CMR
INSTITUTE OF
TECHNOLOGY

Internal Assessment Test 2 – December 2021

Sub:	Programming using C#.NET							Code:	18MCA51
Date:	16-12-21	Duration:	90 mins	Max Marks:	50	Sem:	V	Branch:	MCA

Note: Answer any 5 questions. All questions carry equal marks. Total marks: 50

1. How method overloading is different from overriding . illustrate with an example

In **overriding**, a child class can implement the parent class method in a different way but thechild class method has the same name and same method signature as parent whereas in **overloading** there are multiple methods in a class with the same name and differentparameters.

Overloading

```
1. class Program
2.     {
3.         public int Add(int num1, int num2)
4.         {
5.             return (num1 + num2);
6.         }
7.         public int Add(int num1, int num2, int num3)
8.         {
9.             return (num1 + num2 + num3);
10.        }
11.        public float Add(float num1, float num2)
12.        {
13.            return (num1 + num2);
14.        }
15.        public string Add(string value1, string value2)
16.        {
17.            return (value1 + " " + value2);
18.        }
19.        static void Main(string[] args)
20.        {
21.            Program objProgram = new Program();
22.            Console.WriteLine("Add with two int parameter :" + objProg
   ram.Add(3, 2));
23.            Console.WriteLine("Add with three int parameter :" + objPr
   ogram.Add(3, 2, 8));
24.            Console.WriteLine("Add with two float parameter :" + objPr
   ogram.Add(3 f, 22 f));
25.            Console.WriteLine("Add with two string parameter :" + objP
   rogram.Add("hello", "world"));
26.            Console.ReadLine();
27.        }
28.    }
```

Overriding

Method Overriding is a type of polymorphism. It has several names like “Run Time Polymorphism” or “Dynamic Polymorphism” and sometime it is called “Late Binding”.

Method Overriding means having two methods with same name and same signatures [parameters], one should be in the base class and other method should be in a derived class [child class]. You can override the functionality of a base class method to create a same namemethod with same signature in a derived class. You can achieve method overriding using inheritance. Virtual and Override keywords are used to achieve method overriding.

```
1. class BaseClass
2.     {
3.         public virtual int Add(int num1, int num2)
4.         {
5.             return (num1 + num2);
6.         }
7.     }
8.     class ChildClass: BaseClass
9.     {
10.        public override int Add(int num1, int num2)
11.        {
12.            if (num1 <= 0 || num2 <= 0)
13.            {
14.                Console.WriteLine("Values could not be less than zero
   or equals to zero");
15.                Console.WriteLine("Enter First value : ");
16.                num1 = Convert.ToInt32(Console.ReadLine());
17.                Console.WriteLine("Enter First value : ");
18.                num2 = Convert.ToInt32(Console.ReadLine());
19.            }
20.            return (num1 + num2);
21.        }
22.    }
23.    class Program
24.    {
25.        static void Main(string[] args)
26.        {
27.            BaseClass baseClassObj;
28.            baseClassObj = new BaseClass();
29.            Console.WriteLine("Base class method Add :" + baseClassObj
   .Add(-3, 8));
30.            baseClassObj = new ChildClass();
31.            Console.WriteLine("Child class method Add :" + baseClassOb
   j.Add(-2, 2));
32.            Console.ReadLine();
33.        }
34.    }
```

2. Explain in detail about run time polymorphism and method hiding with relevant example

Overriding is a feature that allows a derived class to provide a specific implementation of a method that is already defined in a base class. The implementation of method in the derived class overrides or replaces the implementation of method in its base class. This feature is also known as runtime polymorphism because the compiler binds a method to an object during the execution of a program (runtime) instead of during the compilation of the program.

When a method get called, the method defined in the derived class is invoked and executed instead of the one in the base class.

To invoke the method of a derived class that is already defined in the base class, you need to perform the following steps:

- Declare the base class method as virtual
- Implement the derived class method using the override keyword

We have created Bclass and Dclass classes, where the Dclass class is the derived class of the Bclass class. The Bclass class contain virtual method Show(). The Dclass class is overriding
the Show() method of the base class by using the override keyword.

