbage collector?

One of the very important services provided by CLR during application execution is automatic memory management(allocation and deallocation of memory). In this service, garbage collector plays an important role, it is a back ground thread that performs periodic checks on the managed heap to identify objects that are no longer required by the program and removes them from memory. When a program starts, the system allocates some memory for the program to get executed.

When a C# program instantiates a class, it creates an object.

The program manipulates the object, and at some point the object may no longer be needed. When the object is no longer accessible to the program and becomes a candidate for garbage collection.

There are two places in memory where the CLR stores items while your code executes i.e stack and Heap.The stack keeps track of what's executing in your code (like your local variables), and the heap keeps track of your objects.Value types can be stored on both the stack and the heap.

For an object on the heap, there is always a reference on the stack that points to it.

The garbage collector starts cleaning up only when there is not enough room on the heap to construct a new object.

The stack is automatically cleared at the end of a method. The CLR takes care of this and you don't have to worry about it.

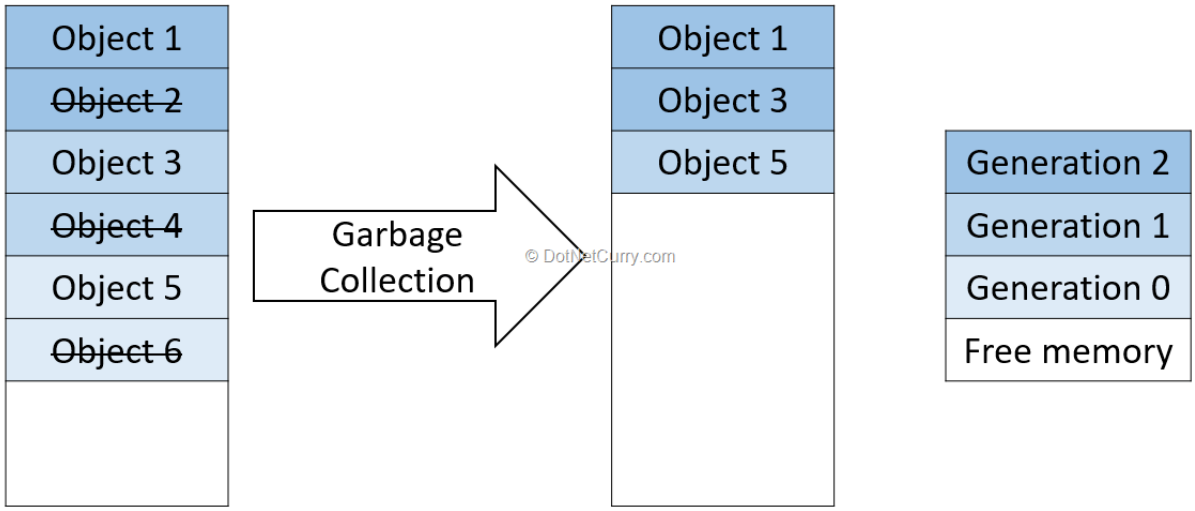
The heap is managed by the garbage collector.

In unmanaged environments without a garbage collector, you have to keep track of which objects were allocated on the heap and you need to free them explicitly. In the .NET Framework, this is done by the garbage collector.

All managed objects in the .NET framework are allocated on the managed heap. They are placed contiguously as they are created.

The garbage collection consists of three phases:

- In the **marking phase,** a list of all objects in use is created by following the references from all the root objects, i.e. variables on stack and static objects. Any allocated objects not on the list become candidates to be deleted from the heap. The objects on the list will be relocated in the compacting phase if necessary, to compact the heap and remove any unused memory between them.
- In the **relocation phase,** all references to remaining objects are updated to point to the new location of objects to which they will be relocated in the next phase.
- In the **compacting phase,** the heap finally gets compacted as all the objects that are not in use any more are released and the remaining objects are moved accordingly. The order of remaining objects in the heap stays intact.



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).
Generation 0 – Short-lived Objects
Generation 1- As a buffer between short lived and long lived objects
Generation 2 – Long lived objects

How value types get collected v/s reference types?

Value types are gets stored on the stack and therefore it gets removed from the stack by its pop method when application is done with its use. Reference types get stored on heap so gets collected by garbage collector.

Dispose v/s Finalize

Dispose	*Finalize*
It is used to free unmanaged resources at any time.	It can be used to free unmanaged resources held by an object before that object is destroyed.
It is called by user code and the class which is implementing dispose method, must has to implement IDisposable interface.	It is called by Garbage Collector and cannot be called by user code.
It is implemented by implementing IDisposable interface Dispose() method.	It is implemented with the help of Destructors
There is no performance costs associated with Dispose method.	There is performance costs associated with Finalize method since it doesn't clean the memory immediately and called by GC automatically.

How to Force Garbage Collection?

You can force this by adding a call to GC.Collect.

What is Finalizer?

inalizers (which are also called **destructors**) are used to perform any necessary final clean-up when a class instance is being collected by the garbage collector.

Remarks

- Finalizers cannot be defined in structs. They are only used with classes.
- A class can only have one finalizer.
- Finalizers cannot be inherited or overloaded.
- Finalizers cannot be called. They are invoked automatically.
- A finalizer does not take modifiers or have parameters.

For example, the following is a declaration of a finalizer for the Car class.

```
class Car
{
  ~Car()  // finalizer
  {
    // cleanup statements...
  }
}
```

The finalizer implicitly calls Finalize on the base class of the object. Therefore, a call to a finalizer is implicitly translated to the following code:

```
protected override void Finalize()
{
  try
  {
    // Cleanup statements...
  }
  finally
  {
    base.Finalize();
  }
}
```

This means that the Finalize method is called recursively for all instances in the inheritance chain, from the most-derived to the least-derived.

The programmer has no control over when the finalizer is called because this is determined by the garbage collector. The garbage collector checks for objects that are no longer being used by the application. If it considers an object eligible for finalization, it calls the finalizer (if any) and reclaims the memory used to store the object. In .NET Framework applications (but not in .NET Core applications), finalizers are also called when the program exits.

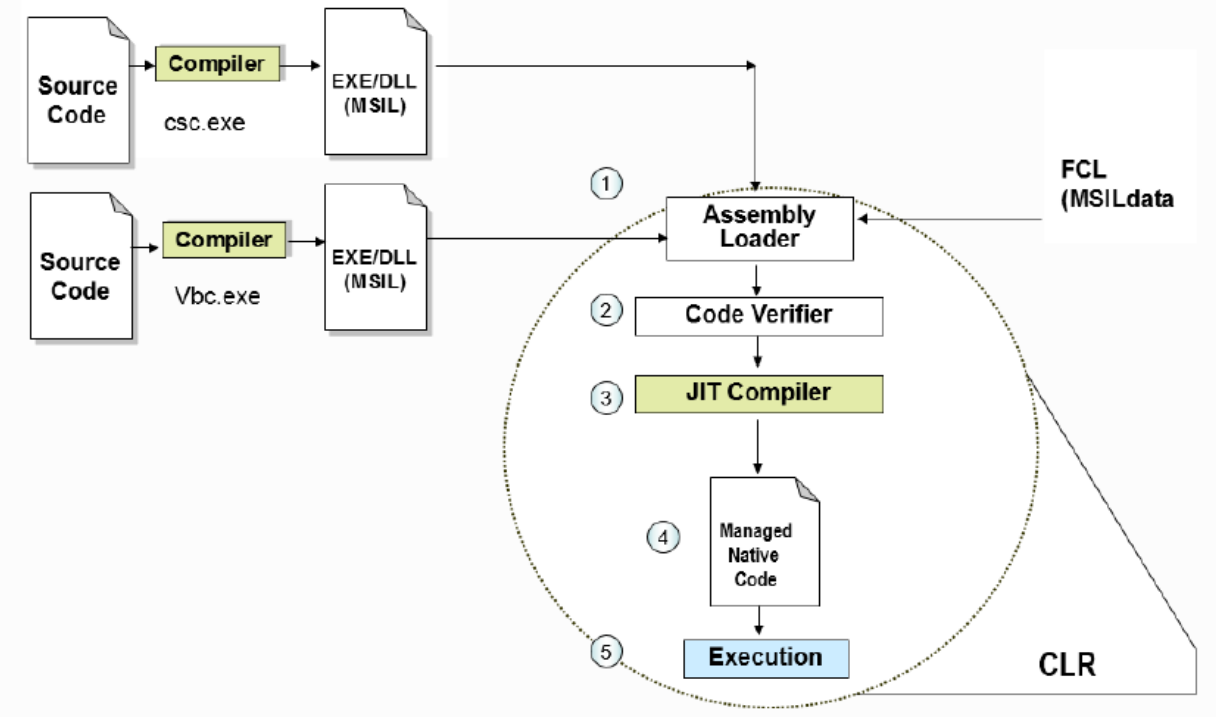
.NET Framework garbage collector implicitly manages the allocation and release of memory for your objects. However, when your application encapsulates unmanaged resources such as windows, files, and network connections, you should use finalizers to free those resources. When the object is eligible for finalization, the garbage collector runs the `Finalize` method of the object.

Explicit release of resources

If your application is using an expensive external resource, we also recommend that you provide a way to explicitly release the resource before the garbage collector frees the object. You do this by implementing a Dispose method from the IDisposable interface that performs the necessary cleanup for the object. This can considerably improve the performance of the application. Even with this explicit control over resources, the finalizer becomes a safeguard to clean up resources if the call to the Dispose method failed.

Explain the execution cycle of DotNet Application.

