Prasanna Wadekar 9970248407 45

SUNBEAM
Institute of Information Technology
LEARNING INITIATIVE

.NET FRAMEWORK

Session 1:

- **Introduction to .Net Framework**

Objectives of .NET Framework 4.5

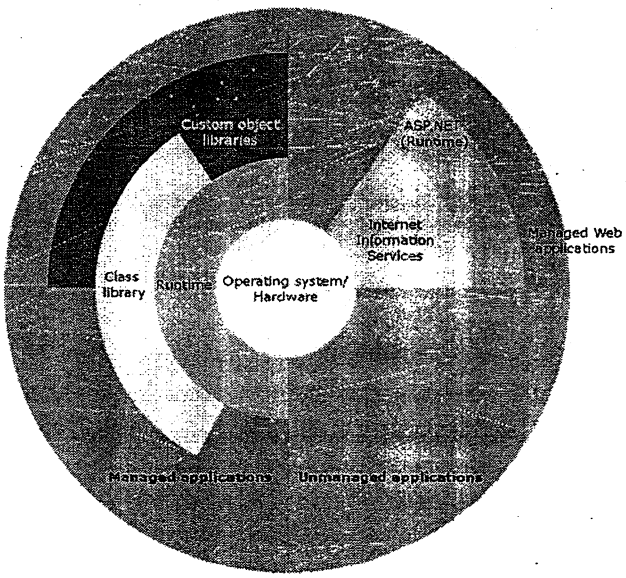
The .NET Framework is a technology that supports building and running the next generation of applications and XML Web services.

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

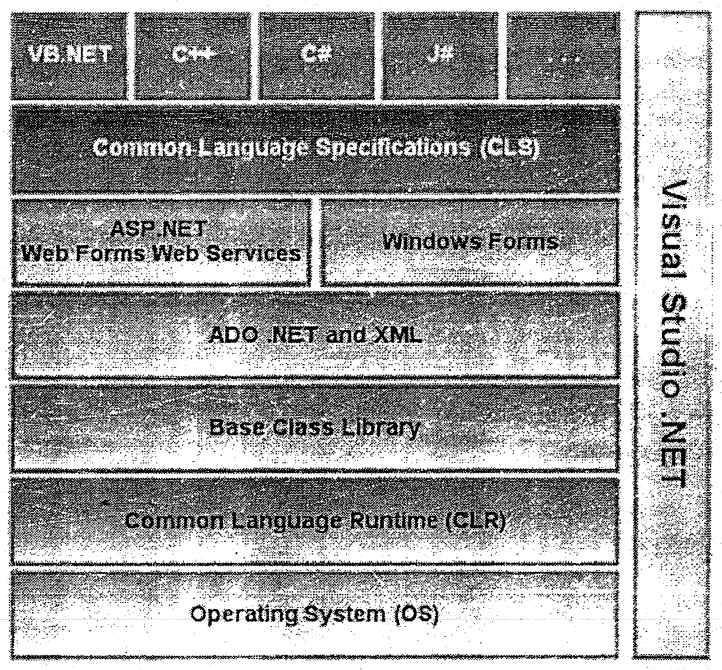
- To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- To provide a code-execution environment that minimizes software deployment and versioning conflicts.
- To provide a 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.

PRASANNA

.NET Framework Context



Dot Net Framework Components



Development in .NET Framework 4.5

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

Visual Studio is an Integrated Development Environment.

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

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

Session 2:

- **Visual Studio .NET 2013 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.

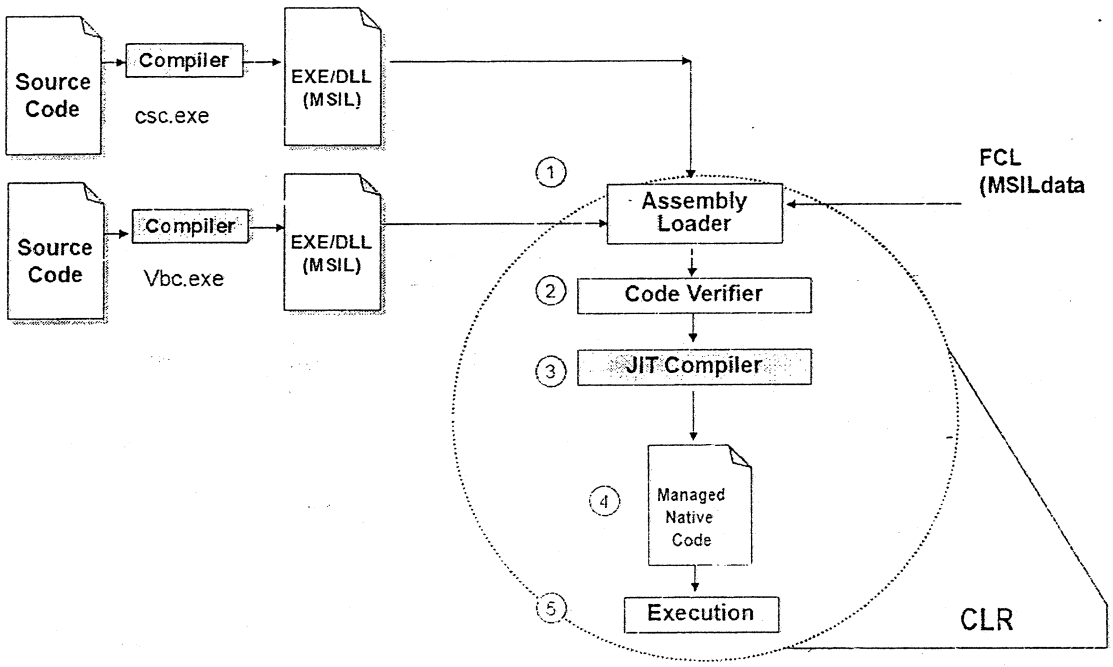
```
static void ViewNames(params string [] names)
{
 Console.WriteLine ("Names: {0}, {1}, {2}",
                    names [0], names [1], names [2]);
}
public static void Main (string [] args)
{
  ViewNames("Nitin", "Nilesh", "Rahul");
}
```