Example:

```
using System;
namespace Chapter4_Examples{
public class Bclass{
public virtual void Show(){
Console.WriteLine("Base Class");
}
}
class Dclass : Bclass{
public override void Show(){
Console.WriteLine("Derived Class");
}
}
class PolyDemo{
static void Main(string[] args){
// calling the overriden method
Dclass objDc = new Dclass();
objDc.Show();
// calling the base class method
Bclass objBc = new Bclass();
objBc.Show();
//Calling the overriden method because preference
will be right hand of assignment operator
Bclass obj = new Dclass();
obj.Show();
Console.ReadLine();
}
}
}
```

Consider, you want to derive a class from a base class and to redefine some methods contained in this base class. In such a case, you cannot declare the base class method as *virtual*.

Method Hiding:

Then, how you can override a method without declaring that method as virtual in the base class?

Ans: This can be possible with the new operator. The new operator is used to override the base class method in the derived class without using the virtual keyword. The new operator tells the compiler that the derived class method hides the base class method

Example:

```
using System;
namespace Class_Demos{
public class Bclass{
public void Show(){
Console.WriteLine("Base Class");
}
}
class Dclass : Bclass{
public new void Show(){
Console.WriteLine("Derived Class");
}
}
class PolyDemo1{
static void Main(string[] args){
// calling the overriden method
Dclass objDc = new Dclass();
objDc.Show();
```

```
// calling the base class method
Bclass objBc = new Bclass();
objBc.Show();
//Calling Base Class Method
objBc = new Dclass();
objBc.Show();
Console.ReadLine();
}
}
}
```

3 . Explain in detail about the following.

1. Sealed class & Method 2.Extension methods

Sealed class & Method

Sealed classes are classes that cannot be inherited. You can use the ***"sealed"*** keyword to define a class as a sealed class

```
using System;
sealed public class Animal{
 public void eat() { Console.WriteLine("eating..."); }
}
public class Dog: Animal
{
   public void bark() { Console.WriteLine("barking..."); }
}
public class TestSealed
{
   public static void Main()
  {
     Dog d = new Dog();
     d.eat();
    d.bark();


   }
}
```

Sealed method

```
using System;
public class Animal{
 public virtual void eat() { Console.WriteLine("eating..."); }
 public virtual void run() { Console.WriteLine("running..."); }

}
public class Dog: Animal
{     public override void eat() { Console.WriteLine("eating bread..."); }
  public sealed override void run() {
  Console.WriteLine("running very fast...");
```

```
  } }
public class BabyDog : Dog
{
   public override void eat() { Console.WriteLine("eating biscuits..."); }
   public override void run() { Console.WriteLine("running slowly..."); }
}
public class TestSealed
{
   public static void Main()
   {
      BabyDog d = new BabyDog();
      d.eat();
     d.run();
   }
}
```

Extension method

Extension method is a method that helps you to extend a class without creating a new derived class or modifying the original class. It works as a *static* method, but is invoked with an instance of the extended class. The extension method can be any static method which uses the "this" keyword before its first parameter.

Example:

```
using System;
namespace Class_Demos{
public static class Myclass{
//Defining extension method with
public static int myExtensionmethod(this string num){
return (Int32.Parse(num));
}
public static int Mystaticmethod(string num){
return (Int32.Parse(num));
}
}
class ExtensionDemo{
static void Main(){
string num = "100";
//invoking method of type extension
int ext = num.myExtensionmethod(); //Line A
Console.WriteLine("The output from myExtensionMethod()"+ext);
//invoking method of type static
int stat = Myclass.Mystaticmethod(num); //Line B
Console.WriteLine("The ou
Console.Read();

}
}
}
```

4.Discuss about types of inheritance in c# and explain how multiple inheritance can be achieved in C#

The next pillar of OOP, Inheritance provides you to reuse existing code and fast implementation time. The relationship between two or more classes is termed as *Inheritance*.

In essence, inheritance allows to extend the behavior of a base (or parent/super) class by enabling a subclass to

inherit core functionality (also called a derived class/child class). All public or protected variables and methods
in the base class can be called in the *derived classes*.
Inheritance comes in two ways:
Classical inheritance (“is-a” relationship)
Containment/delegation model (“has-a” relationship).
Classical inheritance (“is-a” relationship):
When “is-a” relationship have established between classes, we are building a dependency between types. The basic idea behind classical inheritance is that new classes may extend the functionality of other classes.