Execution Process in .NET Environment



1. Sorce code is compiled by language specific compilers.
2. After first compilation, assembly(exe/dll) is created, which is a logical unit of deployment.
3. Assembly is deployed on CLR for actual execution
4. Loader is responsible for loading and initializing assemblies, modules, resources, and types. When you run a .NET application on one of these systems that have an updated OS loader, the OS loader recognizes the .NET application and thus passes control to the CLR. The CLR then finds the entry point, which is typically Main( ), and executes it to jump-start the application. But before Main( ) can execute, the class loader must find the class that exposes Main( ) and load the class. In addition, when Main( ) instantiates an object of a specific class, the class loader also kicks in. In short, the class loader performs its magic the first time a type is referenced. The class loader loads .NET classes into memory and prepares them for execution. Before it can successfully do this, it must locate the target class. the class loader uses the appropriate metadata to initialize the static variables and instantiate an object of the loaded class for you.
5. Code Verifier: The key here is type safety, and it is a fundamental concept for code verification in .NET. Within the VES, the verifier is the component that executes at

runtime to verify that the code is type safe. Note that this type verification is done at runtime and that this is a fundamental difference between .NET and other environments. By verifying type safety at runtime, the CLR can prevent the execution of code that is not type safe and ensure that the code is used as intended. In short, type safety means more reliability.Let's talk about where the verifier fits within the CLR. After the class loader has loaded a class and before a piece of IL code can execute, the verifier kicks in for code that must be verified. The verifier is responsible for verifying that:

- The metadata is well formed, meaning the metadata must be valid.
- The IL code is type safe, meaning type signatures are used correctly.

  Both of these criteria must be met before the code can be executed because JIT compilation will take place only when code and metadata have been successfully verified. In addition to checking for type safety, the verifier also performs rudimentary control-flow analysis of the code to ensure that the code is using types correctly. You should note that since the verifier is a part of the JIT compilers, it kicks in only when a method is being invoked, not when a class or assembly is loaded. You should also note that verification is an optional step because trusted code will never be verified but will be immediately directed to the JIT compiler for compilation.

6. JIT Compiler: JIT compilers play a major role in the .NET platform because all .NET PE files contain IL and metadata, not native code. The JIT compilers convert IL to native code so that it can execute on the target operating system. For each method that has been successfully verified for type safety, a JIT compiler in the CLR will compile the method and convert it into native code.

WHAT IS DELEGATE?

A delegate is reference type that basically encapsulates the reference of methods. It is a similar to class that store the reference of methods which have same signature as delegate has.

It is very similar to a function pointer in C++, but delegate is type safe, object oriented and secure as compare to C++ function pointer.

WHY DO WE USE DELEGATE?

There are following reason because of we use delegate.

1. It is used for type safety.
2. For executing multiple methods through one execution.
3. It allows us to pass methods as parameter.
4. For asynchronous programming.
5. For Call back method implementation.
6. It is also used when working with event based programming.
7. When creating anonymous method.
8. When working with lambda expression.

HOW MANY TYPES OF DELEGATE IN C#?

There are three types of delegates available in C#.

1. Simple/Single Delegate
2. Multicast Delegate
3. Generic Delegate

WHAT ARE THE WAYS TO CREATE AND USE DELEGATE?

There are only five steps to create and use a delegate and these are as following.

1. Declare a delegate type.
2. Create or find a method which has same type of signature.
3. Create object/instance of delegate
4. Pass the reference of method with delegate instance.
5. At last Invoke the delegate object.

WHAT IS SINGLE DELEGATE?

When Delegate takes reference with single method only then it is called Single Delegate. It is used for invocation of only one reference method.

WHAT IS MULTICAST DELEGATE?

When Delegate takes reference with multiple methods then it is called multicast delegate. It is used for invocation of multiple reference methods. We can add or remove references of multiple methods with same instance of delegate using (+) or (-) sign respectively. It need to consider here that all methods will be invoked in one process and that will be in sequence.

WHAT IS MULTICAST DELEGATE

When Delegate takes reference with multiple methods then it is called multicast delegate. It is used for invocation of multiple reference methods. We can add or remove references of multiple methods with same instance of delegate using (+) or (-) sign respectively. It need to consider here that all methods will be invoked in one process and that will be in sequence.

Following example has a delegate name as "TestMultiCastDelegate" with one parameter as int.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;


namespace EventAndDelegateExamples
{
   delegate void TestMultiCastDelegate(int a);
   class MultiCastDelegateExample
   {
      internal void GetSalary(int AnnualSalary)
      {
         Console.WriteLine("Monthly Salary is "+ AnnualSalary / 12);
      }
```

```
    internal void GetSalaryAfterDeduction(int AnnualSalary)
    {
        if (AnnualSalary > 12000)
        {
            Console.WriteLine("Monthly Salry after deduction is "+ ((AnnualSalary / 12) - 500));
        }
        else
        {
            Console.WriteLine("Monthly Salry after deduction is " +AnnualSalary / 12);
        }
    }
  }
}
```

Below code snippet shows that first we are creating a instance of TestMultiCastDelegate and passing the reference of methods [GetSalary] and [GetSalaryAfterDeduction] with "objDelegate2".

```
MultiCastDelegateExample objMultiCastDelegateExample = new MultiCastDelegateExample();


//Creating the instance of the delegate
TestMultiCastDelegate objDelegate2 = null;


//Referencing the multiple methods using + sign
objDelegate2 += objMultiCastDelegateExample.GetSalary;
objDelegate2 += objMultiCastDelegateExample.GetSalaryAfterDeduction;
```

```
objDelegate2.Invoke(60000);
```

HOW TO ACHIEVE CALLBACK IN DELEGATE?

Callback is term where a process is going on and in between it targets some achievement then it return to main method. For callback, we just need to encapsulate the method with delegate.

```
objCallBackMethodExample.CheckEvenEvent += new OnEvenNumberHandler(objCallBackMethodExample.CallBackMethod);
```

Let see an example, CallBackMethodExample is a class which have a method CallBackMethod. It will be executed when some criteria will be fulfill.

```
public delegate void OnEvenNumberHandler(object sender, EventArgs e);


public class CallBackMethodExample
{
    public void CallBackMethod(object sender, EventArgs e)
    {
        Console.WriteLine("Even Number has found !");
    }
}
```

When we are going to call this method using Delegate for callback, only need to pass this method name as a reference.

```
CallBackMethodExample objCallBackMethodExample = new CallBackMethodExample();


objCallBackMethodExample.CheckEvenEvent += new OnEvenNumberHandler(objCallBackMethodExample.CallBackMethod);
Random random = new Random();
for (int i = 0; i < 6; i++)
{
```

```
  var randomNumber = random.Next(1, 10);

  Console.WriteLine(randomNumber);

  if (randomNumber % 2 == 0)

  {

     objCallBackMethodExample.OnCheckEvenNumber();

  }
}
```

HOW TO ACHIEVE ASYNC CALLBACK IN DELEGATE?

Async callback means, process will be going on in background and return back when it will be completed. In C#, Async callback can be achieved using AsyncCallback predefined delegate as following.

```
public delegate void AsyncCallback(IAsyncResult ar);
```

Kindly follow the following example to understand the Async callback. There is a class name as CallBackMethodExample which have two method, one is TestMethod and other one is AsyncCallBackFunction. But as we can see AsyncCallBackFunction is taking IAsyncResult as a parameter.

```
  public delegate void OnEvenNumberHandler(object sender, EventArgs e);

  public class CallBackMethodExample

  {

     public void TestMethod(object sender, EventArgs e)
```

```
    {
        Console.WriteLine("This is simple test method");
    }

    public void AsynCallBackFunction(IAsyncResult asyncResult)
    {
        OnEvenNumberHandler del = (OnEvenNumberHandler)asyncResult.AsyncState;
        del.EndInvoke(asyncResult);
        Console.WriteLine("Wait For 5 Seconds");
        Console.WriteLine("...");
        Console.WriteLine("...");
        Console.WriteLine("...");
        Console.WriteLine("...");
        System.Threading.Thread.Sleep(5000);
        Console.WriteLine("Async Call Back Function Completed");
    }
  }
```

When we are going to invoke the delegate, only just need to pass AsycnCallBackFunction with delegate instance.

```
CallBackMethodExample objCallBackMethodExample = new CallBackMethodExample();
OnEvenNumberHandler objDelegate3 = new OnEvenNumberHandler(objCallBackMethodExample.TestMethod);
objDelegate3.BeginInvoke(null, EventArgs.Empty, new AsyncCallback(objCallBackMethodExample.AsynCallBackFunction), objDelegate3);
Console.WriteLine("End of Main Function");
Console.WriteLine("Waiting for Async Call Back Function");
```

```
End of Main Function
Waiting for Async Call Back Function
This is simple test method
Wait For 5 Seconds
...
...
...
...
Async Call Back Function Completed
```

What are events?

Events are higher level of encapsulation over delegates. Events use delegates internally. Delegates are naked and when passed to any other code, the client code can invoke the delegate. Event provides a publisher / subscriber mechanism model.

So subscribers subscribe to the event and publisher then push messages to all the subscribers. Below is a simple code snippet for the same:-

Create a delegate and declare the event for the same.

```
public delegate void CallEveryone();
public event CallEveryone MyEvent;
```

Raise the event.

```
MyEvent();
```

Attached client methods to the event are fired / notified.