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);
}
```

Session 3:

- 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 application. 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

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.

In Microsoft's implementation of the .NET framework the garbage collector determines if an object is garbage by examining the reference type variables pointing to it. In the context of the garbage collector, reference type variables are known as "roots". Examples of roots include:

- A reference on the stack
- A reference in a static variable
- A reference in another object on the managed heap that is not eligible for garbage collection
- A reference in the form of a local variable in a method

Generations

Objects that have been around for a long time are fairly unlikely to be garbage, and objects that haven't been around for so long are considerably more likely to be garbage. By and large, this assumption proves to be true for most applications. The garbage collector leverages this fact to improve performance by implementing what are known as generations.

You can think of generations as a way of classifying objects by how long they have been alive. The garbage collector groups objects into three generations, known as generations 0, 1 and 2. Objects in generation 0 have never survived a garbage collection. They're brand new! Objects in generation 1 have survived one garbage collection, and objects in generation 2 have survived two or more collections.

When a garbage collection begins, the garbage collector can pick and choose which generations it wishes to collect. On average, collections in generation 0 will be more effective than those in generation 1, and those in generation 1 will be more effective than those in generation 2. The garbage collector can therefore decide to only collect generation 0, for example, and hopefully reclaim a substantial amount of memory without having to collect the entire managed heap.

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

Session 4:

- **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

Known as smart fields

Special methods that assign and retrieve values from the underlying data members.

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
Known as 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 Forest";
}
```

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();
  // Get int from 'database'.
    int? i = dr.GetIntFromDatabase();
    if (i.HasValue)
      Console.WriteLine("Value of 'i' is: {0}", i.Value);
    else
      Console.WriteLine("Value of 'i' is undefined.");
   // Get bool from 'database'.
    bool? b = dr.GetBoolFromDatabase();
    if (b != null)
      Console.WriteLine("Value of 'b' is: {0}", b.Value);
    else
      Console.WriteLine("Value of 'b' is undefined.");
// If the value from GetIntFromDatabase()is null,assign local variable to 100.
    int? myData = dr.GetIntFromDatabase() ?? 100;
    Console.WriteLine("Value of myData: {0}", myData.Value);
 }
static void LocalNullableVariables ()
{   // Define some local nullable types.
    int? nullableInt = 10;
    double? nullableDouble = 3.14;
    bool? nullableBool = null;
    char? nullableChar = 'a';
    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<char> nullableChar = 'a';
    Nullable<int>[] arrayOfNullableInts = new int?[10];
  }}}
```

Session 5:

- **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.
Technique that enhance power of extensibility

```
 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 ()
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");
}
```

Session 6:

- **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);
```

MemberInfo class

Base class for all members of the class

`ParameterInfo, FieldInfo, EventInfo, PropertyInfo, MethodInfo, ConstructorInfo,` etc.

Session 7:

.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

}
```

Usir

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 Emplcyee
{
     int eid;
     string ename;
     //appropriate constructor and properties for Employee Entity
}

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(3, "Neil");
List1.Add(new Employee(4, "Leela");

EmpComparer ec = new EmpComparer();
List1.Sort(ec);
foreach(Employee e in list1)
Console.WriteLine(e.Id + "---------"+ e.Name);
}
```

Prasanna Wadekar

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]);
```

Session 8:

- **Custom Generic Types**

Generic function

```
public static class MyGenericMethods
{
  public 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;
  }

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

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);
 }
}
```

Session 9:

- **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 emp = new Employee();
          Type t = emp.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 inherits 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() ;};

}
```

Session 10:

- **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 () ;}
        }
    }
}
```

```
class Program
  {  static void Main(string [] args)
        {   Account axisBankAccount = new Account(15000);
            //register Event Handlers
            axisBankAccount.UnderBalance+=PayPenalty;
            axisBankAccount.UnderBalance+=BlockBankAccount;
            axisBankAccount.OverBalance+=PayProfessionalTax;
            axisBankAccount.OverBalance+= PayIncomeTax;
            //Perform Banking Operations
            axisBankAccount.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") ;}
}
```

Windows Forms

Winforms used to build traditional Desktop applications.

The WinForm API supports RAD (Rapid Application Development) and relies extensively on properties to enable you to tailor the WinForm to your needs.

Microsoft has supplied its default libraries under the **System.Windows.Forms** namespace .The assembly being **System.dll** and **System.Windows.Forms.dll**.

Event Driven Applications

Individual user actions are translated into "events".

Events are passed one by one to application for processing.

GUI based Events

Mouse Move, Mouse Click, Mouse double Click,Key press, Button click, Menu Selection Change in Focus, Window Activation

Code-behind

Events are handled by methods that live behind visual interface

Known as "code-behind"

Developer job is to program these events.

GUI (Graphics User Interface) applications are based on the notion of forms and controls.

A form represents a window

A form contains zero or more controls

First WinForm Application

```
using System.Windows.Forms ;

public class Form1 : Form

{ public static void Main()

  {  //Run the Application

    Application.Run(new Form1());

  }

}
```

GDI +

GDI+ resides in **System.Drawing.dll** assembly.

All GDI+ classes are reside in the

System.Drawing, **System.Text**, **System.Printing**,

System.Internal , **System.Imaging**,
System.Drawing2D and **System.Design** namespaces.

The Graphics Class

The Graphics class encapsulates GDI+ drawing surfaces. Before drawing any object (for example circle, or rectangle) we have to create a surface using Graphics class. Generally we use Paint event of a Form to get the reference of the graphics. Another way is to override OnPaint method.