Containment / Delegation model (“Has-A”):
The “HAS-A” relationship specifies how one class is made up of other classes.
Example 2.7: Consider we have two different classes Engine and a Car when both of these entities share each other’s object for some work and at the same time they can exists without each other’s dependency (having their
own life time) and there should be no single owner both have to be an independent from each other than type of relationship is known as "has-a" relationship i.e. Association.
Inheritance is of four types, which are as follows:
i. Single Inheritance: Refers to inheritance in which there is only one base class and one derived class. This means that a derived class inherits properties from single base class.
ii. **Hierarchical Inheritance:** Refers to inheritance in which multiple derived classes are inherited from the same base class.
iii. **Multilevel Inheritance:** Refers to inheritance in which a child class is derived from a class, which in turn is derived from another class.
iv. **Multiple Inheritance:** Refers to inheritance in which a child class is derived from multiple base class.
C# supports single, hierarchical, and multilevel inheritance because there is only a single base class. It does not
support multiple inheritance directly. To implement multiple inheritance, you need to use *interfaces*.
Example:

```
using System;
namespace Class_Demos{
class BaseClass{
public int dm;
public void BCMethod(){
Console.WriteLine("I'm a Base Class Method");
}
}
class DerivedClass:BaseClass{
public void DCMethod(){
Console.WriteLine("I'm a Derived Class Method");
}
}
class InherDemo{
static void Main(){
//Create a Base Class Object
Console.WriteLine("I'm accessing Base Class Object");
BaseClass bc = new BaseClass();
bc.dm = 10;
bc.BCMethod();
//Create a Derived Class Object
Console.WriteLine("I'm accessing Derived Class Object");
DerivedClass dc=new DerivedClass();
dc.dm = 20;
dc.BCMethod();
dc.DCMethod();
Console.WriteLine("\nPress ENTER to quit...");
Console.Read();
```

```
}
 }}
```

Interfaces:

"An interface is a collection of data members and member functions, but it does not implement them".

Interface are introduced to provide the feature of multiple inheritance to classes.

Syntax:

```
interface <Interface_Name> {
//Abstract method declaration in interface body
 }
```

Characteristics:

- An interface is always implemented in a class.
- The class that implements the interface needs to implement all the members of the interface.
- Cannot instantiate an object through an interface
- An interface can have the same access modifiers as a class, such as public and private.

Example:

```
using System;
namespace Class_Demos{
public interface Channel{
void Next();
void Previous();
}
public interface Book{
void Next();
void Chapter();
 }
class InterfaceDemo:Channel, Book{
void Channel.Next(){
Console.WriteLine("Channel Next");
}
void Book.Next(){
Console.WriteLine("Book Next");
}
public void Previous(){
Console.WriteLine("Previous");
}
public void Chapter(){
Console.WriteLine("Chapter");
}
static void Main(){
InterfaceDemo ind = new InterfaceDemo();
((Book)ind).Next();//invoking Book method
ind.Previous();
ind.Chapter();
Console.Read();
}
}
 }
```

5. How delegates are used in C#? Discuss single cast and multicast delegates with an example.

A delegate is an object which refers to a method or you can say it is a reference type variablethat can hold a reference to the methods. Delegates in C# are similar to the function pointer in C/C++. It provides a way

which tells which method is to be called when an event is triggered.
For example, if you click an Button on a form (Windows Form application), the programwould call a specific method. In simple words, it is a type that represents references to methods with a particular parameter list and return type and then calls the method in a program for execution when it is needed.
A Delegate can be defined as a delegate type. Its definition must be similar to the functionsignature. A delegate can be defined in a namespace and within a class.
A delegate cannot be used as a data member of a class or local variable within a method. Delegate declarations look almost exactly like abstract method declarations, you just replacethe abstract keyword with the delegate keyword.
Delegates are especially used for implementing events and the call-back methods. Alldelegates are implicitly derived from the System. Delegate class.

In C#, delegate is a reference to the method. It works like function pointer in C and C++. But it is objected-oriented, secured and type-safe than function pointer.

For static method, delegate encapsulates method only. But for instance method, itencapsulates method and instance both.

The best use of delegate is to use as event.

Internally a delegate declaration defines a class which is the derived classof System.Delegate.

