C#
Programming

Mentor as a Service

Introduction to .Net Framework

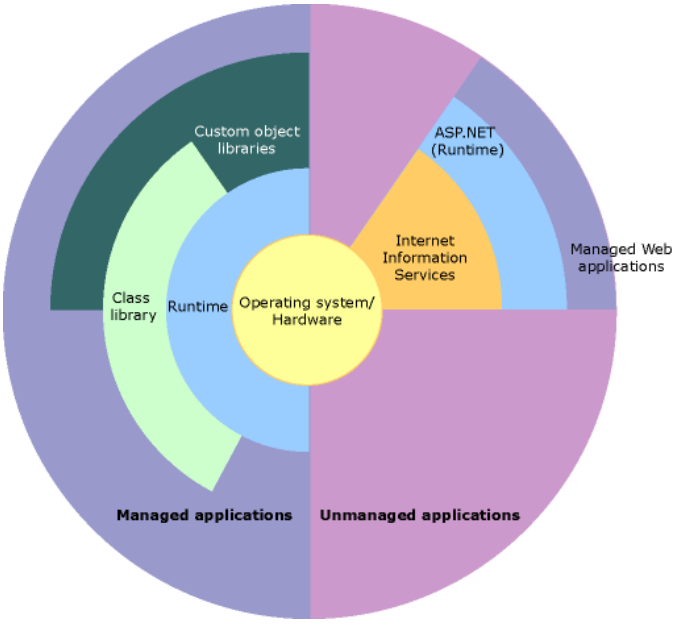
The .NET Framework is a development platform for building apps for Windows, Windows Phone, Windows Server, and Microsoft Azure.

Objectives of .NET Framework 4.5

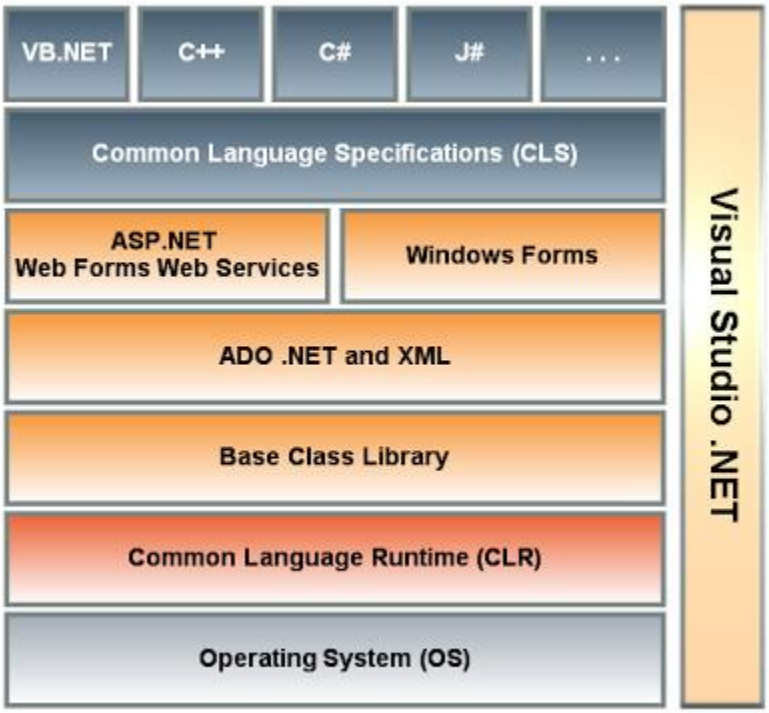
The .NET Framework is designed to fulfill the following objectives:

- To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- To provide a code-execution environment that minimizes software deployment and versioning conflicts.
- To provide a code-execution environment that promotes safe execution of code, including code created by an unknown or semi-trusted third party.
- To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- To make the developer experience consistent across widely varying types of applications, such as Windows-based applications and Web-based applications.
- To build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code.

.NET Framework Context



Dot Net Framework Components



Development in .NET Framework 4.5

Visual Studio is an Integrated Development Environment.

Application Life Cycle Management (ALM) can be managed using Visual Studio .NET

Design **Develop** **Test** **Debug** **Deploy**

· **Visual Studio .NET 2015 as IDE Tool**

The Design goals are:

Maximize Developer Productivity
Simplify Server based Development
Deliver Powerful Design, Development Tools

RAD (Rapid Application Development Tool) for the next generation internet solutions
Enhanced RAD Support for creating Custom components for Business solutions.

Tools for creating Rich Client Applications
Tools for creating ASP.NET web sites
Tools for Creating S-a-a-S modules
Tools for connecting remote servers
Tools for Cloud Connectivity
Tools for Data Access
Tools for Creating, Developing, Testing and Deploying Solutions.
Help and Documentation
Multiple .NET Language Support

Visual Studio Solutions and Projects

Visual Studio Solution consist of multiple Projects

Visual Studio Project consist of Source code, Resources.

· **C# as .NET Programming Language**

C ++ Heritage
Namespaces, Pointers (in unsafe code),
Unsigned types, etc.
Increased Productivity
Short Learning curve
C# is a type safe Object Oriented Programming Language
C# is case sensitive
Interoperability
C# can talk to COM, DLLs and any of the .NET Framework languages

Structure of first C# program

```
using System;
// A "Hello World!" program in C#
public class HelloWorld
{ public static void Main ()
  {
       Console.WriteLine ("Hello, World");

  }
}
```

Passing Command Line Arguments

```
using System;

/* Invoke exe using console */

public class HelloWorld
{
     public static void Main (string [] args)
     {
          Console.WriteLine ("parameter count = {0}", args.Length);
          Console.WriteLine ("Hello {0}", args [0]);
          Console.ReadLine ();

     }
}
```

Execution of .NET Application

C# code is compiled by CSC.exe (C# compiler) into assembly as Managed code.

Managed code is targeted to Common Language Runtime of .NET Framework

Common Language Runtime converts MSIL code into Platform dependent executable code (native code) for targeted operating System**.**

Application is executed by Operating System as Process.

C# Types

A C# Program is a collection of types
    Structure, Classes, Enumerations, Interfaces, Delegates, Events
C# provides a set of predefined types
    e.g. int, byte, char, string, object, etc.
Custom types can be created.
All data and code is defined within a type.
    No global variables, no global function.

Types can be instantiated and used by
    Calling methods, setters and getters, etc.
Types can be converted from one type to another.
Types are organized into namespaces, files, and assemblies.
Types are arranged in hierarchy.

In .NET Types are of two categories

Value Type

Directly contain data on Stack.

Primitives:	`int num; float speed;`
Enumerations:	`enum State {off, on}`
Structures:	`struct Point {int x, y ;}`

Reference Types

Contain reference to actual instance on managed Heap.

Root	`Object`
String	`string`
Classes	`class Line: Shape{ }`
Interfaces	`interface IDrawble {….}`
Arrays	`string [] names = new string[10];`
Delegates	`delegate void operation ();`

Type Conversion

Implicit Conversion

No information loss

Occur automatically

Explicit Conversion

Require a cast

May not succeed

Information (precision) might be lost

```
int x=543454;
long y=x;                  //implicit conversion
short z=(short)x;          //explicit conversion
double d=1.3454545345;
float f= (float) d;        //explicit conversion
long l= (long) d           // explicit conversion
```

Constants and read only variables

```
// This example illustrates the use of constant data and readonly fields.

using System;
using System.Text;

namespace ConstData
{

  class MyMathClass
  {
    public static readonly double PI;
    static MyMathClass()
    { PI = 3.14; }
  }

  class Program
  {
    static void Main(string [] args)
    {
      Console.WriteLine ("***** Fun with Const *****\n");
      Console.WriteLine ("The value of PI is: {0}", MyMathClass.PI);

      // Error! Can't change a constant!
      // MyMathClass.PI = 3.1444;

      LocalConstStringVariable ();
    }

    static void LocalConstStringVariable()
    {
          // A local constant data point.
          const string fixedStr = "Fixed string Data";
          Console.WriteLine(fixedStr);

          // Error!
          //fixedStr = "This will not work!";
    }
  }
}
```

Enumerations

Enumerations are user defined data Type which consist of a set of named integer constants.

```
enum Weekdays { Mon, Tue, Wed, Thu, Fri, Sat}
```

Each member starts from zero by default and is incremented by 1 for each next member.

