New Features of .NET 3.5

- Implicitly typed local variables
- Auto Implemented properties
- Object Initializer
- Collection Initializer
- Anonymous Objects
- Extension Methods
- Lambda Expressions
- LINQ[Language Integrated Query]

Implicitly Typed local variables

- To work with Implicit types .NET comes with a new key word called var
  - var is a new data type
  - var is capable to hold any type of data
  - var x=12;
  - var x="David"
- Based on the initialized value var data type will be converted into primitive type at runtime
  - var x=10=> int x=10
  - var variable need to declared in local scope only.
  - var variable doesn't works in Global Declaration
  - var variable must be initialized in the same line of declaration
  - var variable cannot be used as function arguments

Auto Implemented properties

- A collection of set & get methods with logic is called as property
- A collection of set & get methods without logic is called as Auto Implemented property
- Auto type

```
      Public data type PN
                 {
                 get;
                 set;
                 }
```

Auto Implemented props are required while working with object initializers,collection initializers & LINQ

New Features of .NET 4.0

- Object Initializers:
- Object Initializers are conceptually similar to constructors.
- Collection Initializers:
- Collection initializer is a collection of Object Initializers.
- Collection Initializer looks similar to array of objects
- To work with initializers, a predefined collection is required.

declaration:

```
        list<class obj> ob=new list<class obj>
```

Anonymous types

- Declaring type with out defining its class is called Anonymous types.

  Declaration:

  ```
  Var   x=new {eno=1,ename="abc"};
  ```

  based on above declaration CLR will generates a class automatically at runtime with two auto-implemented properties called as eno,ename.
- this auto generated class is not visible to the programmer

Anonymous Types

```
static void Main()
    {
      var x = new { eno = 12, ename = "abc" };
  Console.WriteLine(x.eno + "  " + x.ename);
      Console.ReadKey();


   }
```

Extension Methods

- Extension methods Extending .NET types with new methods.
- An extension method should be a static method
- Extension methods should be declared in static class.
- Extension methods are associated with any class.
- The diff b/w normal instance method and Extension method is that extension methods are not defined with in the class itself that they are defined some other classes.

Extension Methods

- Extension method must contain at least one argument and first argument must be specified with this keyword.

syntax:

```
      public static void print(this data type   x)
            {


            }
```

Extension Methods

Diff b/w extension method and static method

Extension Method	Static Method
Extension method have the this keyword before the first argument	static method do not have the this keyword
Extensions method will be declared only in static class	static methods will declare in static class and normal class.

Extension Methods

- Extension methods calls only variables.

Note:

- extension methods cannot be applied to properties,events etc.
- Extension methods can not be used to override existing methods.

Lambda Expressions

- A lambda expression is an anonymous function that can be use to create delegates or expression tree types.
- It's just a method without a declaration
- Using lambda expressions, we can write local functions that can be passed as arguments of methods.
- Lambda expressions are particularly helpful for writing LINQ query expressions.
- To create a lambda expression use the lambda operator =>

Lambda Expressions

- In General Lambda expressions specify input parameters (if any) on the left side of the lambda operator => and expression or statements block on the other side.
- Ex: x => x * x
- Above lambda expression specifies a parameter that's named x and returns the value of x squared. we can assign this expression to a delegate type.

Lambda expression example

```
using System;
class X
{
    delegate int cube(int i);
    static void Main(string[] args)
    {
        cube myDelegate = x => x*x*x;
        int j = myDelegate(5);
        Console.Write(j);
    }
}

//Prints 125
```

Lambda Expressions

- (input parameters) => expression[expression lambda]
  - (x, y) => x =y return x value
  - (x, y) => x == y return bool value
  - (int x, string s) => s.Length > x return bool value
  - () => SomeMethod()
  - (input parameters) => {statement;}

Lambda Expressions

- WHY do we need Lambda Expression?
- The main reason for Lambda expression is convenient, it just a syntax sugar, it allows you to write method in the same place you are going to use it. In particular cases when you have method that is being used only once and it also very short method. It can save you the need to declare a seperate method. The benifits of it are:
  It can reduce typing by not specify the name of the function, it's return type and its access modifiere
- It more convenient because you don't need to look where is that method, how does it doing her job
- It's all under the assumpetion that the lambda is short and not complex, otherwise it will make your called funciton more complex