Singlecast Delegate

This is a kind of delegate that can refer to single method at one time. SingleCast Delegates refer to a single method with matching signature. SingleCast Delegates derive from the System.Delegate class.

Single cast delegate programusing System;
using System.Collections.Generic;
using System.Linq;using
System.Text;
using System.Threading.Tasks;
namespace delegatefunction
{
  // Delegate definition
  public delegate int delefunc(int x, int y);

  class Program
  {
    static int add(int a, int b)
    {
      return a + b;
    }
    public static void Main(string[] s)
    {
      // instantiate the delegate
      // delegatename obj = new delegatename(classname.methodname)delefunc d1 =
      new delefunc(Program.add);
      // pass the values and print output

      Console.WriteLine("Addition of numbers = {0}", d1(20, 30));
      Console.ReadKey();
    }
  }
}

Multicast Delegate

A delegate that holds a reference to more than one method is called multicasting delegate. A Multicast Delegate is a delegate that holds the references of more than one function. When we invoke the multicast delegate, then all the functions which are referenced by the delegate are going to be invoked. If you want to call multiple methods using a delegate then all the method signature should be the same.

Multicast Delegate Program using System;
using System.Collections.Generic;using
System.Linq;
using System.Text;
using System.Threading.Tasks;

```
namespace multicastedelexamole
{
   // declaring the delegate
   public delegate void MyDel(int num1, int num2);class Sample
   {
      // Method Add is the first method called by the delegate MyDelstatic void
  Add(int num1, int num2)
      {
         Console.WriteLine("\tAddition: " + (num1 + num2));
      }
           // Method Sub is the second method called by the delegate MyDelstatic void
      Sub(int num1, int num2)
      {
         Console.WriteLine("\tSubtraction: " + (num1 - num2));
      }
// Method Mul is the third method called by the delegate MyDel
      static void Mul(int num1, int num2)
      {
         Console.WriteLine("\tMultiplication: " + (num1 * num2));
      }

      static void Main()
      {
         int num1 = 0;int num2 = 0;
                  // instantiating the delegate with first method as parameterMyDel del = new
         MyDel(Add);
                  // input the values to be passed as argumentsConsole.Write("Enter the value
         of num1: ");
         num1 = int.Parse(Console.ReadLine());

         Console.Write("Enter the value of num2: ");num2 =
         int.Parse(Console.ReadLine());
                  // second method is appended to the delegate objectdel += new
         MyDel(Sub);
                  // third method is appended to the delegate objectdel += new
         MyDel(Mul);

         Console.WriteLine("Call 1:");
// the methods will be executed one after the other and output will be displayed.del(num1, num2);

         Console.ReadKey();
      }
```

```
    }
}
```

6. **Explain in detail about Events? Discuss single cast and multicast Events with an example.**

C# - Events

An event is a notification sent by an object to signal the occurrence of an action. Events in .NET follow the observer design pattern.

The class who raises events is called Publisher, and the class who receives the notification is called Subscriber. There can be multiple subscribers of a single event. Typically, a publisher raises an event when some action occurred. The subscribers, who are interested in getting a notification when an action occurred, should register with an event and handle it.

In C#, an event is an encapsulated delegate. It is dependent on the delegate. The delegate defines the signature for the event handler method of the subscriber class.

Event Sources and Event Handler:

When you talk about events, the equation always has **two tasks** that as follows below.
i. An **event source** pays attention to firing events and then detects the timing when an event should be fired.
ii. The **event handler** deals with receiving the information that an event has fired and then verifies the
information that the event is present.
Event Source: Event source is an object that inform other objects or tasks that something has changed.
Notification in C# takes the form of callbacks.

- The *event source* sends out a general message when it wants to notify other objects about a change, and then any object interested in the message can read and interpret it.
- The *event handling* is in the form of publish–and–subscribe, which requires a generalized broadcast mechanism and that all objects receive all messages.
- Each event source maintains all the events published by it.
- Events are considered to be asynchronous from the perspective of the caller.
- The occurrence of a callback depends upon the *event-firing* object rather than its caller.

Event handler: receive event notifications. When an event source notifies that an event is called, all the event handlers that have registered for the notification of that event start executing. An event handler may be either a static or a non-static method of a class. In case of a static method, event handlers simply respond to a given event source and take a course of action based on the information passed to the handler.