Using Enumeration Types

```
Weekdays day=Weekdays.Mon;
Console.WriteLine("{0}", day);    //Displays Mon
```

Structures

Structure is a value type that is typically used to encapsulate small groups of related variables.

```
public struct Point
{    public int x;
     public int y;
}
```

Arrays

Declare

```
int [] marks;
```

Allocate

```
int [] marks= new int [9];
```

Initialize

```
int [] marks=new int [] {1, 2, 3, 4, 5, 6, 7, 9};
int [] marks={1,2,3,4,5,6,7,8,9};
```

Access and assign

```
Marks2[i] = marks[i];
```

Enumerate

```
foreach (int i in marks) {Console.WriteLine (i); }
```

Params Keyword

It defines a method that can accept a variable number of arguments.

```
static void ViewNames (params string [] names)
{
     Console.WriteLine ("Names: {0}, {1}, {2}",
                    names [0], names [1], names [2]);
}
public static void Main (string [] args)
{
     ViewNames("Nitin", "Nilesh", "Shrinivas");
}
```

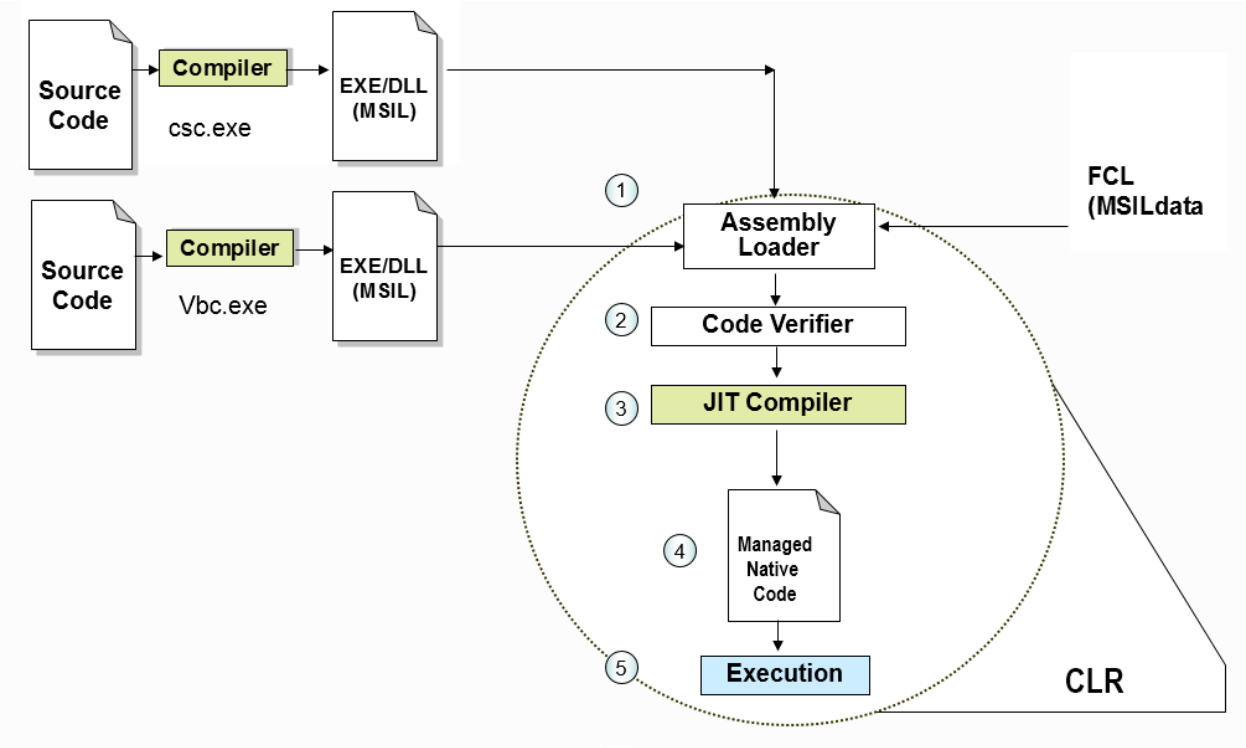
ref and out parameters

```
void static Swap (ref int n1, ref int n2)
{
     int temp =n1; n1=n2; n2=temp;
}

void static Calculate (float radius, out float area, out float circum)
{
     Area=3.14f * radius * radius;
     Circum=2*3.14f * radius;
}

public static void Main ()
{
     int x=10, y=20;
     Swap (ref x, ref y);
     float area, circum;
     Calculate (5, out area, out circum);
}
```

Execution Process in .NET Environment



.NET Assembly

A Logical unit of Deployment on .NET Platform.

Components of Assembly

- Manifest
- Metadata
- MSIL code
- Resources

Types of Assemblies

Private Assembly (bin folder)
Shared Assembly (GAC)
System.Data.dll
System.Drawing.dll
System.Web.dll
Etc.

Windows vs. .NET executables

Windows Exe consist of native code
.NET Exe consist of MSIL code

Inside .NET Framework

Common Language Runtime (CLR):

CLR is the heart of the .NET framework and it does 4 primary important things:

1. Garbage collection
2. CAS (Code Access Security)
3. CV (Code Verification)
4. IL to Native translation.

Common Type System (CTS): -

CTS ensure that data types defined in two different languages get compiled to a common data type. This is useful because there may be situations when we want code in one language to be called in other language. We can see practical demonstration of CTS by creating same application in C# and VB.Net and then compare the IL code of both applications. Here the data type of both IL code is same.

Common Language Specification (CLS):

CLS is a subset of CTS. CLS is a set of rules or guidelines. When any programming language adheres to these set of rules it can be consumed by any .Net language.

e.g. Lang must be object oriented, each object must be allocated on heap,

Exception handling supported.

Also each data type of the language should be converted into CLR understandable types by the Lang compiler.

All types understandable by CLR forms CTS (common type system) which includes:

```
System.Byte, System.Int16, System.UInt16,
System.Int32, System.UInt32, System.Int64,
System.UInt64, System.Char, System.Boolean, etc.
```

Assembly Loader:

When a .NET application runs, CLR starts to bind with the version of an assembly that the application was built with. It uses the following steps to resolve an assembly reference:

1) Examine the Configuration Files
2) Check for Previously Referenced Assemblies
3) Check the Global Assembly Cache
4) Locate the Assembly through Codebases or Probing

MSIL Code Verification and Just In Time compilation (JIT)

When .NET code is compiled, the output it produces is an intermediate language (MSIL) code that requires a runtime to execute the IL. So during assembly load at runtime, it first validates the metadata then examines the IL code against this metadata to see that the code is type safe. When MSIL code is executed, it is compiled Just-in-Time and converted into a platform-specific code that's called native code.
Thus any code that can be converted to native code is valid code. Code that can't be converted to native code due to unrecognized instructions is called Invalid code. During JIT compilation the code is verified to see if it is type-safe or not.

Garbage Collection

"Garbage" consists of objects created during a program's execution on the managed heap that are no longer accessible by the program. Their memory can be reclaimed and reused with no adverse effects.

The garbage collector is a mechanism which identifies garbage on the managed heap and makes its memory available for reuse. This eliminates the need for the programmer to manually delete objects which are no longer required for program execution. This reuse of memory helps reduce the amount of total memory that a program needs to run.

IL dis-assembler

The IL Disassembler is a companion tool to the IL Assembler (**Ilasm.exe**). Ildasm.exe takes a portable executable (PE) file that contains intermediate language (IL) code and creates a text file suitable as input to Ilasm.exe.

.NET Framework Folder

C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework

Object Orientation in C#

Object

A real world entity which has well defined structure and behavior.

Characteristics of an Object are:

- State
- Behavior
- Identity
- Responsibility

Pillars of Object Orientation

- Abstraction
- Encapsulation
- Inheritance
- Typing, Concurrency, Hierarchy, Persistence

Abstraction

Getting essential characteristic of a System depending on the perspective of on Observer.
Abstraction is a process of identifying the key aspects of System and ignoring the rest
Only domain expertise can do right abstraction.