```
private void form1_Paint(object sender, PaintEventArgs e)
{
Graphics g = e.Graphics;
}
```

Graphics class's methods:

`DrawArc`	Draws an arc from the specified ellipse.
`DrawBezier`	Draws a cubic bezier curve.
`DrawBeziers`	Draws a series of cubic Bezier curves.
`DrawClosedCurve`	Draws a closed curve defined by an array of points.
`DrawCurve`	Draws a curve defined by an array of points.
`DrawEllipse`	Draws an ellipse.
`DrawImage`	Draws an image.
`DrawLine`	Draws a line.
`DrawPath`	Draws the lines and curves defined by a GraphicsPath.
`DrawPie`	Draws the outline of a pie section.
`DrawPolygon`	Draws the outline of a polygon.
`DrawRectangle`	Draws the outline of a rectangle.
`DrawString`	Draws a string.
`FillEllipse`	Fills the interior of an ellipse defined by a bounding rectangle.
`FillPath`	Fills the interior of a path.
`FillPie`	Fills the interior of a pie section.
`FillPolygon`	Fills the interior of a polygon defined by an array of points.
`FillRectangle`	Fills the interior of a rectangle with a Brush.
`FillRectangles`	Fills the interiors of a series of rectangles with a Brush.
`FillRegion`	Fills the interior of a Region.

GDI Objects

After creating a **Graphics** object, you can use it draw lines, fill shapes, draw text and so on. The major objects are:

Brush	Used to fill enclosed surfaces with patterns, colors, or bitmaps.
Pen	Used to draw lines and polygons, including rectangles, arcs, and pies
Font	Used to describe the font to be used to render text
Color	Used to describe the color used to render a particular object. In GDI+ color can be alpha blended

```
Pen pn = new Pen( Color.Blue );

or

Pen pn = new Pen( Color.Blue, 100 );
```

The Font Class

The **Font class** defines a particular format for text such as font type, size, and style attributes. You use font constructor to create a font.

Initializes a new instance of the **Font** class with the specified attributes.

```
Font font = new Font("Times New Roman", 26);
```

The Brush Class

The **Brush** class is an abstract base class and cannot be instantiated. We always use its derived classes to instantiate a brush object, such as SolidBrush, TextureBrush, RectangleGradientBrush, and LinearGradientBrush.

```
LinearGradientBrush lBrush = new LinearGradientBrush(rect,
Color.Red, Color.Yellow,LinearGradientMode.BackwardDiagonal);

//OR

Brush brsh = new SolidBrush(Color.Red), 40, 40, 140, 140);
```

The SolidBrush class defines a brush made up of a single color. Brushes are used to fill graphics shapes such as rectangles, ellipses, pies, polygons, and paths.

The TextureBrush encapsulates a Brush that uses an fills the interior of a shape with an image.

The LinearGradiantBrush encapsulates both two-color gradients and custom multi-color gradients.

Rectangle Structure

```
public Rectangle(Point, Size);

or

public Rectangle(int, int, int, int);
```

Point

```
Point pt1 = new Point( 30, 30);
Point pt2 = new Point( 110, 100);
```

Drawing a rectangle

```
protected override void OnPaint(PaintEventArgs pe)
{
Graphics g = pe.Graphics ;
Rectangle rect = new Rectangle(50, 30, 100, 100);
LinearGradientBrush lBrush = new LinearGradientBrush(rect,
Color.Red, Color.Yellow,LinearGradientMode.BackwardDiagonal);
g.FillRectangle(lBrush, rect);
}
```

Drawing an Arc

```
Rectangle rect = new Rectangle(50, 50, 200, 100);
protected override void OnPaint(PaintEventArgs pe)
{
Graphics g = pe.Graphics ;
Pen pn = new Pen( Color.Blue );
Rectangle rect = new Rectangle(50, 50, 200, 100);
g.DrawArc( pn, rect, 12, 84 );
}
```

Drawing a Line

```
protected override void OnPaint(PaintEventArgs pe)
{
Graphics g = pe.Graphics ;
Pen pn = new Pen( Color.Blue );
// Rectangle rect = new Rectangle(50, 50, 200, 100);
Point pt1 = new Point( 30, 30);
Point pt2 = new Point( 110, 100);
g.DrawLine( pn, pt1, pt2 );
}
```

Drawing an Ellipse

```
protected override void OnPaint(PaintEventArgs pe)
{
Graphics g = pe.Graphics ;
Pen pn = new Pen( Color.Blue, 100 );
Rectangle rect = new Rectangle(50, 50, 200, 100);
g.DrawEllipse( pn, rect );
}
```

The FillPath

```
protected override void OnPaint(PaintEventArgs pe)
{
Graphics g = pe.Graphics;
g.FillRectangle(new SolidBrush(Color.White), ClientRectangle);
GraphicsPath path = new GraphicsPath(new Point[] {
new Point(40, 140), new Point(275, 200),
new Point(105, 225), new Point(190, 300),
new Point(50, 350), new Point(20, 180), },
new byte[] {
(byte)PathPointType.Start,
(byte)PathPointType.Bezier,
(byte)PathPointType.Bezier,
(byte)PathPointType.Bezier,
(byte)PathPointType.Line,
(byte)PathPointType.Line,
});
PathGradientBrush pgb = new PathGradientBrush(path);
pgb.SurroundColors = new Color[] {
Color.Green,Color.Yellow,Color.Red, Color.Blue,
Color.Orange, Color.White, };
g.FillPath(pgb, path);
}
```

Drawing Text and Strings

```
protected override void OnPaint(PaintEventArgs pe)
{
Font fnt = new Font("Verdana", 16);
Graphics g = pe.Graphics;
g.DrawString("GDI+ World", fnt, new SolidBrush(Color.Red), 14,10);
}
```

- **.NET Data Access**

ADO.NET

A rich set of classes, interface, structures and enumerated types that manage data access from different types of data stores

ADO.NET Features

- A robust connected, disconnected Data Access Model
- Integrated XML support
- Data from varied Data Sources
- Familiarity to ADO programming model (unmanaged environment)
- Enhanced Support

Connected vs. Disconnected Architecture

	Connected	**Disconnected**
State of Connection	Constantly kept Opened	Closed once data is fetched in cache at client side.
Scalability	Limited	More
Current Data	Always available	Not up to date

ADO.NET components

.NET Data Providers

Allow users to interact with different types of data sources.

ODBC Providers

OLEDB Providers

SQL Data Providers

Oracle Data Providers

DataSets

Explicitly designed for disconnected architecture

ADO.NET Interfaces

IDbConnection : Represents an open connection to a data source.

IDbCommand : Represents an SQL statement that is executed while connected to a data source.

IDataReader : Provides a means of reading one or more forward-only streams of result sets obtained by executing a command at a data source.

IDataAdapter: Provides loosely coupled Data Access with Data Sources.

Connection Object

Has the responsibility of establishing connection with the data source.

Connection has to be explicitly closed in finally block.