```
obj.MyEvent += Function1;
```

Difference between delegate and events: -

They cannot be compared because one derives from the other.

- Actually, events use delegates in bottom. But they add an extra layer of security on the delegates, thus forming the publisher and subscriber model.
- As delegates are function to pointers, they can move across any clients. So any of the clients can add or remove events, which can be confusing. But events give the extra protection / encapsulation by adding the layer and making it a publisher and subscriber model.

What are attributes?

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called *reflection.*

Attributes have the following properties:

- Attributes add metadata to your program. *Metadata* is information about the types defined in a program. All .NET assemblies contain a specified set of metadata that describes the types and type members defined in the assembly. You can add custom attributes to specify any additional information that is required.
- You can apply one or more attributes to entire assemblies, modules, or smaller program elements such as classes and properties.
- Attributes can accept arguments in the same way as methods and properties.
- Your program can examine its own metadata or the metadata in other programs by using reflection.

  There are two types of attributes:

  1. Predefined attributes: these are provided by FCL. Ex. Obsolete, Serializable, Conditional etc.
  2. Custom attributes: these are developed by developers as per the requirements.

Note: All the attributes are derived directly or indirectly from System.Attribute class.

How to create custom attributes?

The primary steps to properly design custom attribute classes are as follows:

- Applying the AttributeUsageAttribute
- Declaring the attribute class
- Declaring constructors
- Declaring properties

Applying the AttributeUsageAttribute

A custom attribute declaration begins with the System.AttributeUsageAttribute, which defines some of the key characteristics of your attribute class. For example, you can specify whether your attribute can be inherited by other classes or specify which elements the attribute can be applied to. The following code fragment demonstrates how to use the AttributeUsageAttribute.

```
[AttributeUsage(AttributeTargets.All, Inherited = false, AllowMultiple = true)]
```

The AttributeUsageAttribute has three members that are important for the creation of custom attributes: AttributeTargets, Inherited, and AllowMultiple.

AttributeTargets Member

In the previous example, AttributeTargets.All is specified, indicating that this attribute can be applied to all program elements. Alternatively, you can specify AttributeTargets.Class,

indicating that your attribute can be applied only to a class, or AttributeTargets.Method, indicating that your attribute can be applied only to a method. All program elements can be marked for description by a custom attribute in this manner.

You can also pass multiple AttributeTargets values. The following code fragment specifies that a custom attribute can be applied to any class or method.

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
```

Inherited Property

The AttributeUsageAttribute.Inherited property indicates whether your attribute can be inherited by classes that are derived from the classes to which your attribute is applied. This property takes either a true (the default) or false flag. In the following example, MyAttribute has a default Inheritedvalue of true, while YourAttribute has an Inherited value of false.

```
// This defaults to Inherited = true.
public class MyAttribute : Attribute
{
   //...
}
[AttributeUsage(AttributeTargets.Method, Inherited = false)]
public class YourAttribute : Attribute
{
   //...
}
```

The two attributes are then applied to a method in the base class MyClass.

```
public class MyClass
{
   [MyAttribute]
   [YourAttribute]
   public virtual void MyMethod()
   {
      //...
   }}
```

Finally, the class YourClass is inherited from the base class MyClass. The method MyMethod shows MyAttribute, but not YourAttribute.

```
public class YourClass : MyClass
{
   // MyMethod will have MyAttribute but not YourAttribute.
```

```
    public override void MyMethod()
    {
        //...
    }

}
```

AllowMultiple Property

The AttributeUsageAttribute.AllowMultiple property indicates whether multiple instances of your attribute can exist on an element. If set to true, multiple instances are allowed; if set to false(the default), only one instance is allowed.

In the following example, MyAttribute has a default AllowMultiple value of false, while YourAttribute has a value of true.

```
public class MyAttribute : Attribute
{
}

[AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
public class YourAttribute : Attribute
{
}
```

When multiple instances of these attributes are applied, MyAttribute produces a compiler error. The following code example shows the valid use of YourAttribute and the invalid use of MyAttribute.

```
public class MyClass
{
    // This produces an error.
    // Duplicates are not allowed.
    [MyAttribute]
    [MyAttribute]
    public void MyMethod()
    {
        //...
    }

    // This is valid.
    [YourAttribute]
    [YourAttribute]
    public void YourMethod()
```

```
  {
    //...
  }
}
```

If both the AllowMultiple property and the Inherited property are set to true, a class that is inherited from another class can inherit an attribute and have another instance of the same attribute applied in the same child class. If AllowMultiple is set to false, the values of any attributes in the parent class will be overwritten by new instances of the same attribute in the child class.

Declaring the Attribute Class

After you apply the AttributeUsageAttribute, you can begin to define the specifics of your attribute. The declaration of an attribute class looks similar to the declaration of a traditional class, as demonstrated by the following code.

```
[AttributeUsage(AttributeTargets.Method)]
public class MyAttribute : Attribute
{
  // . . .
}
```

This attribute definition demonstrates the following points:

- Attribute classes must be declared as public classes.
- By convention, the name of the attribute class ends with the word **Attribute**. While not required, this convention is recommended for readability. When the attribute is applied, the inclusion of the word Attribute is optional.
- All attribute classes must inherit directly or indirectly from System.Attribute.
- In Microsoft Visual Basic, all custom attribute classes must have the System.AttributeUsageAttribute attribute.

Declaring Constructors

Attributes are initialized with constructors in the same way as traditional classes. The following code fragment illustrates a typical attribute constructor. This public constructor takes a parameter and sets a member variable equal to its value.

```
public MyAttribute(bool myvalue)
{
  this.myvalue = myvalue;
}
```

You can overload the constructor to accommodate different combinations of values. If you also define a property for your custom attribute class, you can use a combination of named and positional parameters when initializing the attribute. Typically, you define all required parameters as positional and all optional parameters as named. In this case, the attribute cannot be initialized without the required parameter. All other parameters are optional. Note that in Visual Basic, constructors for an attribute class should not use a ParamArray argument.

The following code example shows how an attribute that uses the previous constructor can be applied using optional and required parameters. It assumes that the attribute has one required Boolean value and one optional string property.

```
// One required (positional) and one optional (named) parameter are applied.
[MyAttribute(false, OptionalParameter = "optional data")]
public class SomeClass
{
   //...
}
 [MyAttribute(false)]
public class SomeOtherClass
{
   //...
}
```

Declaring Properties

If you want to define a named parameter or provide an easy way to return the values stored by your attribute, declare a property. Attribute properties should be declared as public entities with a description of the data type that will be returned. Define the variable that will hold the value of your property and associate it with the **get** and **set** methods. The following code example demonstrates how to implement a simple property in your attribute.

```
public bool MyProperty
{
   get {return this.myvalue;}
   set {this.myvalue = value;}
}
```

Custom Attribute Example

This section incorporates the previous information and shows how to design a simple attribute that documents information about the author of a section of code. The attribute in this example stores the name and level of the programmer, and whether the code has been

reviewed. It uses three private variables to store the actual values to save. Each variable is represented by a public property that gets and sets the values. Finally, the constructor is defined with two required parameters.

```
[AttributeUsage(AttributeTargets.All)]
public class DeveloperAttribute : Attribute
{
   // Private fields.
   private string name;
   private string level;
   private bool reviewed;

   // This constructor defines two required parameters: name and level.

   public DeveloperAttribute(string name, string level)
   {
      this.name = name;
      this.level = level;
      this.reviewed = false;
   }

   // Define Name property.
   // This is a read-only attribute.


  public virtual string Name
   {
      get {return name;}
   }

   // Define Level property.
   // This is a read-only attribute.

   public virtual string Level
   {
      get {return level;}
   }

   // Define Reviewed property.
   // This is a read/write attribute.

   public virtual bool Reviewed
   {
      get {return reviewed;}
```

```
    set {reviewed = value;}
  }
}
```

You can apply this attribute using the full name, DeveloperAttribute, or using the abbreviated name, Developer, in one of the following ways.

```
[Developer("Joan Smith", "1")]
```

-or-

```
[Developer("Joan Smith", "1", Reviewed = true)]
```

The first example shows the attribute applied with only the required named parameters, while the second example shows the attribute applied with both the required and optional parameters.

What is reflection?

Reflection provides objects (of type Type) that describe assemblies, modules and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. If you are using attributes in your code, reflection enables you to access them.

Uses for Reflection C#

There are several uses including:

1. Use Module to get all global and non-global methods defined in the module.
2. Use MethodInfo to look at information such as parameters, name, return type, access modifiers and implementation details.
3. Use EventInfo to find out the event-handler data type, the name, declaring type and custom attributes.
4. Use ConstructorInfo to get data on the parameters, access modifiers, and implementation details of a constructor.
5. Use Assembly to load modules listed in the assembly manifest.
6. Use PropertyInfo to get the declaring type, reflected type, data type, name and writable status of a property or to get and set property values.
7. Use CustomAttributeData to find out information on custom attributes or to review attributes without having to create more instances.

Other uses for Reflection include constructing symbol tables, to determine which fields to persist and through serialization.

Reflection Examples

This example shows how to dynamically load assembly, how to create object instance, how to invoke method or how to get and set property value.