Abstraction of Person Object

- Useful for social survey
- Useful for healthcare industry
- Useful for Employment Information

Encapsulation

Hiding complexity of a System.
Encapsulation is a process of compartmentalizing the element of an abstraction that constitute its structure and behavior.

- Servers to separate interface of an abstraction and its implementation.
- User is aware only about its interface: any changes to implementation does not affect the user.

Inheritance

Classification helps in handing complexity.
Factoring out common elements in a set of entities into a general entity and then making it more and more specific.
Hierarchy is ranking or ordering of abstraction.
Code and Data Reusability in System using is a relationship.

Typing

Typing is the enforcement of the entity such that objects of different types may not be interchanges, or at the most, they may be interchanged only in restricted ways.

Concurrency

Different objects responding simultaneously.

Persistence

Persistence of an object through which its existence transcends time and or space.

Namespace and Class

Namespace is a collection .NET Types such as structure, class, interfaces, etc.

```
namespace  EmployeeApp
{
   public class Employee
  {
    private  string empName;
    private  int empID;
    private  float currPay;
    private   int empAge;
    private  string empSSN;
    private static string companyName;
    public Employee ()
    { empID=18; currPay=15000; }
    public Employee (int id, float basicSal)
    { empID=id; currPay= basicSal; }
    public ~Employee()
    { //DeInitializtion }
    public void GiveBonus(float amount)
    {      currPay += amount;    }
    public void DisplayStats()
    {
     Console.WriteLine("Name: {0}", empName);
     Console.WriteLine("ID:   {0}", empID);
     Console.WriteLine("Age:  {0}", empAge);
     Console.WriteLine("SSN:  {0}", empSSN);
     Console.WriteLine("Pay:  {0}", currPay);
    }
 }
}
```

Partial class

A class can be spread across multiple source files using the keyword partial.
All source files for the class definition are compiled as one file with all class members.
Access modifiers used for defining a class should be consistent across all files.

Properties (smart fields)

Have two assessors:

Get retrieves data member values.

Set enables data members to be assigned

```
public int EmployeeID
{ get {return _id;}
  set {_id=value ;}
}
```

Indexers (smart array)

```
public class Books
{
 private string [] titles= new string [100];
 public string this [int index]
  {
     get{ if (index <0 || index >=100)
               return 0;
          else
               return titles [index];
        }
     set{
          if (! index <0 || index >=100)
               return 0;
          else
               titles [index] =value;
        }
   }
}
public static void Main ()
{ Books mybooks=new Books ();
  Mybooks [3] =”Mogali in Jungle”;
}
```

Singleton class

```
public class OfficeBoy
 {      private static OfficeBoy _ref = null;
        private  int _val;
        private  OfficeBoy()   {   _val = 10;  }
        public  int Val {   get  {  return _val;  }
                            set { _val = value; }
                        }
       public static OfficeBoy GetObject ()
        {    if (_ref == null)
               _ref = new OfficeBoy ();
             return _ref;
        }
 }
static void Main(string[] args)
 {    OfficeBoy sweeper, waiter;
      string s1; float f1;
      sweeper = OfficeBoy.GetObject(); waiter = OfficeBoy.GetObject();
      sweeper.Val = 60;
     Console.WriteLine("Sweeper Value : {0}", sweeper.Val);
     Console.WriteLine("Waiter Value  : {0}", waiter.Val);
     s1 = sweeper.Val.ToString();
     f1 = (float)sweeper.Val;
     sweeper.Val = int.Parse(s1);
     sweeper.Val = Convert.ToInt32(s1);
 }
```

Arrays

Multidimensional Arrays (Rectangular Array)

```
int [  , ]  mtrx = new int [2, 3];
     Can initialize declaratively
int [ , ] mtrx = new int [2, 3] { {10, 20, 30}, {40, 50, 60} }
```

Jagged Arrays

An Array of Arrays

```
int [ ]  [ ]  mtrxj = new  int [2] [];
```

Must be initialize procedurally.

Nullable Types

```
class  DatabaseReader
{
  public int? numericValue = null;
  public bool? boolValue = true;
  public int? GetIntFromDatabase()      { return numericValue; }
  public bool? GetBoolFromDatabase()    { return boolValue; }
}
public static void Main (string[] args)
 {
    DatabaseReader  dr = new DatabaseReader();
    int? i = dr.GetIntFromDatabase();
    if (i.HasValue)
      Console.WriteLine("Value of 'i' is: {0}", i.Value);
    else
      Console.WriteLine("Value of 'i' is undefined.");
    bool? b = dr.GetBoolFromDatabase();
    int? myData = dr.GetIntFromDatabase() ?? 100;
    Console.WriteLine("Value of myData: {0}", myData.Value);
 }
static void LocalNullableVariables ()
{   int? nullableInt = 10;
    double? nullableDouble = 3.14;
    bool? nullableBool = null;
    int?[] arrayOfNullableInts = new int?[10];
    // Define some local nullable types using Nullable<T>.
    Nullable<int> nullableInt = 10;
    Nullable<double> nullableDouble = 3.14;
    Nullable<bool> nullableBool = null;
    Nullable<int> [] arrayOfNullableInts = new int?[10];
  }}}
```

Overloading

Method Overloading

Overloading is the ability to define several methods with the same name, provided each method has a different signature

```
public class MathEngine
{
public static double FindSquare (double number) { // logic defined   }
public static double FindSquare (int number) { // another logic defined }
}
public static void Main ()
{
     double res= MathEngine.FindSquare(12.5);
     double num= MathEngine.FindSquare(12);
}
```

Operator Overloading

Giving additional meaning to existing operators.

```
 public static Complex Operator + (Complex c1, Complex c2)
 {
     Complex temp= new Complex();
     temp.real = c1.real+ c2.real;
     templ.imag = c1.image + c2.imag;
     return temp;
 }
public static void Main ()
{
     Complex o1= new Complex (2, 3);
     Complex o2= new Complex (5, 4);
     Complex o3= 1+ o2;
     Console.WriteLine (o3.real + " " + o3.imag);
}
```

Operator overloading restrictions

Following operators cannot be overloaded.

Conditional logical &&, ||	Array indexing operator [], Cast Operators ()	Array indexing operator [] Cast Operators ()
Assignment operators +=,-=,*=,/+ etc	=,.,? :,->, new, is, sizeof, typeof	The comparison operator, if overloaded, must be overloaded in pairs. If == is overloaded then != must also be overloaded

C # Reusability

Inheritance

Provides code reusability and extensibility.

Inheritance is a property of class hierarchy whereby each derived class inherits attributes and methods of its base class.

Every Manager is Employee.

Every Wage Employee is Employee.

```
class Employee
{   public double CalculateSalary ()
       {return basic_sal + hra+ da ;}
}

class Manager: Employee
{
   public double CalculateIncentives ()
    {
     //code to calculate incentives
     Return incentives;
   }
}

static void Main ()
{
     Manager mgr =new Manager ();
     double Inc=mgr. CalculateIncentives ();
     double sal=mgr. CalculateSalary ();
}
```

Constructors in Inheritance

```
class Employee
{ public  Employee ()
    {Console.WriteLine ("in Default constructor") ;}
  public Employee (int eid, ….)
     {Console.WriteLine ("in Parameterized constructor") ;}
}
class Manager: Employee
{
 public Manager (): base ()    {……..}
 public Manager (int id): base (id,….)    {…..}
}
```

Polymorphism

Ability of different objects to responds to the same message in different ways is called Polymorphism.

```
horse.Move();
car.Move();
aeroplane.Move();
```

Virtual and Override

Polymorphism is achieved using virtual methods and inheritance.
Virtual keyword is used to define a method in base class and override keyword is used in derived class.

```
class Employee
{ public virtual double CalculateSalary ()
     {return basic_sal+ hra + da ;}
}
class Manager: Employee
{
  public override double CalculateSalary ()
     {return (basic_sal+ hra + da + allowances);}
}
staic void Main ()
{ Employee mgr= new Manager ();
  Double salary= mgr. CalculateSalary ();
  Console.WriteLine (salary);
}
```

Shadowing

Hides the base class member in derived class by using keyword new.

```
class Employee
{public virtual double CalculateSalary ()
     {return basic_sal;}
}
class SalesEmployee:Employee
{ double sales, comm;
  public new double CalculateSalary ()
     {return basic_sal+ (sales * comm) ;}
}
static void Main ()
{ SalesEmployee sper= new SalesEmployee ();
  Double salary= sper.CalculateSalary ();
  Console.WriteLine (salary);
}
```

Sealed class

Sealed class cannot be inherited

```
sealed class SinglyList
{
  public virtual double Add ()
  {// code to add a record in the linked list}
}
public class StringSinglyList:SinglyList
{
  public override double Add ()
  {  // code to add a record in the String linked list}
 }
```

Concrete class vs. abstract classes

Concrete class

Class describes the functionality of the objects that it can be used to instantiate.