```
SqlConnection conSql= new SqlConnection();
conSql.ConnectionString = "server=DatabaseServer; Initial Catalog=
Transflower; user id= sa; password=sa";
conSql.Open();
```

Command Object

Used to specify the type of interaction to perform with the database like select, insert, update and delete.

Exposes properties like:

- `CommandText`
- `CommandType`
- `Connection`

Exposes several execute methods like

- `ExecuteScalar()`
- `ExecuteReader()`
- `ExecuteNonQuery()`

```
//Inserting Data
string insertString = "insert into dept (deptId,deptName,loc) values
                      (10,'Mktg','Mumbai')";
string updateString = "update dept set deptName='Marketing' where
                                  deptName='Mktg';
string deleteString = "delete from dept where deptName='ABC'";
conSql.Open();
SqlCommand cmd= new SqlCommand(insertString, conSql);
cmd.ExecuteNonQuery();
```

Getting Single Value

```
SqlCommand cmdSql = new SqlCommand();
cmdSql.Connection = conSql;
cmdSql.CommandText = "Select Count(*) from emp";
int count = (int) cmdSql.ExecuteScalar();
MessageBox.Show(count.ToString());
```

The DataReader Object

Used to only read data in forward only sequential manner.

```
string queryStr = "Select deptName, loc from dept";
conSql.Open();
sqlCommand cmdSql = new SqlCommand(queryStr, conSql);
SqlDataReader dataReader= cmdSql.ExecuteReader();
while(dataReader.Read())
{
      MessageBox.Show("Last Name is "+ dataRead[0].ToString());
      MessageBox.Show("First Name is "+ dataRead[1].ToString());
}
dataReader.Close();
```

Adding parameters to Command

```
conSql.Open();
SqlCommand cmd = new SqlCommand("select * from emp where empNo
                  =@eno", conSql);
SqlParameter param = new SqlParameter();
      param.ParameterName = "@eno";
      param.Value = 100;
      Cmd.Parameters.Add(param);

SqlDataReader reader =cmd.ExecuteDataReader();
while(reader.Read())
{
      //Display data
}
```

Calling a Stored Procedure

```
//Stored Procedure in SQL Server
      CREATE PROCEDURE DeleteEmpRecord (@eno)
      AS delete from emp where empno = @eno;
      RETURN

//C# Code
SqlCommand cmd = new SqlCommand ("DeleteEmpRecord", conSql);
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.Add( new SqlParameter("@eno",100);
```

Multiple Queries

Multiple queries can be executed using a single command object.

```
SqlCommand cmd = new SqlCommand("select * from dept; select * from
                                                    emp", conSql);
SqlDataReader reader =cmd.ExecuteDataReader();
  //code to access first result set
bool result = dr.NextResult();
  // code to access next result set
```

Disconnected Data Access

Data Adapter

Represents a set of data commands and a database connection that are used to fill the dataset and update a sql server database.

Forms a bridge between a disconnected ADO.NET objects and a data source.

Supports methods

Fill();

Update ();

```
string sqlStr= "SELECT * FROM Orders";
SqlAdapter da = new SqlAdapter (sqlStr, con);
```

DataAdapter Properties

SelectCommand

Insertcommand

DeleteCommand

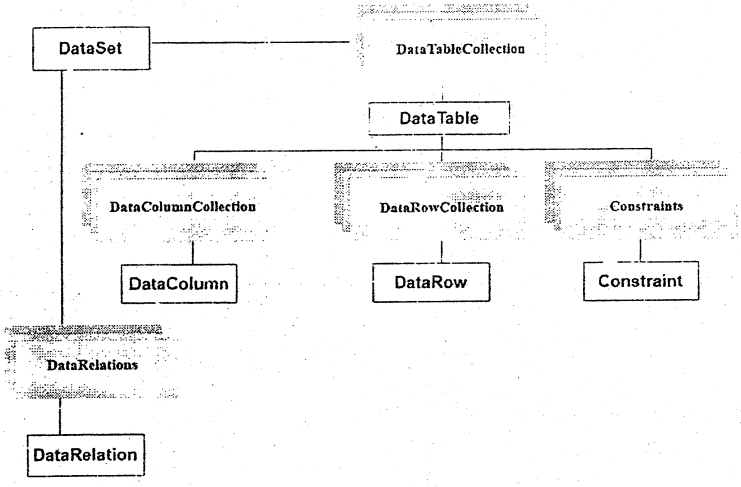
UpdateCommand

```
da.SelectCommand.CommandText = "SELECT customerID , ContactName FROM
Customers";
Sqlcommand cmd = new SqlCommand ("INSERT into Customers(CustomerID,
                                  CompanyName) VALUES (898, 'Sunbeam');
da.InsertCommand = command;
```

DataSet

Main object of disconnected architecture

Can store tables with their schema at client side.



Role of CommandBuilder Object

Automatically generates Insert, Update, Delete queires by using SelectCommand property of DataAdapter

```
SqlConnection con = new SqlConnection("server=databaseServer;
            Initial Catalog= Transflower; userid=sa; password=sa");
SqlDataAdapter da= new SqlDataAdapter
                            ("Select * from Customers", con);
SqlCommandBuilder cmdBuilder= new SqlCommandBuilder(da);
DataSet ds= new DataSet();
Da.Fill(ds, "Cusomters");
```

Constraints

Constraints restrict the data allowed in a data column or set of data columns.

Constraint classes in the System.Data namespace

`UniqueConstraint`

`ForeignKeyConstraint`

Using existing primary Key constraint