Create instance from assembly that is in your project References

The following examples create instances of DateTime class from the System assembly:

```
// create instance of class DateTime
DateTime dateTime = (DateTime)Activator.CreateInstance(typeof(DateTime));

// create instance of DateTime, use constructor with parameters (year, month, day)
DateTime dateTime = (DateTime)Activator.CreateInstance(typeof(DateTime),
                                    new object[] { 2008, 7, 4 });
```

Create instance from dynamically loaded assembly

All the following examples try to access to **sample class Calculator** from Test.dll assembly. The calculator class can be defined like this.

```
namespace Test
{
   public class Calculator
   {
      public Calculator() { ... }
      private double _number;
      public double Number { get { ... } set { ... } }
      public void Clear() { ... }
      private void DoClear() { ... }
      public double Add(double number) { ... }
      public static double Pi { ... }
      public static double GetPi() { ... }
   }
}
```

Examples of using reflection to load the Test.dll assembly, to create instance of the Calculator class and to access its members (public/private, instance/static).

```
// dynamically load assembly from file Test.dll
Assembly testAssembly = Assembly.LoadFile(@"c:\Test.dll");

// get type of class Calculator from just loaded assembly
Type calcType = testAssembly.GetType("Test.Calculator");

// create instance of class Calculator
object calcInstance = Activator.CreateInstance(calcType);

// get info about property: public double Number
PropertyInfo numberPropertyInfo = calcType.GetProperty("Number");

// get value of property: public double Number
double value = (double)numberPropertyInfo.GetValue(calcInstance, null);

// set value of property: public double Number
numberPropertyInfo.SetValue(calcInstance, 10.0, null);

// get info about static property: public static double Pi
PropertyInfo piPropertyInfo = calcType.GetProperty("Pi");

// get value of static property: public static double Pi
double piValue = (double)piPropertyInfo.GetValue(null, null);

// invoke public instance method: public void Clear()
calcType.InvokeMember("Clear",
   BindingFlags.InvokeMethod | BindingFlags.Instance | BindingFlags.Public,
   null, calcInstance, null);

// invoke private instance method: private void DoClear()
calcType.InvokeMember("DoClear",
   BindingFlags.InvokeMethod | BindingFlags.Instance | BindingFlags.NonPublic,
   null, calcInstance, null);

// invoke public instance method: public double Add(double number)
double value = (double)calcType.InvokeMember("Add",
   BindingFlags.InvokeMethod | BindingFlags.Instance | BindingFlags.Public,
   null, calcInstance, new object[] { 20.0 });

// invoke public static method: public static double GetPi()
double piValue = (double)calcType.InvokeMember("GetPi",
```

```
    BindingFlags.InvokeMethod | BindingFlags.Static | BindingFlags.Public,
    null, null, null);
// get value of private field: private double _number
double value = (double)calcType.InvokeMember("_number",
    BindingFlags.GetField | BindingFlags.Instance | BindingFlags.NonPublic,
    null, calcInstance, null);
```

File Handling in C#

All the objects created at runtime reside on heap section primary memory. Once they are out of scope the objects are released by garbage collector. So the data stored in an object cannot be reused once the program is terminated. Therefore we store the data on secondary storage devices in the form of file. File handling is an unmanaged resource in your application system. It is outside your application domain (unmanaged resource). It is not managed by CLR.

What is a stream?

A stream is a sequence of bytes. In the file system, streams contain the data that is written to a file, and that gives more information about a file than attributes and properties. When you open a file for reading or writing, it becomes stream. Stream is a sequence of bytes traveling from a source to a destination over a communication path. The two basic streams are input and output streams. Input stream is used to read and output stream is used to write. The System.IO namespace includes various classes for file handling.

Stream is the abstract base class of all streams. A stream is an abstraction of a sequence of bytes, such as a file, an input/output device, an inter-process communication pipe, or a TCP/IP socket. The Stream class and its derived classes provide a generic view of these different types of input and output, and isolate the programmer from the specific details of the operating system and the underlying devices.

Streams involve three fundamental operations:

- You can read from streams. Reading is the transfer of data from a stream into a data structure, such as an array of bytes.
- You can write to streams. Writing is the transfer of data from a data structure into a stream.
- Streams can support seeking. Seeking refers to querying and modifying the current position within a stream. Seek capability depends on the kind of backing store a stream has. For example, network streams have no unified concept of a current position, and therefore typically do not support seeking.

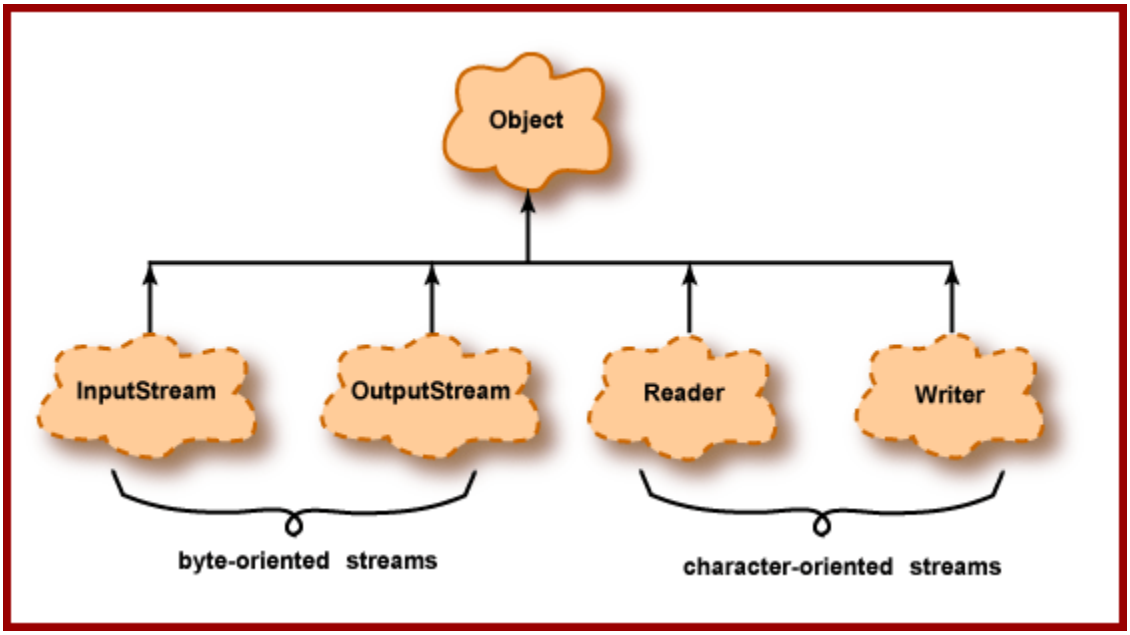
Some of the more commonly used streams that inherit from Stream are FileStream, and MemoryStream.

What is FileStream class?

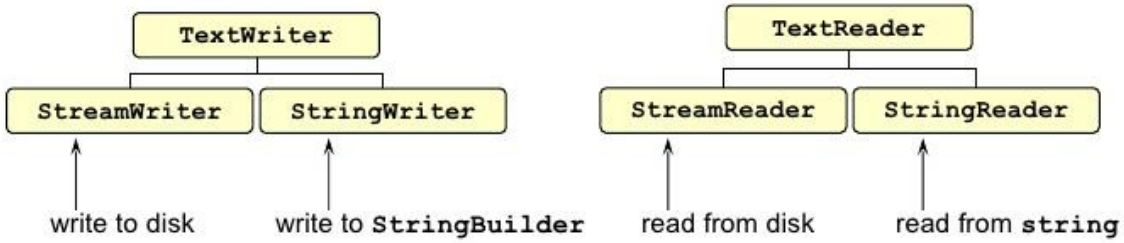
Use the FileStream class to read from, write to, open, and close files on a file system, and to manipulate other file-related operating system handles, including pipes, standard input, and standard output. You can use the Read, Write, CopyTo, and Flush methods to perform

synchronous operations, or the ReadAsync, WriteAsync, CopyToAsync, and FlushAsync methods to perform asynchronous operations.

Why there is a need of reader and writer classes?



As shown in the picture above, FileStream is byte oriented. But if we want to read and write characters then we need character reader and writer streams .



TextReader is the abstract base class of StreamReader and StringReader, which read characters from streams and strings, respectively. Use these derived classes to open a text file for reading a specified range of characters, or to create a reader based on an existing stream.

TextWriter is the abstract base class of StreamWriter and StringWriter, which write characters to streams and strings, respectively. Create an instance of TextWriter to write an object to a string, write strings to a file, or to serialize XML.

The BinaryReader class is used to read binary data from a file. A BinaryReader object is created by passing a FileStream object to its constructor.

The BinaryWriter class is used to write binary data to a stream. A BinaryWriter object is created by passing a FileStream object to its constructor.

Program of reading a file using StreamReader class:

```
using System;
using System.Collections.Generic;
using System.IO;


class Program
{
   static void Main()
   {
      List<string> list = new List<string>();
      using (StreamReader reader = new StreamReader("file.txt"))
      {
         string line;
         while ((line = reader.ReadLine()) != null)
         {
            list.Add(line); // Add to list.
            Console.WriteLine(line); // Write to console.
         }
      }
   }
}
```

Program for reading and writing binary data using BinaryReader and BinaryWriter:

```
class Program
  {
     static void Main(string[] args)
     {

        // Create the new, empty data file.
        string fileName = @"C:\Temp.data";
        if (File.Exists(fileName))
        {
           Console.WriteLine(fileName + " already exists!");
           return;
        }
        FileStream fs = new FileStream(fileName, FileMode.CreateNew);
        // Create the writer for data.
        BinaryWriter w = new BinaryWriter(fs);
        // Write data to Test.data.
        for (int i = 0; i < 11; i++)
        {
           w.Write((int)i);
        }
        w.Close();
        fs.Close();
        // Create the reader for data.
        fs  = new FileStream(fileName, FileMode.Open, FileAccess.Read);
        BinaryReader r = new BinaryReader(fs);
        // Read data from Test.data.
        for (int i = 0; i < 11; i++)
        {
           Console.WriteLine(r.ReadInt32());
        }
        r.Close();
        fs.Close();
     }
  }
```