Abstract class

Provides all of the characteristics of a concrete class except that it does not permit object instantiation.

An abstract class can contain abstract and non-abstract methods.

Abstract methods do not have implementation.

```
abstract class Employee
 {  public virtual double CalculateSalary();
     { return basic +hra + da ;}
    public abstract double CalculateBonus();
  }
 class Manager: Employee
  {   public override double CalculateSalary();
       { return basic + hra + da + allowances ;}
      public override double CalaculateBonus ()
       { return basic_sal * 0.20 ;}
  }

static void Main ()
{    Manager mgr=new Manager ();
     double bonus=mgr. CalaculateBonus ();
     double Salary=mgr. CalculateSalary ();
}
```

Object class

Base class for all .NET classes

Object class methods

- **public bool Equals(object)**
- **protected void Finalize()**
- **public int GetHashCode()**
- **public System.Type GetType()**
- **protected object MemberwiseClone()**
- **public string ToString()**

Polymorphism using Object

The ability to perform an operation on an object without knowing the precise type of the object.

```
void Display (object o)
{
     Console.WriteLine (o.ToString ());
}
public static void Main ()
{  Display (34);
   Display ("Transflower");
   Display (4.453655);
   Display (new Employee ("Ravi", "Tambade");
}
```

Interface Inheritance

For loosely coupled highly cohesive mechanism in Application.

An interface defines a Contract

Text Editor uses Spellchecker as interfaces.

EnglishSpellChecker and FrenchSpellChecker are implementing contract defined by SpellChecker interface.

```
interface ISpellChecker
{ ArrayList CheckSpelling (string word) ;}
class EnglishSpellChecker:ISpellChecker
{
  ArrayList CheckSpelling (string word)
   {// return possible spelling suggestions}
}
class FrenchSpellChecker:ISpellChecker
{
  ArrayList CheckSpelling (string word)
   {// return possible spelling suggestions}
}
class TextEditor
{
 public static void Main()
 {    ISpellChecker checker= new EnglishSpellChecker ();
     ArrayList words=checker. CheckSpelling ("Flower");
     …
 }
}
```

Explicit Interface Inheritance

```
interface IOrderDetails    { void ShowDetails() ;}
interface ICustomerDetails { void ShowDetails() ;}
class Transaction: IOrderDetails, ICustomerDetils
{
     void IOrderDetails. ShowDetails()
     {  // implementation for interface IOrderDetails ;}
     void ICustomerDetails. ShowDetails()
     {  // implementation for interface IOrderDetails ;}
}
public static void Main()
{
     Transaction obj = new Transaction();
     IOrderDetails od = obj;
     od.ShowDetails();
     ICustomerDetails cd = obj;
     cd.ShowDetails();
}
```

Abstract class vs. Interface

	Abstract class	**Interface**
Methods	At least one abstract method	All methods are abstract
Best suited for	Objects closely related in hierarchy.	Contract based provider model
Multiple Inheritance	Not supported	Supported
Component Versioning	By updating the base class all derived classes are automatically updated.	Interfaces are immutable

Building cloned Objects

```
class StackClass: ICloneable
 {  int size; int [] sArr;
    public StackClass (int s) { size=s; sArr= new int [size]; }
    public object Clone()
    {  StackClass s = new StackClass(this.size);
     this.sArr.CopyTo(s.sArr, 0);
     return s;
    }
 }
public static void Main()
{ StackClass stack1 = new StackClass (4);
  Stack1 [0] = 89;
  …..
 StackClass stack2 = (StackClass) stack1.Clone ();
}
```

Reflection

Reflection is the ability to examine the metadata in the assembly manifest at runtime.
Reflection is useful in following situations:

- Need to access attributes in your programs metadata.
- To examine and instantiate types in an assembly.
- To build new types at runtime using classes in **System.Reflection.Emit** namespace.
- To perform late binding, accessing methods on types created at runtime.

System. Type class
Type class provides access to metadata of any .NET Type.

System. Reflection namespace
Contains classes and interfaces that provide a managed view of loaded types, methods and fields
These types provide ability to dynamically create and invoke types.

Type	Description
Module	Performs reflection on a module
Assembly	Load assembly at runtime and get its type
MemeberInfo	Obtains information about the attributes of a member and provides access to member metadata

Assembly class

```
MethodInfo method;
string methodName;
object result = new object ();
object [] args = new object [] {1, 2};
Assembly asm = Assembly.LoadFile (@"c:/transflowerLib.dll");
Type [] types= asm.GetTypes();
foreach (Type t in types)
{     method = t.GetMethod(methodName);
     string typeName= t.FullName;
     object obj= asm.CreateInstance(typeName);
      result = t.InvokeMember (method.Name, BindingFlags.Public |
      BindingFlags.InvokeMethod | BindingFlags.Instance, null, obj, args);
     break;
}string res = result.ToString();
Console.WriteLine ("Result is: {0}", res);
```

Garbage Collection

COM

Programmatically implement reference counting and handle circular references

C++

Programmatically uses the new operator and delete operator

Visual Basic

Automation memory management

Manual vs. Automatic Memory Management

Common problems with manual memory management

- Failure to release memory
- Invalid references to freed memory

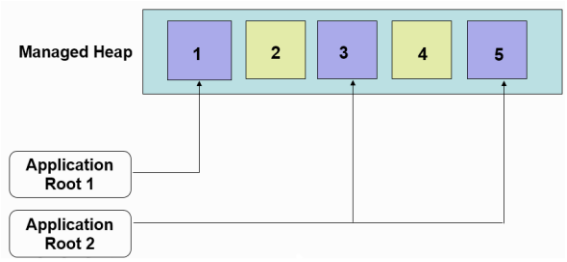
Automatic memory management provided with .NET Runtime

- Eases programming task
- Eliminates a potential source of bugs.

Garbage Collector

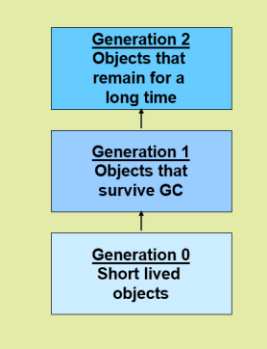
Manages the allocation and release of memory for your application.

Application Roots.



1. Identifies live object references or application roots and builds their graph
2. Objects not in the graph are not accessible by the application and hence considered garbage.
3. Finds the memory that can be reclaimed.
4. Move all the live object to the bottom of the heap, leaving free space at the top.
5. Looks for contiguous block objects & then shifts the non-garbage objects down in memory.
6. Updates pointers to point new locations.

Generational Garbage collection



Resource Management Types

Implicit Resource Management

With Finalize () method.
Will be required when an object encapsulates unmanaged resources like: file, window or network connection.

Explicit Resource Management

By implementing IDisposable Interface and writing Dispose method.

Implicit Recourse Management with Finalization

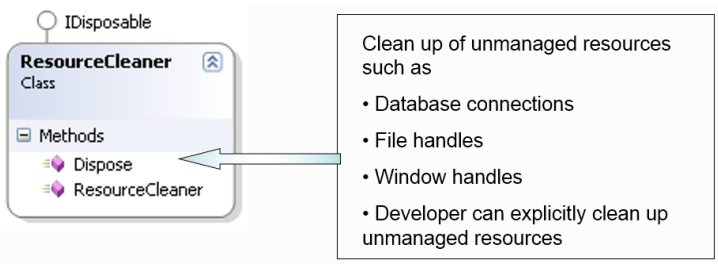
Writing Destructors in C#:

```
Class Car{
        ~Car () //destructor  {  // cleanup statements}
}
```

Explicit Resource Management

Implement IDisposable Interface.