Declare an Event

An event can be declared in two steps:

1. Declare a delegate.
2. Declare a variable of the delegate with event keyword.

The following example shows how to declare an event in publisher class.

Example: Declaring an Event

```
public delegate void Notify();  // delegate

public class ProcessBusinessLogic
{
   public event Notify ProcessCompleted; // event
```

```
}
```

In the above example, we declared a delegate Notify and then declared an event ProcessCompleted of delegate type Notify using "event" keyword in the ProcessBusinessLogic class. Thus, the ProcessBusinessLogic class is called the publisher. The Notify delegate specifies the signature for the ProcessCompleted event handler. It specifies that the event handler method in subscriber class must have a void return type and no parameters.

Now, let's see how to raise the ProcessCompleted event. Consider the following implementation.

Example: Raising an Event

```
public delegate void Notify();  // delegate

public class ProcessBusinessLogic
{
   public event Notify ProcessCompleted; // event

   public void StartProcess()
   {
      Console.WriteLine("Process Started!");
      // some code here..
      OnProcessCompleted();
   }

   protected virtual void OnProcessCompleted() //protected virtual method
   {
      //if ProcessCompleted is not null then call delegate
      ProcessCompleted?.Invoke();
   }
}
```

Above, the StartProcess() method calls the method onProcessCompleted() at the end, which raises an event. Typically, to raise an event, protected and virtual method should be defined with the name On<EventName>. Protected and virtual enable derived classes to override the logic for raising the event. However, A derived class should always call the On<EventName> method of the base class to ensure that registered delegates receive the event.

The OnProcessCompleted() method invokes the delegate using ProcessCompleted?.Invoke();. This will call all the event handler methods registered with the ProcessCompleted event.

The subscriber class must register to ProcessCompleted event and handle it with the method whose signature matches Notify delegate, as shown below.

Example: Consume an Event

```
class Program
{
   public static void Main()
   {
      ProcessBusinessLogic bl = new ProcessBusinessLogic();
      bl.ProcessCompleted += bl_ProcessCompleted; // register with an event
      bl.StartProcess();
   }

   // event handler
   public static void bl_ProcessCompleted()
   {
      Console.WriteLine("Process Completed!");
   }
}
```

Above, the Program class is a subscriber of the ProcessCompleted event. It registers with the event using += operator. Remember, this is the same way we add methods in the invocation list of multicast delegate. The bl_ProcessCompleted() method handles the event because it matches the signature of

the Notify delegate.

Multiple Event Handlers:

Like delegates, events can be multicast. This enables multiple objects to respond to an event notification, you

used the += operator to add the event handler.

Example:

MultiEventDemo.cs

```
using System;
namespace Class_Demos{
class EventTestClass{
int nvalue; //The value to track
public delegate void ValueChangedEventHandler();
public event ValueChangedEventHandler Changed;
protected virtual void onChanged(){
if (Changed != null)
Changed();
else
Console.WriteLine("Event fired. No handler");
}
public EventTestClass(int nvalue){
SetValue(nvalue);
}
public void SetValue(int nv){
if (nvalue != nv){
nvalue = nv;
onChanged(); //Fire the event
}
else
Console.WriteLine("No Fire");
}
}
class MultiEventDemo{
public void HandleChange1() {
Console.WriteLine("Handler 1 called");
}
public void HandleChange2(){
Console.WriteLine("Handler 2 called");
}
static void Main(){
EventTestClass etc = new EventTestClass(3);
MultiEventDemo med = new MultiEventDemo();
//Create a handler for this class
etc.Changed += new EventTestClass.ValueChangedEventHandler(med.HandleChange1);
etc.Changed+= new EventTestClass.ValueChangedEventHandler(med.HandleChange2);
//event detached from the object
etc.Changed−= new EventTestClass.ValueChangedEventHandler(med.HandleChange2);
etc.SetValue(5);
etc.SetValue(5);
etc.SetValue(3);
Console.WriteLine("\nPress ENTER to quit...");
Console.ReadLine();
}
}
```

}

7. Explain in detail about exception and its type along with its handling mechanism“

An exception is a problem that arises during the runtime (execution of a program) such as division of a number
by zero, passing a string to a variable that holds an integer value.
C# exception handling is built upon four keywords: **try**, **catch**, **finally**, and **throw**.