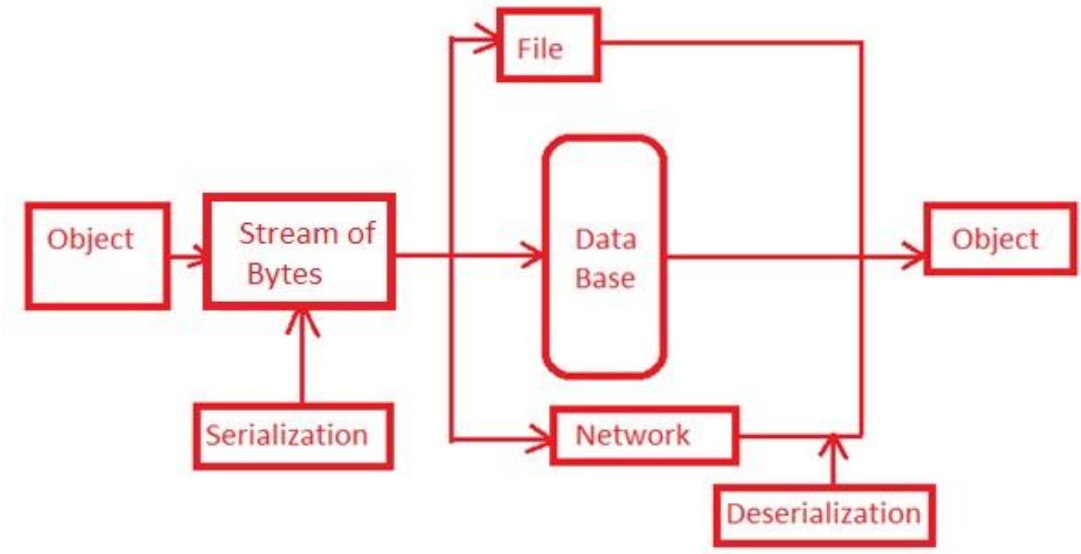
What is Serialization?

Serialization is converting object to stream of bytes. It is a process of bringing an object into a form that it can be written on stream. It's the process of converting the object into a form so

that it can be stored on a file, database or memory. It is transferred across the network. Main purpose is to save the state of the object.

What is deserialization?

Deserialization is converting stream of byte to object. Deserialization is the reverse process of serialization. It is the process of getting back the serialization object (so that) it can be loaded into memory. It lives the state of the object by setting properties, fields etc.



What are different types of Serialization techniques?

Different Types of Serialization

The Microsoft .NET Framework provides an almost bewildering variety of ways to serialize an object. This chapter focuses on XML serialization, but before we get into the details, I'd like to briefly examine and compare the various serialization methods offered.

XML Serialization

XML serialization allows the public properties and fields of an object to be reduced to an XML document that describes the publicly visible state of the object. This method serializes only public properties and fields—private data will not be persisted, so XML serialization does not provide full fidelity with the original object in all cases. However, because the persistence

format is XML, the data being saved can be read and manipulated in a variety of ways and on multiple platforms.

The benefits of XML serialization include the following:

Allows for complete and flexible control over the format and schema of the XML produced by serialization.

Serialized format is both human-readable and machine-readable.

Easy to implement. Does not require any custom serialization-related code in the object to be serialized.The XML Schema Definition tool (xsd.exe) can generate an XSD Schema from a set of serializable classes, and generate a set of serializable classes from an XSD Schema, making it easy to programmatically consume and manipulate nearly any XML data in an object-oriented (rather than XML-oriented) fashion.

Objects to be serialized do not need to be explicitly configured for serialization, either by the SerializableAttribute or by implementing the ISerializable interface.

The restrictions of XML serialization include the following:

The class to be serialized must have a default (parameterless) public constructor.

Read-only properties are not persisted.

Only public properties and fields can be serialized.

SOAP Serialization

SOAP serialization is similar to XML serialization in that the objects being serialized are persisted as XML. The similarity, however, ends there. The classes used for SOAP serialization reside in theSystem.Runtime.Serialization namespace rather than the System.Xml.Serializationnamespace used by XML serialization. The run-time serialization classes (which include both the SoapFormatter and the BinaryFormatter classes) use a completely different mechanism for serialization than the XmlSerializer class.

The benefits of SOAP serialization include the following:

Produces a fully SOAP-compliant envelope that can be processed by any system or service that understands SOAP.Supports either objects that implement the ISerializable interface to control their own serialization, or objects that are marked with the SerializableAttribute attribute.

Can deserialize a SOAP envelope into a compatible set of objects.

Can serialize and restore non-public and public members of an object.

The restrictions of SOAP serialization include the following:

The class to be serialized must either be marked with the SerializableAttributeattribute, or must implement the ISerializable interface and control its own serialization and deserialization.

Only understands SOAP. It cannot work with arbitrary XML schemas.

Binary Serialization

Binary serialization allows the serialization of an object into a binary stream, and restoration from a binary stream into an object. This method can be faster than XML serialization, and the binary representation is usually much more compact than an XML representation. However, this performance comes at the cost of cross-platform compatibility and human readability.

The benefits of binary serialization include the following:

It's the fastest serialization method because it does not have the overhead of generating an XML document during the serialization process.

The resulting binary data is more compact than an XML string, so it takes up less storage space and can be transmitted quickly.

Supports either objects that implement the ISerializable interface to control its own serialization, or objects that are marked with the SerializableAttribute attribute.

Can serialize and restore non-public and public members of an object.

The restrictions of binary serialization include the following:

The class to be serialized must either be marked with the SerializableAttributeattribute, or must implement the ISerializable interface and control its own serialization and deserialization.

The binary format produced is specific to the .NET Framework and it cannot be easily used from other systems or platforms.

The binary format is not human-readable, which makes it more difficult to work with if the original program that produced the data is not available.

Program for binary serialization:

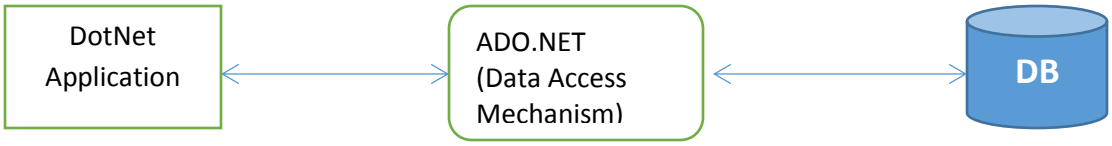
```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

namespace ConsoleApplication1
{
   class Program
   {
      public static void SerializeData()
      {
         string str = "hello world.";
         // Create file to save the data.
         FileStream fs = new FileStream(@"D:\MyDataFile.txt", FileMode.Create);
         // BinaryFormatter object will perform the serialization
         BinaryFormatter bf = new BinaryFormatter();
         // Serialize() method serializes the data to the file
         bf.Serialize(fs, str);
         // Close the file
         fs.Close();
      }
      public static void DeSerializeData()
      {
         // Open file to read the data
         FileStream fs = new FileStream(@"D:\MyDataFile.txt", FileMode.Open);
         // BinaryFormatter object performs the deserialization
         BinaryFormatter bf = new BinaryFormatter();
         // Create the object to store the deserialized data
         string data = "";
         data = (string)bf.Deserialize(fs);
          fs.Close(); //close file
         // Display the deserialized strings
         Console.WriteLine("Your deserialize data is ");
         Console.WriteLine(data);
      }
      static void Main(string[] args)
      {
         SerializeData();
         DeSerializeData();
```

```
        Console.ReadLine();
    }
} }
```

What does ActiveX Data Object.NET (ADO.NET) mean?

ActiveX Data Object.NET (ADO.NET) is a software library in the .NET framework consisting of software components providing data access services.



ADO.NET is designed to enable developers to write managed code for access to data sources, which can be relational or non-relational (such as XML or application data). This feature of ADO.NET helps to create data-sharing, distributed applications. ADO.NET provides connected access to a database connection using the .NET-managed providers and disconnected access using datasets, which are applications using the database connection only during retrieval of data or for data update. Dataset is the component helping to store the persistent data in memory to provide disconnected access for using the database resource efficiently and with better scalability.

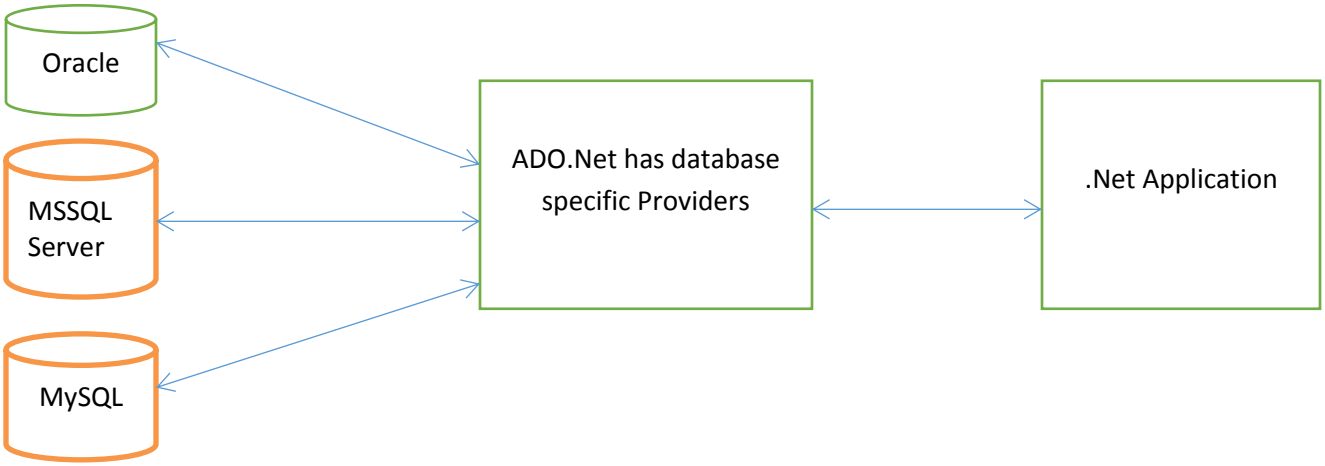
The architecture of ADO.NET is based on two primary elements:
.NET framework data provider and DataSet

A .NET data provider is a software library consisting of classes that provide data access services such as connecting to a data source, executing commands at a data source and fetching data from a data source with support to execute commands within transactions. It resides as a lightweight layer between data source and code, providing data access services with increased performance.