Defines Dispose () method to release allocated unmanaged resources.



.NET Collection Framework

A Collection is a set of similarly typed objects that are grouped together.

System.Array class

The base class for all array Types.

```
int[] intArray= new int[5] { 22,11,33,44,55 };
foreach (int i in intArray )
{ Console.WriteLine( "\t {0}", i); }
Array.Sort(intArray);
Array.Reverse(intArray);
```

Collection Interfaces

Allow collections to support a different behavior

Interface	Description
IEnumertor	Supports a simple iteration over collection
IEnumerable	Supports foreach semantics
ICollection	Defines size, enumerators and synchronization methods for all collections.
IList	Represents a collection of objects that could be accessed by index
IComaprable	Defines a generalized comparison method to create a type-specific comparison
IComparer	Exposes a method that compares two objects.
IDictionary	Represents a collection of Key and value pairs

Implementing IEnumerable Interface

```
public class Team:IEnumerable
{
 private player [] players;
 public Team ()
 {
      Players= new Player [3];
      Players[0] = new Player("Sachin", 40000);
      Players[1] = new Player("Rahul", 35000);
      Players[2] = new Player("Mahindra", 34000);
 }

 public IEnumertor GetEnumerator ()
 {
     Return players.GetEnumerator();
 }
}


public static void Main()
{
Team India = new Team();
foreach(Player c in India)
 {
  Console.WriteLine (c.Name, c.Runs);
 }
}
```

Implementing ICollection Interface

To determine number of elements in a container.
Ability to copy elements into `System.Array` type

```
public class Team:ICollection
{
 private Players [] players;
 public Team() {……..}
 public int Count {get {return players.Count ;}
}

 // other implementation of Team
}
```

```
public static void Main()
{
Team India = new Team ();
foreach (Player c in India)
 {
  Console.WriteLine (c.Name, c.Runs);
 }
}
```

Implementing IComparable Interface

```
public class Player:IComparable
{

  int IComparable.CompareTo(object obj)
  {
    Player temp= (Player) obj;
    if (this. Runs > temp.Runs)
       return 1;
    if (this. Runs < temp.Runs)
     return -1;
    else
      return 0;
  }
}
public static void Main()
{
 Team India = new Team();

 // add five players with Runs

 Arary.Sort(India);

// display sorted Array

}
```

Using Iterator Method

```
public  class Team
{
 private  player [] players;
 public  Team ()
 {
      Players= new Player [3];
      Players[0] = new Player("Sachin", 40000);
      Players[1] = new Player("Rahul", 35000);
      Players[2] = new Player("Mahindra", 34000);
 }

 public IEnumertor GetEnumerator()
 {
     foreach (Player p in players)
     {yield return p ;}
 }
}

public static void Main()
{  Team India = new Team();
   foreach(Player c in India)
   {
     Console.WriteLine(c.Name, c.Runs);
   }
}
```

Collection Classes

ArrayList class

Represents list which is similar to a single dimensional array that can be resized dynamically.

```
ArrayList countries = new ArrayList ();
countries.Add("India");
countries.Add("USA");
countries.Add("UK");
countries.Add("China");
countries.Add("Nepal");

Console.WriteLine("Count: {0}", countries. Count);

foreach(object obj in countries)
 {
  Console.WriteLine("{0}", obj);

 }
```

Stack class

Represents a simple Last- In- First- Out (LIFO) collection of objects.

```
Stack  numStack = new Stack();
numStack.Push(23);
numStack.Push(34);
numStack.Push(76);
numStack.Push(9);
Console.WriteLine("Element removed", numStack.Pop());
```

Queue class

Represents a first- in, first- out (FIFO) collection of objects.

```
Queue myQueue = new Queue ();
myQueue.Enqueue("Nilesh");
myQueue.Enqueue("Nitin");
myQueue.Enqueue("Rahul");
myQueue.Enqueue("Ravi");

Console.WriteLine("\t Capacity: {0}", myQueue.Capacity);
Console.WriteLine("(Dequeue) \t {0}", myQueue.Dequeue());
```

HashTable class

Represents a collection of Key/ value pairs that are organized based on the hash code of the key.

Each element is a key/ value pair stored in a DictionaryEntry object.

```
Hashtable h = new Hashtable();
h.Add("mo", "Monday");
h.Add("tu", "Tuesday");
h.Add("we", "Wednesday");
IDictionaryEnumerator e= h. GetEnumerator( );
     while(e.MoveNext( ))
     {
       Console.WriteLine(e.Key + "\t" + e.Value);
     }
```

Generics

Are classes, structures, interfaces and methods that have placeholders (type parameters) for one or more of the types they store or use.

```
class HashTable<K,V>
{
 public HashTable();
 public object Get(K);
 public object Add(K, V);
}

HashTable <string,int> addressBook;
..
addressBook.Add("Amit Bhagat", 44235);
..
int extension = addressBook.Get("Shiv Khera");
```

List<T> class

Represents a strongly typed list of objects that can be accessed by index.

Generic equivalent of ArrayList class.

```
List<string> months = new List <string> ();
months.Add("January");
months.Add("February");
months.Add("April");
months.Add("May");
months.Add("June");
foreach(string mon in months)

Console.writeLine(mon);

Months.Insert(2,"March");
```

List of user defined objects

```
class Employee
{
     int eid;//appropriate constructor and properties for Employee Entity
     string ename;
}
class EmpComparer:IComparer<Employee>
{
  public int Compare(Employee e1, Employee e2)
  { int ret = e1.Name.Length.CompareTo(e2.Name.Length); return ret;
  }
}

public static void Main ()
{
List<Employee>list1 = new List<Employee>();
List1.Add(new Employee(1, "Raghu");
List1.Add(new Employee(2, "Seeta");
List1.Add(new Employee(4, "Leela");
EmpComparer ec = new EmpComparer();
List1.Sort(ec);
foreach(Employee e in list1)
Console.WriteLine(e.Id + "---------"+ e.Name);
}
```

Stack<T> class

```
Stack<int>numStack = new Stack<int>();

numStack.Push(23);
numStack.Push(34);
numStack.Push(65);

Console.WriteLine("Element removed: ", numStack. Pop());
```

Queue<T> class

```
Queue<string> q = new Queue<string>();
q.Enqueue("Message1");
q.Enqueue("Message2");
q.Enqueue("Message3");

Console.WriteLine("First message: {0}", q.Dequeue());
Console.WriteLine("The element at the head is {0}", q.Peek());
IEnumerator<string> e= q.GetEnumerator();
while(e.MoveNext())
Console.WriteLine(e.Current);
```

LinkedList<T> class

Represents a doubly linked List

Each node is on the type LinkedListNode

```
LinkedList<string> l1= new LinkedList<string>();
L1.AddFirst(new LinkedListNode<string>("Apple"));
L1.AddFirst(new LinkedListNode<string>("Papaya"));
L1.AddFirst(new LinkedListNode<string>("Orange"));
L1.AddFirst(new LinkedListNode<string>("Banana"));

LinkedListNode<string> node=l1.First;
Console.WriteLine(node. Value);
Console.WriteLine(node.Next.Value);
```

Dictionary<K, V> class

Represents a collection of keys and values.