```
da.FillSchema(ds, schematype.Source, "Customers");
Or
da.MissingSchemaAction = AddWithKey;
da.Fill(ds,"Customers");
```

ADO.NET and XML

With ADO.NET it is easy to

Convert data into XML

Generate a matching XSD schema

Perform an XPath search on a result set.

Interact with an ordinary XML document through the ADO.net

DataSet XML Methods

Getxml()

GetXmlSchema()

ReadXml()

ReadXmlSchema()

WriteXml()

WriteXmlSchema()

InferXmlSchema()

Concurrency in Disconnected Architecture

Disadvantage of disconnected architecture

- Conflict can occur when two or more users retrieve and then try to update data in the same row of a table
- The second use's changes could overwrite the changes made by the first user.

Solutions -

Optimistic concurrency

Retrieves and updates just on row at a time

File Handling and Serialization

Input and Output operation in C#

I/O in C# is stream based.

Stream is flow of data from a source to a receiver through a channel.

Two types of Streams

Byte Streams
Character Streams

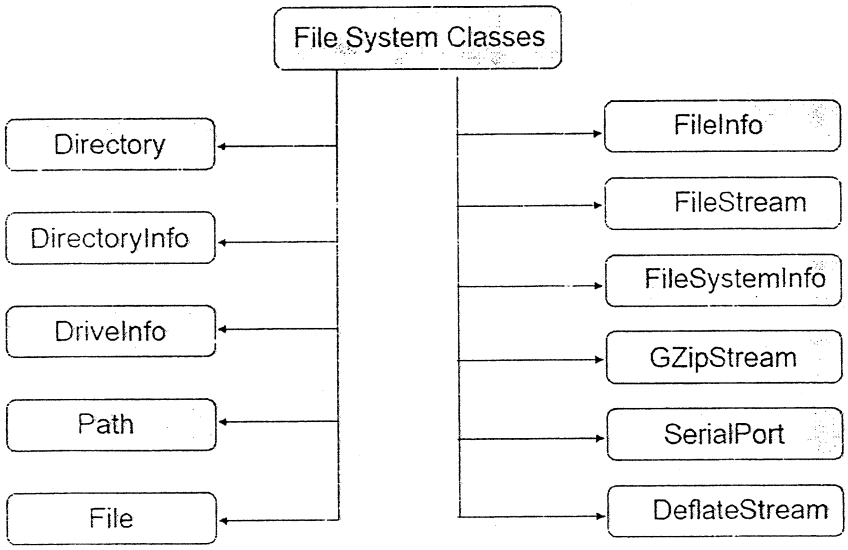
Three predefined streams are

Console.Out
Console.In
Console.Error

System.IO Namespace

File IO using System.IO:

System.IO namespace defines all the stream classes.



Directory Class

Static class and helps to manage a single directory.

```
string DirectoryName = @"C:\MyDirectory";
if (Directory.Exists(DirectoryName))
     {MessageBox.Show("Exists"); }
else
     {Directory.CreateDirectory (DirectoryName);
       MessageBox.Show("Created");
      }
```

DirectoryInfo Class

Extends the FileSystemInfo class.

Performing operations such as copying, moving, renaming, creating, and deleting.

```
string DirectoryName = @"C:\MyDirectory";
         if (Directory.Exists(DirectoryName))
            { MessageBox.Show("Exists");     }
         else
           { DirectoryInfo d= new DirectoryInfo (DirectoryName);
             d.Create();
              MessageBox.Show("Created");
           }
```

DriveInfo and Path Class

DriveInfo class

Models a drive and gives its information

```
string s = @"C:\";
DriveInfo d = new DriveInfo(s);
MessageBox.Show (d.AvailableFreeSpace.ToString (), d.Name);
```

Path class

Used to manage file and directory paths.

```
string s = @"C:\temp\MyData.text\machine.config";
MessageBox.Show (Path.GetFileName(s));
MessageBox.Show (Path.GetTempPath ());
```

File Class

Static class and helps to manage a single file

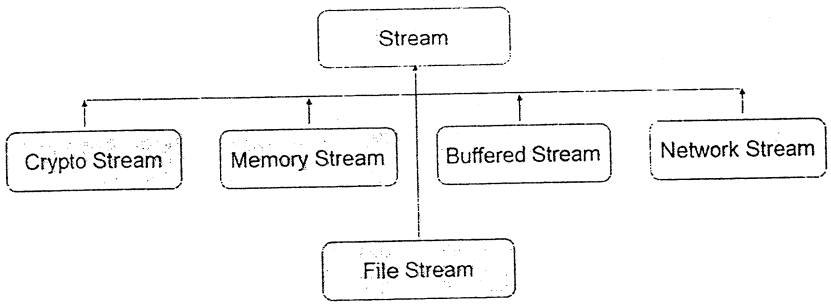
```
string FileName = @"C:\MyFile.dat";
if (File.Exists (FileName))
      {
            MessageBox.Show ("Exists")
      }
else
      {
            File.Create(FileName);
            MessageBox.Show ("Created");
      }
```

Other File System Classes

DeflatesStream
GZipStream
SerialPort
FileSystemInfo

Stream class

Stream is an abstract base class for all other stream classes
Common I/O Stream Classes



FileStream class

Used to perform operations liked read, write, open, and close operations on files on a file system.

For better performance, FileStream class buffers input and output.

BinaryReader and BinaryWriter Class

BinaryReader class is used to read data in binary files.

BinaryWriter class is used to write data to binary files.

```
FileStream fs = newFileStream ( @"C:\FileIO.txt",
FileMode.Create,FileAccess.Write);
BinaryWriter bw = new BinaryWriter (fs);
bw. Write ("Transflower");
bw. Close ();
fs.Close ();
```

The Character Stream Wrapper Classes

TextReader

Abstract class to read sequential series of characters

Inherited by

StreamReader

StringReader

TextWriter

Abstract class to write sequential series of characters

Inherited by

StreamWriter

StringWriter

StreamReader

Used to read data from stream.

```
StreamReader sr = StreamReader (@"c:\FileIO.txt ");
MessageBox.Show (sr.ReadToEnd().ToString());
Sr.Close ();
```

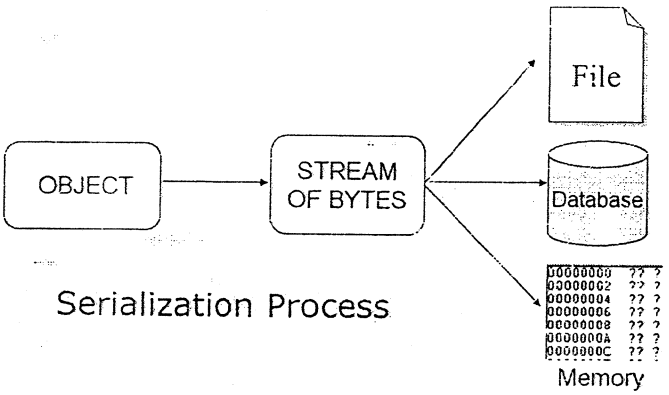
StreamWriter Class

Used to write data to a stream.