The .NET data provider is a component of ADO.NET, a subset of the .NET framework class library. The ADO.NET data access mode is designed such that the data set object can be used to represent an in-memory, relational structure with built-in XML support that can exist in a standalone, disconnected manner with its data, which can be passed through various layers of a multitier application. ADO.NET provides a set of interfaces to implement a custom .NET provider for specific data access needs, such as easier maintenance and better performance.

A .NET data provider makes it possible to process data directly in the data source or data stored in data sets, allowing for manipulation by the user. Data from various sources can also be combined, or passed between tiers of the application.

A .NET data provider serves as a channel to retrieve and update data existing in the data store irrespective of data sources. ADO.Net already has given providers that are database specific.



A .NET data provider consists of the following core objects:

- The Connection object is used to connect to a specific data source
- The Command object executes a command against a data source
- DataReader reads data from the data source in read-only, forward-only mode
- DataAdapter populates a data set and resolves an update with the data source

A .NET data provider abstracts the database's interaction with the application and therefore simplifies application development. However, to achieve the best performance of an application together with capability and integrity, the right .NET data provider has to be selected based on factors like design, the data source of the application, application type (middle or single tier), etc.

DataSet

The huge mainstream of applications built today involves data manipulation in some way -- whether it be retrieval, storage, change, translation, verification, or transportation. For an application to be scalable and allow other apps to interact with it, the app will need a common mechanism to pass the data around. Ideally, the vehicle that transports the data should contain the base data, any related data, and metadata, and should be able to track changes to the data. Here's where the ADO.NET DataSet steps in. The ADO.NET DataSet is a data construct that can contain several relational rowsets, the relations that link those rowsets, and the metadata for each rowset. The DataSet also tracks which fields have changed, their new values and their original values, and can store custom information in its Extended Properties collection. The DataSet can be exported to XML or created from an XML document, thus enabling increased

interoperability between applications.

When we look at the DataSet object model, we see that it is made up of three collections; Tables, Relations, and ExtendedProperties. These collections make up the relational data structure of the DataSet. The DataSet.Tables property is a DataTableCollection object, which contains zero or more DataTable objects. Each DataTable represents a table of data from the data source. Each DataTable is made of a Columns collection and a Rows collection, which are zero or more DataColumns or DataRows.

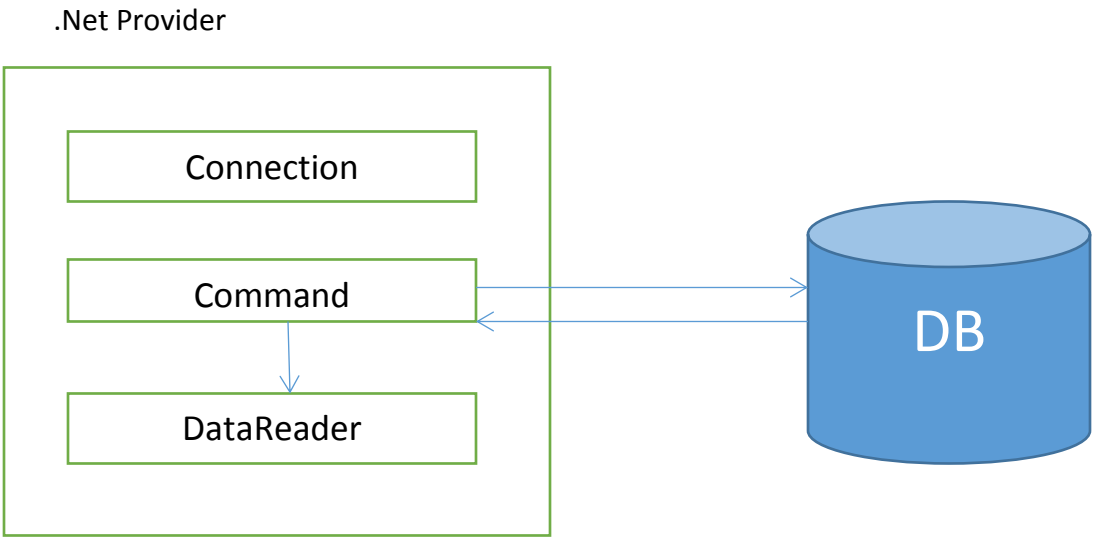
The Relations property as a DataRelationCollection object, which contains zero or more DataRelation objects. The DataRelation objects define a parent-child relationship between two tables based on foreign key values. On the other hand, The ExtendedProperties property is a PropertyCollection object, which contains zero or more user-defined properties. The ExtendedProperties collection can be used contains zero or more user-defined properties. This property collection can be used to store custom data related to the DataSet, such as the time when the DataSet was constructed.

One of the key points to remember about the DataSet is that it doesn't care where it originated. Unlike the ADO 2.x Recordset, the DataSet doesn't track which database or XML document its data came from. In this way, the DataSet is a standalone data store that can be passed from tier to tier in an n-tiered architecture.

There are two ways of Data access mechanisms.

1. Connected Architecture
2. Disconnected Architecture

Connected Architecture



Connection Oriented architecture is achieved by the use of Connection, Command and DataReader object.

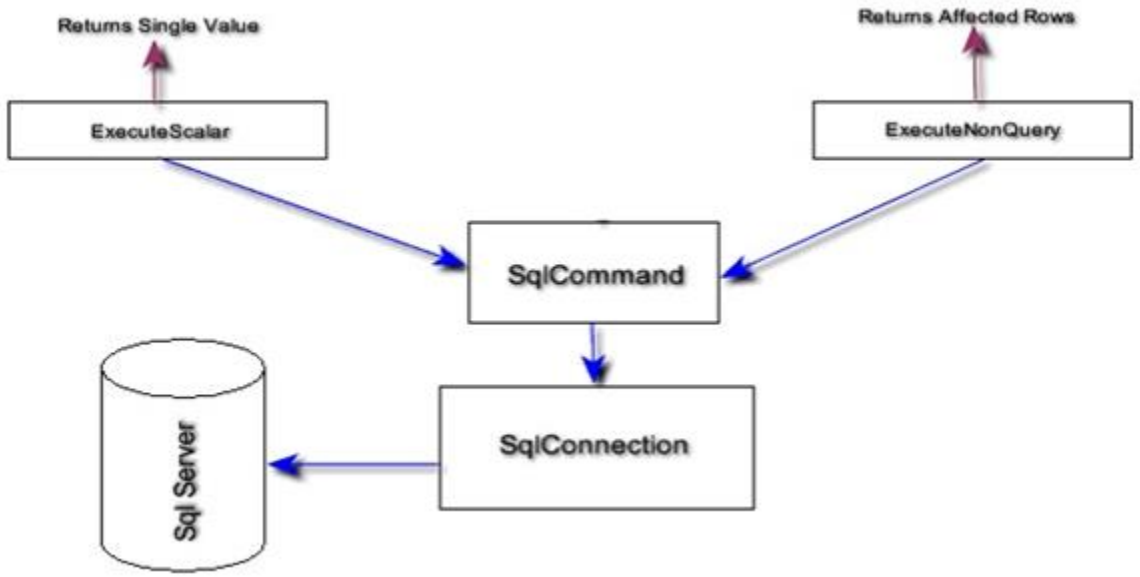
The Connection objects define the data provider, database manager instance, database, security credentials, and other connection-related properties.

Important methods of connection object
Open() - Opens a database connection with the property settings specified by the ConnectionString.
Close() - Closes the database connection.

The Command object works with the Connection object and used to execute SQL queries or Stored Procedures against the data source. You can perform insert, update, delete, and select operations with this object. It requires an instance of a Connection Object for executing the SQL statements as example.

Steps for executing SQL query with command object

- Create a Connection Object and initialize with connection string.
- Create the command object and initialize with SQL query and connection object.
- Open the connection.
- Execute query with the help of any one method of command object.
- Store the result in DataReader object (In case of select SQL query)
- Close the connection.



Important method of command object

ExecuteReader: This method works on select SQL query. It returns the DataReader object. Use DataReader read () method to retrieve the rows.

ExecuteScalar: This method returns single value. Its return type is Object. When you want single value (First column of first row), then use ExecuteScalar () method. Extra columns or rows are ignored. ExecuteScalar method gives better performance if SQL statements have aggregate function.

ExecuteNonQuery: If you are using Insert, Update or Delete SQL statement then use this method. Its return type is Integer (The number of affected records).

ExecuteXMLReader: It returns an instance of XmlReader class. This method is used to return the result set in the form of an XML document.

Important properties of Command class

- Connection
- CommandText
- CommandType
- CommandTimeout

CommandType is an important property of Command class. This property is used to determine which type of command being executed. It supports the following enumerators.

CommandType.StoredProcedure: It informs the Command object that the stored procedure will be used in place of simple SQL statements.

CommandType.Text: It informs the Command object that the CommandText value contains a SQL query.

CommandType.TableDirect: It indicates that the CommandText property contains the name of a table.

Using DataReader object
DataReader object works in connected mode. It is read only and forward only object. It is fast compare to DataSet. DataReader provides the easy way to access the data from database. It can increase the performance of application because it reads records one by one. Use read() method of DataReader to access the record.