Keys cannot be duplicate.

```
Dictionary<int, string> phones= new Dictionary<int, string>();
phones.Add(1, "James");
phones.Add(35, "Rita");
phones.Add(16, "Meenal");
phones.Add(41, "jim");

phones[16] = "Aishwarya";

Console.WriteLine("Name {0}", phones [12]);
if (!phone.ContainsKey(4))
phones.Add(4,"Tim");
Console.WriteLine("Name is {0}", phones [4]);
```

Custom Generic Types

Generic function

```
static void Main(string[] args)
   { // Swap 2 ints.
     int a = 10, b = 90;
     Console.WriteLine("Before swap: {0}, {1}", a, b);
     Swap<int>(ref a, ref b);
     Console.WriteLine("After swap: {0}, {1}", a, b);
     Console.WriteLine();

     // Swap 2 strings.
     string s1 = "Hello", s2 = "There";
     Console.WriteLine("Before swap: {0} {1}!", s1, s2);
     Swap<string>(ref s1, ref s2);
     Console.WriteLine("After swap: {0} {1}!", s1, s2);
     Console.WriteLine();

     // Compiler will infer System.Boolean.
     bool b1=true, b2=false;
     Console.WriteLine("Before swap: {0}, {1}", b1, b2);
     Swap (ref b1, ref b2);
     Console.WriteLine("After swap: {0}, {1}", b1, b2);
     Console.WriteLine();

     // Must supply type parameter if the method does not take params.
     DisplayBaseClass<int>();
     DisplayBaseClass<string>();
   }

 static void Swap<T>(ref T a, ref T b)
   {
     Console.WriteLine("You sent the Swap() method a {0}", typeof(T));
     T temp; temp = a; a = b; b = temp;
   }

 static void DisplayBaseClass<T>()
  {
    Console.WriteLine("Base class of {0} is: {1}.",
                      typeof(T), typeof(T).BaseType);
  }
```

Custom Structure

```
// A generic Point structure.
public  struct Point<T>
{// Generic state date.
  private T xPos;
  private T yPos;

  // Generic constructor.
  public Point(T xVal, T yVal)
    { xPos = xVal;  yPos = yVal;  }

 // Generic properties.
  public T X
    {  get { return xPos; }  set { xPos = value; }    }
  public T Y
   { get { return yPos; }  set { yPos = value; }    }
  public override string ToString()
  {   return string.Format("[{0}, {1}]", xPos, yPos);   }
  // Reset fields to the default value of the type parameter.
  public void ResetPoint()
  {  xPos = default(T); yPos = default(T);   }
}
```

```
static void Main(string[] args)
 {
   // Point using ints.
   Point<int> p = new Point<int>(10, 10);
   Console.WriteLine("p.ToString()={0}", p.ToString());
   p.ResetPoint();
   Console.WriteLine("p.ToString()={0}", p.ToString());

   // Point using double.
   Point<double> p2 = new Point<double>(5.4, 3.3);
   Console.WriteLine("p2.ToString()={0}", p2.ToString());
   p2.ResetPoint();
   Console.WriteLine("p2.ToString()={0}", p2.ToString());
 }
```

Custom Generic collection class

```
public class Car
  { public string PetName;  public int Speed;
    public Car(string name, int  currentSpeed)
    {    PetName = name;   Speed = currentSpeed;   }
    public Car() { }
  }

public class SportsCar : Car
  {  public SportsCar(string p, int s): base(p, s) { }
    // Assume additional SportsCar methods.
  }
public class MiniVan : Car

  {  public MiniVan(string p, int s)  : base(p, s) { }
    // Assume additional MiniVan methods.
  }

// Custom Generic Collection
  public class CarCollection<T> : IEnumerable<T> where T : Car
  { private List<T>  arCars = new List<T>();
    public T GetCar(int pos) { return arCars[pos]; }
    public void AddCar(T c)  { arCars.Add(c); }
    public void ClearCars()  { arCars.Clear(); }
    public int Count { get { return arCars.Count; }
```

```
// IEnumerable<T> extends IEnumerable,
//therefore we need to implement both versions of GetEnumerator().
    IEnumerator<T> IEnumerable<T>.GetEnumerator()
    {return arCars.GetEnumerator(); }
    IEnumerator IEnumerable.GetEnumerator()
    { return arCars.GetEnumerator(); }
 // This function will only work because of our applied constraint.
   public void PrintPetName(int pos)
   {    Console.WriteLine(arCars[pos].PetName);   }

}
 static void Main(string[] args)
    { // Make a collection of Cars.
      CarCollection<Car>  myCars = new CarCollection<Car>();
      myCars.AddCar(new Car("Alto", 20));
      myCars.AddCar(new Car("i20", 90));
      foreach (Car c in myCars)
      {  Console.WriteLine("PetName: {0}, Speed: {1}",
                                          c.PetName, c.Speed);
      }
     // CarCollection<Car> can hold any type deriving from Car.
      CarCollection<Car>   myAutos = new CarCollection<Car>();
      myAutos.AddCar(new MiniVan("Family Truckster", 55));
      myAutos.AddCar(new SportsCar("Crusher", 40));
      foreach (Car c in myAutos)
       {   Console.WriteLine("Type: {0}, PetName: {1}, Speed: {2}",
                             c.GetType().Name, c.PetName, c.Speed);
       }
    }
```

Exceptions Handling

Abnormalities that occur during the execution of a program (runtime error).

.NET framework terminates the program execution for runtime error.

e.g. divide by Zero, Stack overflow, File reading error, loss of network connectivity

Mechanism to detect and handle runtime error.

```
int a, b=0;

Console.WriteLine("My program starts");

try
 {
     a= 10/b;
 }
catch(Exception e)
{
      Console.WriteLine(e.Message);
}

Console.WriteLine("Remaining program");
```

.NET Exception classes

SystemException	FormatException
ArgymentException	IndexOutOfException
ArgumentNullException	InvalidCastExpression
ArraytypeMismatchException	InvalidOperationException
CoreException	NullReferenceException
DivideByZeroException	OutOfMemoryException
StackOverflowException	

User Defined Exception classes

Application specific class can be created using ApplicationException class.

```
class StackFullException:ApplicationException
{
  public string message;
  public StackFullException(string msg)
  {
     Message = msg;
  }
}

public static void Main(string [] args)
{
  StackClass stack1= new StackClass();

  try
  {  stack1.Push(54);
     stack1.Push(24);
     stack1.Push(53);
     stack1.Push(89);
  }

 catch(StackFullException s)
 {
     Console.WriteLine(s.Message);
 }
}
```

Attributes

Declarative tags that convey information to runtime.

Stored with the metadata of the Element

.NET Framework provides predefined Attributes

The Runtime contains code to examine values of attributes and to act on them

Types of Attributes

Standard Attributes

Custom Attributes

Standard Attributes

.NET framework provides many pre-defined attributes.

General Attributes
COM Interoperability Attributes
Transaction Handling Attributes
Visual designer component- building attributes

```
[Serializable]
public class Employee
{
    [NonSerialized]
   public string name;

}
```

Custom Attributes

User defined class which can be applied as an attribute on any .NET compatibility Types like:

Class
Constructor
Delegate
Enum
Event
Field
Interface
Method
Parameter
Property
Return Value
Structure

Attribute Usage

AttributeUsuageAttribute step 1

It defines some of the key characteristics of custom attribute class with regards to its application , inheritance, allowing multiple instance creation, etc.

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

AttributeUsuageAttribute step 2

Designing Attribute class

Attribute classes must be declared as public classes.
All Attribute classes must inherit directly or indirectly from **System.Attribute .**

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

public class MyAttribute:System.Attribute

{

    ….

}
```

AttributeUsuageAttribute step 3

Defining Members

Attributes are initialized with constructors in the same way as traditional classes.

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

Attribute properties should be declared as public entities with description of the data type that will be returned.

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

Applying Custom Attribute

Custom attribute is applied in following way.

Retrieving Custom Attributes

```
 [Developer("Ravi Tambade", "1"]
public class Employee
{

}
public static void Main()
     {
          Employee tflemp = new Employee();
          Type t = tflemp.GetType();
          foreach(Attribute a in t.GetCustomAttributes(true))
          {
           Developer r= (Developer) a;
           //Access r.Name and r.Level
         }
     }
```

Delegates

A delegate is a reference to a method.

All delegates inherit from the `System.delegate` type

It is foundation for Event Handling.

Delegate Types

Unicast (Single cast)
Multicast Delegate

Unicast (Single cast) Delegate

Steps in using delegates

i. Define delegate
ii. Create instance of delegate
iii. Invoke delegate

```
delegate string strDelegate(string str);

strDelegate strDel =new strDelegate(strObject.ReverseStr);

string str=strDel("Hello Transflower");

// or use this Syntax

string str =strDel.Invoke("Hello Transflower");
```

Multicast Delegate

A Multicast delegate derives from `System.MulticastDelegate` class.
It provides synchronous way of invoking the methods in the invocation list.
Generally multicast delegate has void as their return type.