```
StreamWriter sw = new StreamWriter (@"C:\FileIO.txt");
sw.Write("Transflower");
sw.Close();
```

Serialization

Persistence is mapped by using serialization.
Serialization is a process of converting data into portable format.



Serialization Process

Type of Serialization

Binary Serialization
System.Runtime.Serialization.Formatters.Binary

XML Serialization
System.Xml.Serialization

SOAP Serialization
System.Runtime.Serialization.Formatters.Soap

[NonSerialized] attribute

Indicates that a field of a serializable class should not be serialized
To reduce the size of the serialized object add the [NonSerialized] attribute to the member
For example

```
[NonSerialized] public string Name;
```

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

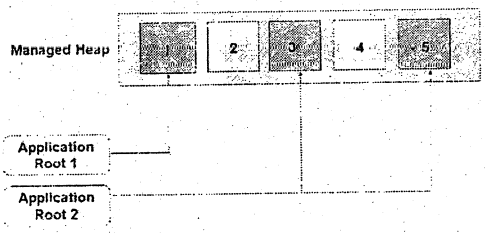
Automatic memory management provided with .NET Runtime

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

Garbage Collector

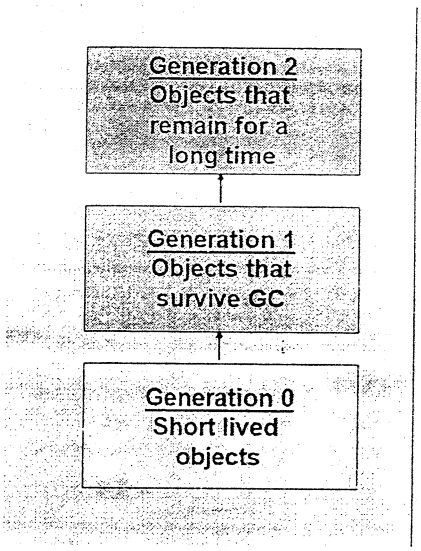
Manages the allocation and release of memory for your application.

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

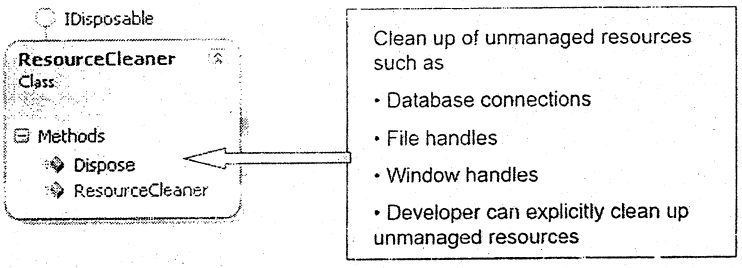
Writing Destructors in C#:

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

Explicit Resource Management

Implement IDisposable Interface.

Defines Dispose () method to release allocated unmanaged resources.



Multithreading

Multitasking

Ability to have more than one program working at the same time.

The objective to utilize the idle time of the CPU.

Multitasking can be done in 2 ways:

Non Pre-emptive

Pre-emptive

Multithreading

Multithreading is the ability of an operating system to execute different part of a program simultaneously.

When to use multithreading ?

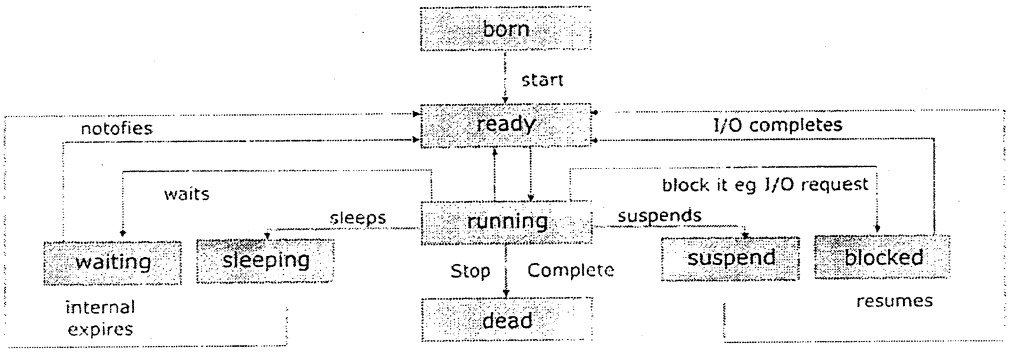
Performing operations that take a large amount of time.

Prioritization of the tasks.

Application has to wait for some event to occur.

Thread Life Cycle

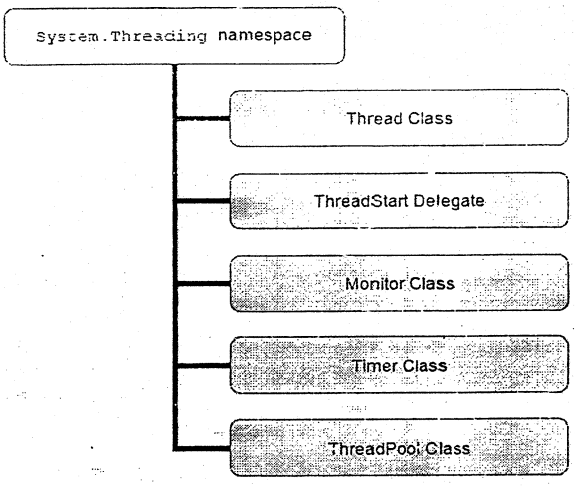
Thread is a path of Execution in a running application.



Thread Life Cycle

System. Threading namespace

Provides a number of classes and other types to support multithreading



Thread Class

```
class ThreadDemo {
private string fName, lName;
 //parameterized constructor defined here
public void Run ()
    {
     Console.WriteLine ("First Name is "+ fName);
     Thread.Sleep (500);
     Console.WriteLine ("Last Name is "+ lName);
      }
}
Class TestThreading
{
 Static void Main ()
 {
  ThreadDemo t1 = new ThreadDemo ("Rahul", "Navale");
  ThreadDemo t2 = new ThreadDemo ("Ravi", "Tambade");
  Thread first = new Thread (new ThreadStart (t1.Run));
  Thread second = new Thread (new ThreadStart (t2.Run));
      first.IsBackground = true;
      second.IsBackground = true;
      first.Start();
      second.Start();
 }
}
```

Thread Scheduling

Threads are schedule for execution using priority.
Thread priorities are defined as ThreadPriority enumeration

Highest
AboveNormal
Normal
BelowNormal
Lowest

```
Thread worker=new Thread(new ThreadStart (StartMethod));
worker.Name = "TransflowerThread";
worker.Priority = ThreadPriority.AboveNormal;
worker.Start();
```

Why Thread Synchronization?

Controlled access to resources needs to be given.
Provided by lock on the object to prevent second thread modifying it.