For initializing the DataReader object, call the ExecuteReader method of the Command object. It returns an instance of the DataReader.

```
SqlCommand cmd = new SqlCommand("Your SQL query", conn);
SqlDataReader readerObj = cmd.ExecuteReader();
```

Once your task completed with the data reader, call Close() method to close the dataReader.

```
readerObj.Close();
```

Important properties of DataReader object

FieldCount: It provides the number of columns in the current row.

HasRows: It provides information that, whether data reader contains row or not.

IsClosed: It indicates that whether data reader is closed or not.

RecordsAffected: Returns the number of affected records.

You can also get the value of particular column of the table by using the data reader object.

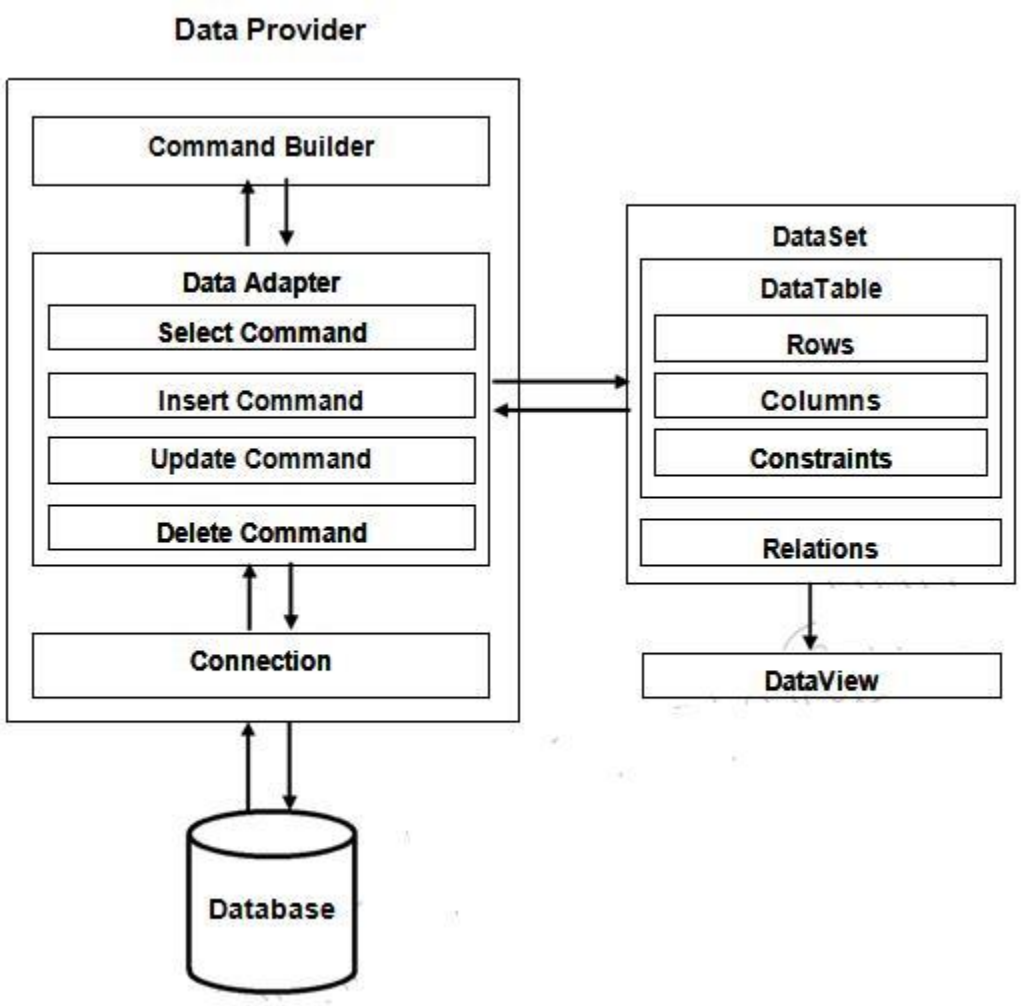
Disconnected Architecture

A connected mode of operation in ADO.Net is one in which the connection to the underlying database is alive throughout the lifetime of the operation. Meanwhile, a disconnected mode of

operation is one in which ADO.Net retrieves data from the underlying database, stores the data retrieved temporarily in the memory, and then closes the connection to the database.
The architecture of ADO.net in which data retrieved from database can be accessed even when connection to database was closed is called as disconnected architecture. Disconnected architecture of ADO.net was built on classes connection, dataadapter, commandbuilder and dataset and dataview.

Connection : Connection object is used to establish a connection to database and connectionit self will not transfer any data.

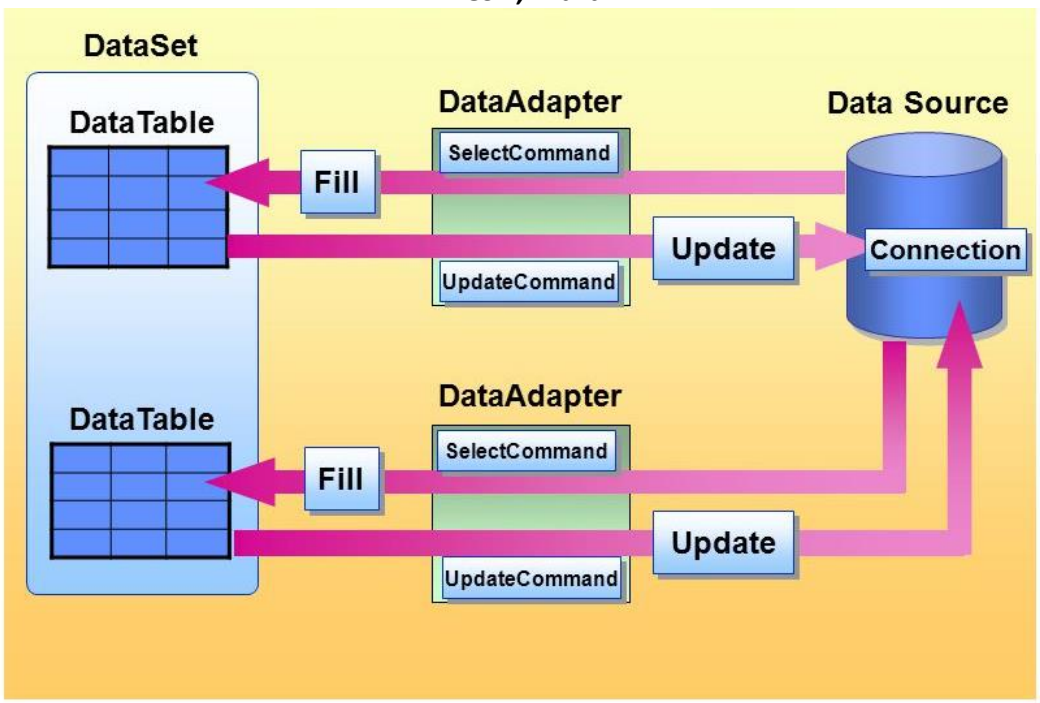
DataAdapter : DataAdapter is used to transfer the data between database and dataset. It has commands like select, insert, update and delete. Select command is used to retrieve data from database and insert, update and delete commands are used to send changes to the data in dataset to database. It needs a connection to transfer the data.



CommandBuilder : by default dataadapter contains only the select command and it doesn'tcontain insert, update and delete commands. To create insert, update and delete commands for the dataadapter, commandbuilder is used. It is used only to create these commands for the dataadapter and has no other purpose.

DataSet : Dataset is used to store the data retrieved from database by dataadapter and make it available for .net application.
To fill data in to dataset **fill()** method of dataadapter is used and has the following syntax.
Da.Fill(Ds,"TableName");
When fill method was called, dataadapter will open a connection to database, executes select command, stores the data retrieved by select command in to dataset and immediately closes the connection.
As connection to database was closed, any changes to the data in dataset will not be directly sent to the database and will be made only in the dataset. To send changes made to data in dataset to the database, **Update()** method of the dataadapter is used that has the following syntax.
Da.Update(Ds,"Tablename");

When Update method was called, dataadapter will again open the connection to database, executes insert, update and delete commands to send changes in dataset to database and immediately closes the connection. As connection is opened only when it is required and will be automatically closed when it was not required, this architecture is called disconnected architecture. A dataset can contain data in multiple tables.