```
delegate void strDelegate(string str);

strDelegate delegateObj;

strDelegate Upperobj = new strDelegate(obj.UppercaseStr);

strDelegate Lowerobj = new strDelegate(obj.LowercaseStr);

delegateObj=Upperobj;

delegateObj+=Lowerobj;

delegateObj("Welcome to Transflower");
```

Delegate chaining

Instances of delegate can be combined together to form a chain

Methods used to manage the delegate chain are

- Combine method
- Remove method

```
//calculator is a class with two methods Add and Subtract
Calculator obj1 = new Calculator();
CalDelegate [] delegates = new CalDelegate[];
          { new CalDelegate(obj1.Add),
           new CalDelegate(Calculator. Subtract)};
CalDelegate chain = (CalDelegate)delegate.Combine(delegates);
Chain = (CalDelegate)delegate.Remove(chain, delegates [0]);
```

Asynchronous Delegate

It is used to invoke methods that might take long time to complete.

Asynchronous mechanism more based on events rather than on delegates.

```
delegate void strDelegate(string str);
public class Handler
{
public static string UpperCase(string s) {return s. ToUpper() ;}
}
strDelegate caller = new strDelegate(handler. UpperCase);
IAsyncResult result = caller.BeginInvoke("transflower", null, null);
// . . .
String returnValue = caller.EndInvoke(result);
```

Anonymous Method

It is called as inline Delegate.

It is a block of code that is used as the parameter for the delegate.

```
delegate string strDelegate(string str);
public static void Main()
{
strDelegate upperStr = delegate(string s) {return s.ToUpper() ;};
}
```

Events

- An Event is an automatic notification that some action has occurred.
- An Event is built upon a Delegate

```
public delegate void AccountOperation();
public class Account
    {   private int balance;
        public event AccountOperation UnderBalance;
        public  event AccountOperation OverBalance;
        public Account() {balance = 5000 ;}
        public Account(int amount) {balance = amount ;}
        public void Deposit(int amount)
        {  balance = balance + amount;
            if (balance > 100000) {  OverBalance();   }
        }
        public void Withdraw(int amount)
        {   balance=balance-amount;
            if(balance < 5000) { UnderBalance () ;}
        }}}
```

Event Registrations using Event Handlers

```
class Program
  {  static void Main(string [] args)
        {   Account axisBanktflAccount = new Account(15000);
            //register Event Handlers
            axisBanktflAccount.UnderBalance+=PayPenalty;
            axisBanktflAccount.UnderBalance+=BlockBankAccount;
            axisBanktflAccount.OverBalance+=PayProfessionalTax;
            axisBanktflAccount.OverBalance+= PayIncomeTax;
            //Perform Banking Operations
            axisBanktflAccount.Withdraw(15000);
            Console.ReadLine();
        }
     //Event handlers
      static void PayPenalty()
        {Console.WriteLine("Pay Penalty of 500 within 15 days");    }
      static void BlockBankAccount()
        {Console.WriteLine("Your Bank Account has been blocked") ;}
      static void PayProfessionalTax()
        {Console.WriteLine("You are requested to Pay Professional Tax") ;}
      static void PayIncomeTax()
      {Console.WriteLine("You are requested to Pay Income Tax as TDS") ;}
}
```

LINQ Language Integrated Query

The C# 3.0 language enhancements build on C# 2.0 to increase developer productivity: they make written code more concise and make working with data as easy as working with objects. These features provide the foundation for the LINQ project, a general purpose declarative query facility that can be applied to in-memory collections and data stored in external sources such as XML files and relational databases.

The C# 3.0 language enhancements consist of:

Auto-implemented properties	automate the process of creating properties with trivial implementations
Implicitly typed local variables	permit the type of local variables to be inferred from the expressions used to initialize them
Implicitly typed arrays	a form of array creation and initialization that infers the element type of the array from an array initializer
Extension methods	which make it possible to extend existing types and constructed types with additional methods
Lambda expressions	an evolution of anonymous methods that concisely improves type inference and conversions to both delegate types and expression trees
Expression trees	permit lambda expressions to be represented as data (expression trees) instead of as code (delegates)
Object and collection initializers	you can use to conveniently specify values for one or more fields or properties for a newly created object, combining creation and initialization into a single step
Query expressions	provide a language-integrated syntax for queries that is similar to relational and hierarchical query languages such as SQL and XQuery
Anonymous types	are tuple types automatically inferred and created from object initializers

Use of Automatically Implemented Properties

Easy Initialization with Object and Collection Initializers

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

namespace NewLanguageFeatures
{
    public class Customer
    {
        public int CustomerID { get; private set; }
        public string Name { get; set; }
        public string City { get; set; }

        public Customer(int ID)
        {
            CustomerID = ID;
        }

        public override string ToString()
        {
            return Name + "\t" + City + "\t" + CustomerID;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Customer c = new Customer(1);
            c.Name = "Maria Anders";
            c.City = "Berlin";

            Console.WriteLine(c);
        }

    }
}
```

Implicitly Typed Local Variables and Implicitly Typed Arrays

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

namespace NewLanguageFeatures
{
    public class Customer
    {
        public int CustomerID { get; private set; }
        public string Name { get; set; }
        public string City { get; set; }

        public Customer(int ID)     { CustomerID = ID; }

        public override string ToString()
        {
            return Name + "\t" + City + "\t" + CustomerID;
        }
    }

    class Program
    {

        static void Main(string[] args)
        {
            List<Customer> customers = CreateCustomers();

            Console.WriteLine("Customers:\n");
            foreach (Customer c in customers)
            Console.WriteLine(c);
        }

        static List<Customer> CreateCustomers()
        {
           return new List<Customer>
           {
        new Customer(1) { Name = "Maria Anders",       City = "Berlin"    },
        new Customer(2) { Name = "Laurence Lebihan",  City = "Marseille" },
        new Customer(3) { Name = "Elizabeth Brown",   City = "London"    },
        new Customer(4) { Name = "Ann Devon",         City = "London"    },
        new Customer(5) { Name = "Paolo Accorti",     City = "Torino"    },
        new Customer(6) { Name = "Fran Wilson",       City = "Portland"  },
        new Customer(7) { Name = "Simon Crowther",    City = "London"    },
        new Customer(8) { Name = "Liz Nixon",         City = "Portland"  }
           };
       }
    }
}
```

Extending Types with Extension Methods

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

namespace NewLanguageFeatures
{
    public class Customer
    {
        public int CustomerID { get; private set; }
        public string Name { get; set; }
        public string City { get; set; }

        public Customer(int ID)
        {
            CustomerID = ID;
        }

        public override string ToString()
        {
            return Name + "\t" + City + "\t" + CustomerID;
        }
    }

   class Program
    {
        static void Main(string[] args)
        {
            List<Customer> customers = CreateCustomers();

            Console.WriteLine("Customers:\n");
            foreach (Customer c in customers)
                Console.WriteLine(c);
        }

        static List<Customer> CreateCustomers()
        {
           return new List<Customer>
           {
            new Customer(1) { Name = "Maria Anders",        City = "Berlin"    },
            new Customer(2) { Name = "Laurence Lebihan",    City = "Marseille" },
            new Customer(3) { Name = "Elizabeth Brown",     City = "London"    },
            new Customer(4) { Name = "Ann Devon",           City = "London"    },
            new Customer(5) { Name = "Paolo Accorti",       City = "Torino"    },
            new Customer(6) { Name = "Fran Wilson",         City = "Portland"  },
            new Customer(7) { Name = "Simon Crowther",      City = "London"    },
            new Customer(8) { Name = "Liz Nixon",           City = "Portland"  }
           };
       }
    }
}
```