```
enum operation {credit, debit};
class AccountUser
{          ...
  BankAccount ac;
   public void run ()
      {
        lock (ac)
         {
             if (operation == Operation.debit)
                 ac.Debit (amt);
            else
                ac.Credit (amt);
         }
      }
}
```

Thread Synchronization

Common problems of multithreading

Deadlocks

Avoided by managing timeouts using threading classes like Monitor class.

Race condition

Avoided by using InterLocked class

Different strategies to synchronize access to instance and static methods and instance fields

Synchronized contexts

Synchronized code regions

Manual synchronization

Synchronization Context

[Synchronization] enables simple, automatic synchronization for instances of the class.
Static fields and methods are not protected from concurrent access by multiple threads

```
using System.Runtime.Remoting.Contexts;
[Synchronization]
class BankAccount
        {
            static int sCount = 0; // multiple threads can access
            int iCount = 0; // only 1 thread can access at a time
            public void Debit(float amt)
                  {
                    // … only one thread can access at a time
                  }
         }
```

Synchronized Code regions

Restricted access to a block of code, commonly called a critical section is required.

Monitor class

Controls access to these objects by granting a lock on an object to a single thread.
Expose methods like Enter (), Exit (), Wait (), Pulse () and PulseAll().

```
public void Credit ()
{   Monitor.Enter (x);
            try {. . .}
            finally {….}
    Monitor.Exit (x);
```

```
}
```

Manual Synchronization

Access to a variable shared by multiple threads is synchronized manually using InterLocked class.

InterLocked class expose methods like Add (), Increment (), Decrement (), CompareExchange(), etc. to perform atomic operations on such variables.

```
class AccountUser
  {
      static int count; . . .
      public AccountUser () {InterLocked.Increment (ref count);}
     ~AccountUser () {InterLocked.Decrement(ref count); }
   }
```

Use Mutex class

To synchronize between threads across processes Use readWriteLock

In scenarios with x single "writer" and multiple "readers"

```
private static Mutex mutex = new Mutex ();

private static void UseResources()
      {
    Mutex.WaitOne();
   // Place code to access resources her
    Thread.Sleep (500);
    Console.WriteLine ("{0}is leaving the protected area\r\n",
                    Thread.CurrentThread.Name);
    Mutex.ReleaseMutex();
      }
```

Timer Class

Used to periodically execute a method

```
public class Test
{ public static void OnTimer()
   {// background task . . .
   }
   public static void Main ()
   {
     TimerCallback dcallback = new TimerCallback (Test.OnTimer);
     long dTime = 15; //wait before the first tick (in min)
     long pTime = 130;//timer during subsequent invocations (in min)
      Timer timer = new Timer (dcallback, null, dTime, pTime);
     // do something with the timer object Timer.Dispose();
   }
}
```

Thread Pool

Provides a pool of thread that can be used **to post work items, process asynchronous I/O, wait on behalf of other threads, and process timers.**

Used to improve efficiency.

Thread Pooling should not be used when:

Need a task to have a particular priority
Have a task that might run for a long time
Need to place threads into a single-threaded apartment
Need a stable identity to be associated with the thread

Think before Multithreading

- Keeping track of and switching between threads consumes memory resources and CPU time.
- Programming with multiple threads can be complex.
- Shared resources utilization problem.

COM Interoperability

Managed Vs Unmanaged Code

Managed Code

- Targeted to CLR
- Memory managed by Garbage collector
- MSIL code
- .NET complaint

Unmanaged Code

- Code not targeted to CLR
- Memory not managed by Garbage collector
- Operating System complaint

CLR provides two mechanisms for interoperation with unmanaged code.

- Platform Invocation Services
- COM Interoperability

Unmanaged Code – unsafe block

Pointers are used in unsafe block.

Unsafe block

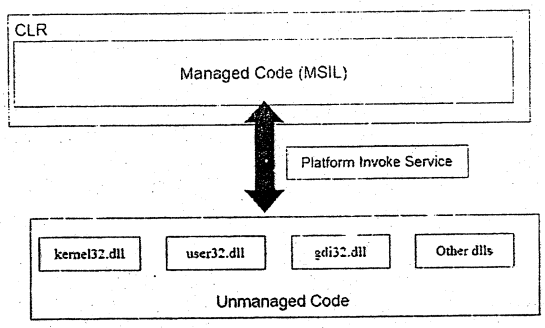
Can be whole class, struct, interface, delegate or any of members of these.

```
public unsafe struct Node
 {
    public int Value;
    public long* Left;
    public long* Top;
 }

unsafe static void main ()
 {
        Point point;
        Point* p = &point;
        p->x = 10;
       p->y = 30;
 }
```

Platform Invoke Service

Used to invoke unmanaged functions (Win32 API) from dlls.



How PInvoke works?

1. Locates the dll containing the required Windows API function.
2. Loads the dll in the memory.
3. Locates the address of the function in the memory and pushes the argument on stack.
4. Transfers the control to unmanaged function for execution.
5. Returns the exception from these functions to be handled by managed code.

Using PInvoke