2.2 The try/catch/throw/finally statement

 try: A try block identifies a block of code for which particular exceptions is activated. It is followed by
one or more catch blocks.
 catch: A program catches an exception with an exception handler at the place in a program where you
want to handle the problem. The catch keyword indicates the catching of an exception.
 finally: The finally block is used to execute a given set of statements, whether an exception is thrown or
not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.

Example 2.1:

ExcepDemo.cs

```
using System;
namespace Class_Demos{
class ExcepDemo{
static void Main(string[] args){
int div=0, number=10;
try{
div = 100 / number;
}
catch (DivideByZeroException ex){
Console.WriteLine("Exception occured : " +ex.Message);
}
finally{
Console.WriteLine("Result is: {0}", div);
Console.WriteLine("\nPress ENTER to Quit...");
Console.ReadKey();
}
}
}
}
```

 throw: A program throws an exception when a problem shows up.
This is done using a throw keyword. The *throw* statement takes only a single argument to throw the
exception. When a throw statement is encountered, a program terminates.
In the following snippet, we have
thrown a new
DivideByZeroException explicitly.

```
try{ throw new DivideByZeroException(); }
catch{Console.WriteLine(“Exception”); }
```

Example:

ThrowStat.cs

```
using System;
namespace Class_Demos{
class ThrowStat{
static void Main(string[] args){
int number;
```

```
Console.WriteLine("Enter a number");
number = int.Parse(Console.ReadLine());
try{
if (number > 10)
throw new Exception("OutofSize");
}
catch (Exception ex){
Console.WriteLine("Exception occured : " +ex.Message);
}
finally{
Console.WriteLine("This is last statment");
Console.WriteLine("\nPress ENTER to Quit...");
Console.ReadKey();
}
}
}
}
```

Custom Exception(ApplicationException):

The **ApplicationException** is thrown by a user program, not by the common language runtime. If you are designing an application that needs to create its **own exceptions**.

To create your own exception class, here are some important recommendations:

- Give a meaningful name to your Exception class and end it with Exception.
- Throw the most specific exception possible.
- Give meaningful messages.
- Do use InnerExceptions.
- When wrong arguments are passed, throw an ArgumentException or a subclass of it, if necessary.

System.Exception:

System.**Exception** class represents an error that occurs during runtime of an application. This class is the **base class** for all exceptions.

When an error occurs, either the system or the currently executing application reports it by throwing an exception containing information about the error. Once thrown, an exception is handled by the application or by the default exception handler.

8. Write a C# Program to Illustrate Exception Handling for Invalid Typecasting in Unboxing

```
class TestUnboxing
{
   static void Main()
   {
      int num = 123;
      object obj = num;
      try
      {
         int j = (short)obj;
         System.Console.WriteLine("Unboxing");
      }
      catch (System.InvalidCastException e)
      {
```

```
            System.Console.WriteLine("{0} Error: Incorrect unboxing", e.Message);
        }
        System.Console.Read();
    }
}
```