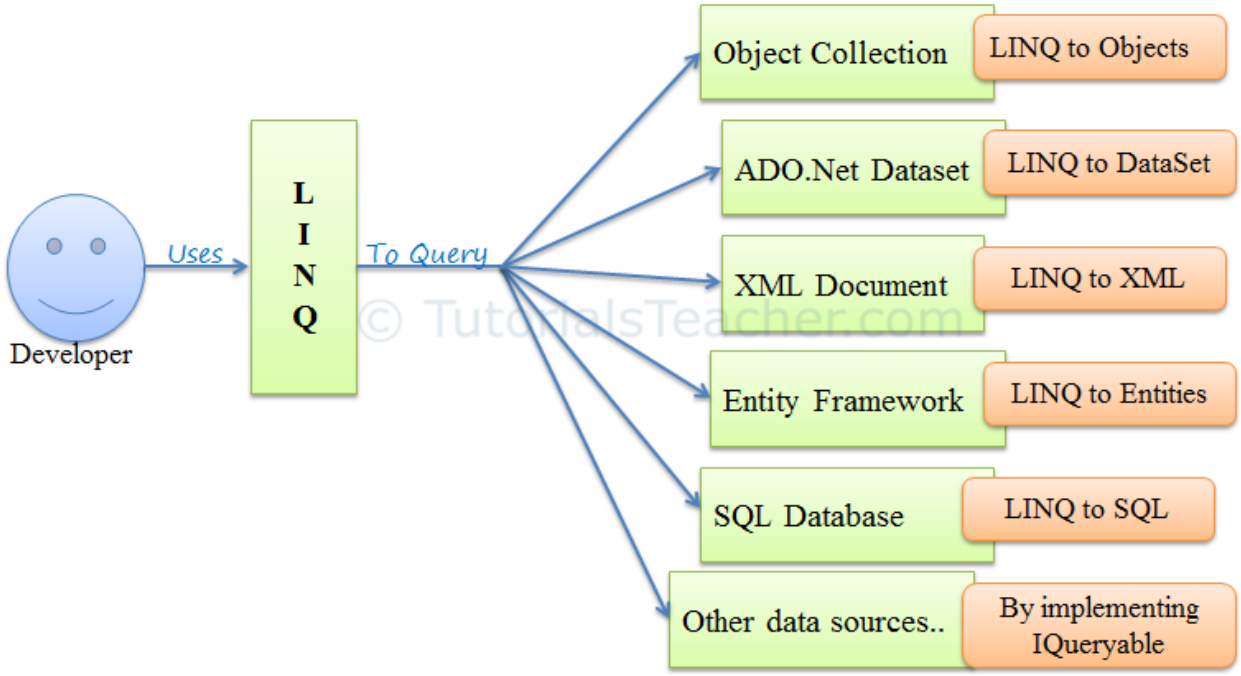


Difference between Connected and Disconnected Architecture

	Connected	Disconnected
Classes used	**Connection, Data Reader, Command**	**Connection, Command Builder, DataSet, DataAdapter, DataView**
Performance	Provide **faster** performance and speed.	Provides **slower** performance and speed when compared to connected arch.
Data Storage	DataReader can store only **one result Set**	DataSet Can store **multiple result sets.**
Data Accessing	**DataReader** object	**DataSet** object
Data Update	**DataReader** Can't update data directly, it follows read only manner.	Can update data using **Update ( )** of **DataAdapter**.
Data Persistence	Data Reader **can't persist** the data	Data Set **can persist** the data.
Data loosing	Data will not available once connection is lost.	**Data still available** in DataSet object even connection is lost.
No of Lines of code	No of lines of code will be more when using **DataReader** for accessing data.	Very few lines of code will be used for accessing data using DataSet.
Memory Usage	DataReader uses **less memory** and **system resources**	DataSet uses **more memory** and **system resources** when compared to DataReader.
Serializing	Serializing to XML or JSON format will become **complex** when using DataReader result.	Can **quickly serialize** DataSet to XML using **WriteXML( )**
Fetching huge data	**DataReader is most preferable** when trying to fetch millions or billions of records.	**Don't prefer DataSet** when fetching huge data.

What is LINQ?

LINQ is an acronym for Language Integrated Query, which is descriptive for where it's used and what it does. The *Language Integrated* part means that LINQ is part of programming language syntax. In particular, both C# and VB are languages that ship with .NET and have LINQ capabilities. Another programming language that supports LINQ is Delphi Prism. The other part of the definition, *Query*, explains what LINQ does; LINQ is used for querying data. Notice that I used the generic term "data" and didn't indicate what type of data. That's because LINQ can be used to query many different types of data, including relational, XML, and even objects. Another way to describe LINQ is that it is programming language syntax that is used to query data.



LINQ queries return results as objects. It enables you to uses object-oriented approach on the result set and not to worry about transforming different formats of results into objects.

We Already have ADO.NET, so Why Another Data Access Technology?

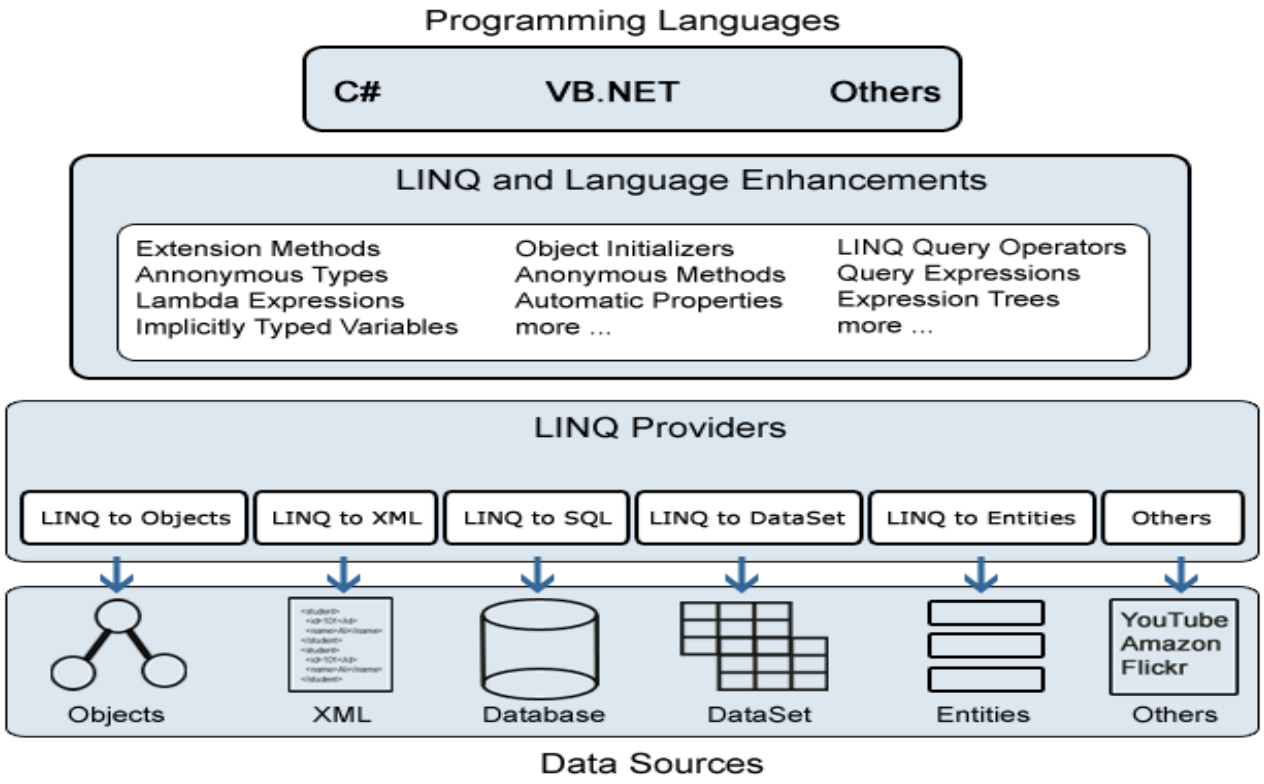
Most applications work with data in one form or another, meaning that data is very important to the work we do as software engineers. It's so important that the tools we use are constantly evolving, with the next generation building and improving upon the previous. This doesn't change with LINQ, which is the next giant leap in data development technology beyond ADO.NET.

ADO.NET is an object-oriented library, yet we must still reason about data from a relational perspective. In simple scenarios, we can bind ADO.NET objects directly to user interfaces (UI), but many other situations require the translation of the ADO.NET data into business objects with relationships, rules, and semantics that don't translate automatically from a relational data store. For example, a relational data store will model Orders and Customers with a foreign key from an Order table to a Customer table, but the object representation of this data is a Customer object with a collection of Order objects. Similar situations occur for other storage types such as hierarchical, multi-value, and flat data sources. This gap between the representation of data from the storage site to the objects you use in your applications is called an *Impedance Mismatch*. While ADO.NET is an object library for working with relational data, LINQ is a SQL-like syntax that produces usable objects. LINQ helps reduce this Impedance Mismatch.

Note: The operative term in the previous paragraph is "reduce". LINQ does not eliminate Impedance Mismatch, because you must still reason about your data store format. However, LINQ does remove a lot of the plumbing work you have to do to re-shape your data as an object.

More about Data Sources

LINQ is intended to make it easy to query data sources. One of the more popular uses of LINQ is to query relational databases. However, as you see here, LINQ can query objects. That's not all, the .NET Framework includes libraries that allow anyone to create a LINQ provider that can query any data source. Out of the box, Microsoft ships LINQ to Objects, LINQ to XML, LINQ to SQL (SQL Server), and LINQ to Entities (Entity Framework). There are also 3rd party LINQ providers that make it easy to query specialized data sources



In the query, we select elements from an array in descending order (high to low). We filter out elements <= 2. In the loop, we evaluate the expression and print the results.**Var**

C# program that uses query expression

```
using System;
using System.Linq;


class Program
{
   static void Main()
   {
      int[] array = { 1, 2, 3, 6, 7, 8 };
```

```
    // Query expression.
    var elements = from element in array
                   orderby element descending
                   where element > 2
                   select element;
    // Enumerate.
    foreach (var element in elements)
    {
      Console.Write(element);
      Console.Write(' ');
    }
    Console.WriteLine();
  }
}
```

Output

```
8 7 6 3
```

C# program that removes duplicate elements

```
using System;
using System.Linq;
class Program
{
  static void Main()
  {
    // Declare an array with some duplicated elements in it.
    int[] array1 = { 1, 2, 2, 3, 4, 4 };
    // Invoke Distinct extension method.
    var result = array1.Distinct();
    // Display results.

foreach (int value in result)
```

```
    {
        Console.WriteLine(value);
    }
  }
}
```

Output

```
1
2
3
4
```

C# program that uses Contains

```
using System;
using System.Collections.Generic;
using System.Linq;
class Program
{
  static void Main()
  {
    var list = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    // Use extension method.
    bool a = list.Contains<int>(7);
    // Use instance method.
    bool b = list.Contains(7);
    Console.WriteLine(a);
    Console.WriteLine(b);
  }
}
```

Output True True