Working with Lambda Expressions

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

namespace NewLanguageFeatures
{
    public static class Extensions
    {
        public static List<T> Append<T>(this List<T> a, List<T> b)
        {
            var newList = new List<T>(a);
            newList.AddRange(b);
            return newList;
        }

        public static bool Compare(this Customer customer1, Customer customer2)
        {
            if (customer1.CustomerID == customer2.CustomerID &&
                customer1.Name == customer2.Name &&
                customer1.City == customer2.City)
            {
                return true;
            }
            return false;
        }
    }

    public class Customer
    {
        public int CustomerID { get; private set; }
        public string Name { get; set; }
        public string City { get; set; }

        public Customer(int ID)
        {
            CustomerID = ID;
        }

        public override string ToString()
        {
            return Name + "\t" + City + "\t" + CustomerID;
        }
    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
           var customers = CreateCustomers();

           var addedCustomers = new List<Customer>
           {
         new Customer(9)    { Name = "Paolo Accorti", City = "Torino" },
         new Customer(10)   { Name = "Diego Roel", City = "Madrid" }
           };

           var updatedCustomers = customers.Append(addedCustomers);

            var newCustomer = new Customer(10)
     {
         Name = "Diego Roel",
         City = "Madrid"
     };

            foreach (var c in updatedCustomers)
            {
                if (newCustomer.Compare(c))
                {
                 Console.WriteLine("The new customer was already in the list");
                    return;
                }
            }
            Console.WriteLine("The new customer was not in the list");
        }

        static List<Customer> CreateCustomers()
        {
           return new List<Customer>
           {
            new Customer(1) { Name = "Maria Anders",        City = "Berlin"    },
            new Customer(2) { Name = "Laurence Lebihan",    City = "Marseille" },
            new Customer(3) { Name = "Elizabeth Brown",     City = "London"    },
            new Customer(4) { Name = "Ann Devon",           City = "London"    },
            new Customer(5) { Name = "Paolo Accorti",       City = "Torino"    },
            new Customer(6) { Name = "Fran Wilson",         City = "Portland"  },
            new Customer(7) { Name = "Simon Crowther",      City = "London"    },
            new Customer(8) { Name = "Liz Nixon",           City = "Portland"  }
           };
       }
    }
}
```

Using Lambda Expressions to Create Expression Trees

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

namespace NewLanguageFeatures
{
    public static class Extensions
    {
        public static List<T> Append<T>(this List<T> a, List<T> b)
        {
            var newList = new List<T>(a);
            newList.AddRange(b);
            return newList;
        }

        public static bool Compare(this Customer customer1, Customer customer2)
        {
            if (customer1.CustomerID == customer2.CustomerID &&
                customer1.Name == customer2.Name &&
                customer1.City == customer2.City)
            {
                return true;
            }
            return false;
        }
    }

    public class Customer
    {
        public int CustomerID { get; private set; }
        public string Name { get; set; }
        public string City { get; set; }

        public Customer(int ID)
        {
            CustomerID = ID;
        }

        public override string ToString()
        {
            return Name + "\t" + City + "\t" + CustomerID;
        }
    }
```

```
class Program
    {
        static void Main(string[] args)
        {
            var customers = CreateCustomers();

            foreach (var c in FindCustomersByCity(customers, "London"))
                Console.WriteLine(c);
        }

        public static List<Customer> FindCustomersByCity(
            List<Customer> customers,
            string city)
        {
            return customers.FindAll(c => c.City == city);
        }

        static List<Customer> CreateCustomers()
        {
           return new List<Customer>
           {
            new Customer(1) { Name = "Maria Anders",        City = "Berlin"    },
            new Customer(2) { Name = "Laurence Lebihan",    City = "Marseille" },
            new Customer(3) { Name = "Elizabeth Brown",     City = "London"    },
            new Customer(4) { Name = "Ann Devon",           City = "London"    },
            new Customer(5) { Name = "Paolo Accorti",       City = "Torino"    },
            new Customer(6) { Name = "Fran Wilson",         City = "Portland"  },
            new Customer(7) { Name = "Simon Crowther",      City = "London"    },
            new Customer(8) { Name = "Liz Nixon",           City = "Portland"  }
           };
       }
    }
}
```

Understanding Queries and Query Expressions

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

namespace NewLanguageFeatures
{
    public static class Extensions
    {
        public static List<T> Append<T>(this List<T> a, List<T> b)
        {
            var newList = new List<T>(a);
            newList.AddRange(b);
            return newList;
        }

        public static bool Compare(this Customer customer1, Customer customer2)
        {
            if (customer1.CustomerID == customer2.CustomerID &&
                customer1.Name == customer2.Name &&
                customer1.City == customer2.City)
            {
                return true;
            }
            return false;
        }
    }

    public class Customer
    {
        public int CustomerID { get; private set; }
        public string Name { get; set; }
        public string City { get; set; }

        public Customer(int ID)
        {
            CustomerID = ID;
        }

        public override string ToString()
        {
            return Name + "\t" + City + "\t" + CustomerID;
        }
    }
```

```
class Program
    {
        static void Main(string[] args)
        {
            Func<int, int> addOne = n => n + 1;
            Console.WriteLine("Result: {0}", addOne(5));

            Expression<Func<int, int>> addOneExpression = n => n + 1;

            var addOneFunc = addOneExpression.Compile();
            Console.WriteLine("Result: {0}", addOneFunc(5));
        }

        public static List<Customer> FindCustomersByCity(
            List<Customer> customers,
            string city)
        {
            return customers.FindAll(c => c.City == city);
        }

        static List<Customer> CreateCustomers()
        {
           return new List<Customer>
           {
        new Customer(1) { Name = "Maria Anders", City = "Berlin"},
        new Customer(2) { Name = "Laurence Lebihan",  City = "Marseille" },
        new Customer(3) { Name = "Elizabeth Brown",   City = "London"    },
        new Customer(4) { Name = "Ann Devon",         City = "London"    },
        new Customer(5) { Name = "Paolo Accorti",     City = "Torino"    },
        new Customer(6) { Name = "Fran Wilson",       City = "Portland"  },
        new Customer(7) { Name = "Simon Crowther",    City = "London"    },
        new Customer(8) { Name = "Liz Nixon",         City = "Portland"  }
           };
       }
    }
}
```

Anonymous Types and Advanced Query Creation

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

namespace NewLanguageFeatures
{
    public static class Extensions
    {
        public static List<T> Append<T>(this List<T> a, List<T> b)
        {
            var newList = new List<T>(a);
            newList.AddRange(b);
            return newList;
        }

        public static bool Compare(this Customer customer1, Customer customer2)
        {
            if (customer1.CustomerID == customer2.CustomerID &&
                customer1.Name == customer2.Name &&
                customer1.City == customer2.City)
            {   return true;     }
            return false;
        }
    }

    public class Store
    {
        public string Name { get; set; }
        public string City { get; set; }
        public override string ToString()
        {
            return Name + "\t" + City;
        }
    }
    public class Customer
    {
        public int CustomerID { get; private set; }
        public string Name { get; set; }
        public string City { get; set; }

        public Customer(int ID) { CustomerID = ID; }

        public override string ToString()
        {     return Name + "\t" + City + "\t" + CustomerID; }
    }
```

```
class Program
    {
        static void Query()
        {
            var stores = CreateStores();
            var numLondon = stores.Count(s => s.City == "London");
            Console.WriteLine("There are {0} stores in London. ", numLondon);
        }

        static void Main(string[] args)
        {
            Query();
        }

        public static List<Customer> FindCustomersByCity(
            List<Customer> customers,
            string city)
        {
            return customers.FindAll(c => c.City == city);
        }

        static List<Store> CreateStores()
        {
           return new List<Store>
           {
              new Store { Name = "Jim's Hardware",    City = "Berlin"},
              new Store { Name = "John's Books",      City = "London"},
              new Store { Name = "Lisa's Flowers",    City = "Torino"},
              new Store { Name = "Dana's Hardware",   City = "London"},
              new Store { Name = "Tim's Pets",        City = "Portland"},
              new Store { Name = "Scott's Books",     City = "London"},
              new Store { Name = "Paula's Cafe",      City = "Marseille"}
           };
        }
        static List<Customer> CreateCustomers()
        {
           return new List<Customer>
           {
              new Customer(1) { Name = "Maria Anders", City = "Berlin"},
              new Customer(2) { Name = "Laurence Lebihan",City = "Marseille"},
              new Customer(3) { Name = "Elizabeth Brown", City = "London"},
              new Customer(4) { Name = "Ann Devon",   City = "London"},
              new Customer(5) { Name = "Paolo Accorti", City = "Torino"},
              new Customer(6) { Name = "Fran Wilson", City = "Portland"},
              new Customer(7) { Name = "Simon Crowther", City = "London"},
              new Customer(8) { Name = "Liz Nixon",City = "Portland"  }
           };
       }
    }
}
```