9. . "Catching on exceptions programmatically is good and necessary mechanism" justify withsuitable examples.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace GMT_TCF
{
class Program {
static void Main(string[] args)
{
Console.WriteLine(" Enter the dividend");
int m= Convert.ToInt32(Console.ReadLine());
Console.WriteLine(" Enter the divisor");
int n = Convert.ToInt32(Console.ReadLine());
try { int k = m / n;
Console.WriteLine("Output is:" + k.ToString());
 }
catch (DivideByZeroException e) {
Console.WriteLine("Exception Caught:" + e.Message);
} finally {
Console.ReadLine();
} }
```

10. Explain the components of ADO.NET entity framework architecture.

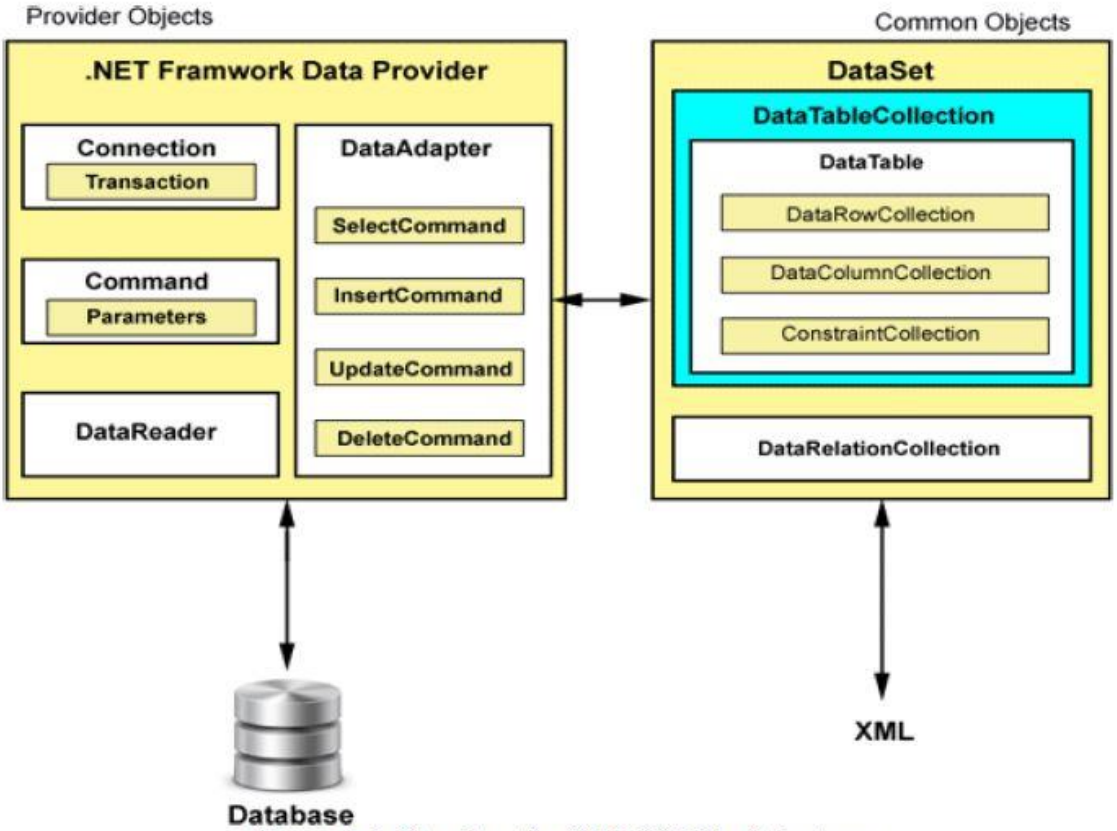

Figure 3.1: Showing the ADO.NET Architecture

ADO.NET is a data access technology from Microsoft .Net Framework, which provides communication
between relational and non-relational systems through a common set of components.
ADO.NET consist of a set of Objects that expose data access services to the .NET environment.
ADO.NET is designed to be easy to use, and Visual Studio provides several wizards and other features that can
use to generate ADO.NET data access code.
The **two key components** of ADO.NET are **Data Providers** and **DataSet**.
The .Net Framework includes mainly **three Data Providers** for ADO.NET. They are

i. Microsoft SQL Server Data Provider ii. OLEDB Data Provider iii. ODBC Data Provider

SQL Server uses the SqlConnection object, OLEDB uses the OleDbConnection Object and ODBC uses
OdbcConnection Object respectively.
The **four Objects** from the .Net Framework provides the functionality of Data Providers in the ADO.NET.
They are
i. The **Connection Object** provides physical connection to the Data Source.
ii. The **Command Object** uses to perform SQL statement or stored procedure to be executed at the Data
Source.
iii. The **DataReader Object** is a stream-based, forward-only, read-only retrieval of query results from the
Data Source, which do not update the data.
iv. The **DataAdapter Object**, which populate a Dataset Object with results from a Data Source
DataSet provides a disconnected representation of result sets from the Data Source, and it is completely
independent from the Data Source.

Data Provider in ADO.NET

A data provider is a set of related

		components that work together to provide data in an efficient manner. It is used in ADO.NET for connecting to a database, executing commands, and retrieving results. The results are either processed directly or manipulated between tiers and displayed after combining with multiple sources. The data provider increase performance without compromising on functionality.			