```
using System.Runtime.InteropServices;
public class TestWin32
{
[DllImport("user32.dll", CharSet=CharSet.Auto,
EntryPoint="MessageBoxA"),CharSet.Ansi,
ExactSpelling=false,CallingConvention.StdCall]
public static extern int MsgBox(int hWnd, string text, string
                                                caption, unit type);
}

private void button3_Click (object sender, EventArgs e)
{
  TestWin.MsgBox(0, "Hello World", "Platform Invoke Sample", 0);
}
```

Interoperability

Reusing COM components into the .NET application irrespective of the technology in which it is developed.

Ensures Installation of the .NET framework does not affect existing COM applications.

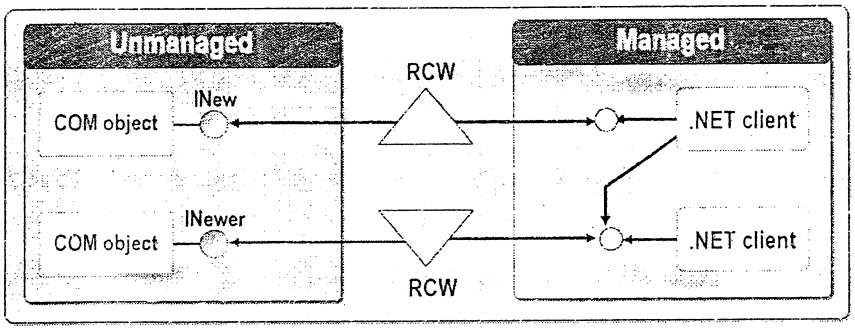
COM components and .NET applications can communicate with each other.

COM component vs. .NET components

Component	COM	.NET
Coding model	Interface	Object
Identifier	GUID	Strong name
Compatibility	Binary	Type
Type definition	Type Library	Metadata
Versioning	No	Yes

Using COM component in .NET environment

- Ensure that required COM components is registered with windows registry.
- Add Reference of COM component in .NET application
- Use component by importing namespace as usual.



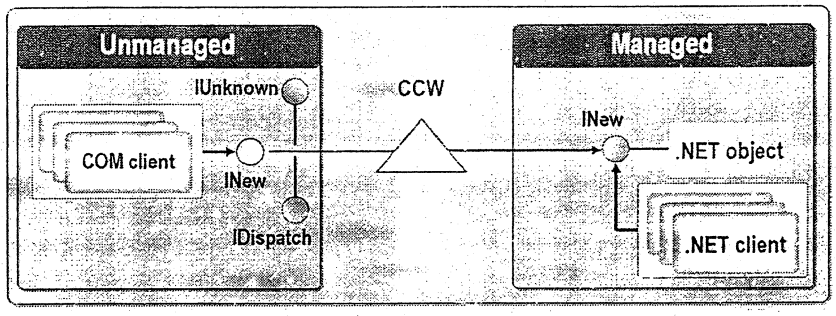
Using .NET component form COM client

Set the attribute in AssemblyInfo.cs file

[assembly:ComVisible(true)].

Add a reference of .tlb (.NET component).

Use .tlb component like activeX dll.



.NET Remoting

- A mechanism for communicating between objects which are not in the same process.
- Allows Inter Process communication (IPC) as well as Remote Procedural Call (RPC).
- Provides a framework that allows objects to interact with each other across application domains.
- Provides a number of services, including object activation and object lifetime support.

What is Remote Object ?

Any object outside the application domain of the caller application should be considered remote, where the object will be reconstructed. Local objects that cannot be serialized cannot be passed to a different application domain, and are therefore non remotable.

Any object can be changed into a remote object by deriving it from `MarshalByRefObject`, or by making it serializable either by adding the `[Serializable]` tag or by implementing the `ISerializable` interface.

Types of Remotable Object

- Single Call
- Singleton Objects
- Client-Activated Objects (CAO)
-

Domains

In .NET, when an application is loaded in memory, a process is created, and within this process, an application domain is created.

The application is actually loaded in the application domain.

If this application communicates with another application, it has to use Remoting because the other application will have its own domain, and across domains, object cannot communicate directly.

Different application domains may exist in same process, or they may exist in different processes.

Contexts

The .NET runtime further divides the application domain into contexts.

All applications have a default context in which objects are constructed, unless otherwise instructed.

A context, like an application domain, forms a .NET Remoting boundary. Access requests must be marshaled across contexts.

Proxies

When a call is made between objects in the same Application Domain, only a normal local call is required; however, a call across Application Domains requires a remote call. \

In order to facilitate a remote call, a proxy is introduced by the .NET framework at the client side. This proxy is an instance of the `TransparentProxy` class, directly available to the client to communicate with the remote object.

The proxy object ensures that all calls made on the proxy are forwarded to the correct remote object instance.

In .NET Remoting, the proxy manages the marshaling process and the other tasks required to make cross-boundary calls. The .NET Remoting infrastructure automatically handles the creation and management of proxies.

Channels

Channel provides a mechanism by which a stream of bytes is sent from one point to the other (client to server etc.).

> NET Remoting channels existing in `System.Runtime.Remoting.Channels`, the `TcpChannel` and the `HttpChannel`.

Serialization Formatters

.NET Remoting uses serialization to copy marshal-by-value objects, and to send reference of objects which are marshal-by-reference, between application domains.

The .NET Framework supports two kinds of serialization:

Binary serialization and XML serialization.

Formatters are used for encoding and decoding the messages before they are transported by the channel.

There are two native formatters in the .NET runtime:

Binary : `System.Runtime.Serialization.Formatters.Binary`

SOAP: `System.Runtime.Serialization.Formatters.Soap`

How to Implement Remoting

Create a Remotable Object

```
public interface RemoteInterface
 {
    int Sum(int iNum1, int iNum2);
    string Hello(String name);
 }

public class RemoteImpl : MarshalByRefObject, RemoteInterface
    {
      public RemoteImpl()
      { Console.WriteLine("RemoteImpl Object created!"); }

    public int Sum(int iNum1, int iNum2)
      { Console.WriteLine("Sum() called!");
        return iNum1 + iNum2;
      }
    public string Hello(string name)
      {
         Console.WriteLine("Hello() called!");
         return "Hello " + name;
      }
    }
```

Create a Server to Expose the Remotable Object

Server Application configuration File (app.config)

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown
          mode="SingleCall"
          type="RemoteImplLib.RemoteImpl, RemoteImplLib"
          objectUri="dac"/>
      </service>
      <channels>
        <channel ref="tcp" port="8467"/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Server Class

```
class Server
 {
   static void Main(string[] args)
   {  RemotingConfiguration.Configure("Server.exe.config", true);
      Console.WriteLine("Remoting Server Ready!!\n");
      Console.ReadLine();
   }
  }
```

Prasanna Wadekar

Create a Client to Use the Remotable Object

Client Application configuration File

```
<?xml version="1.0"?>
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown type="RemoteImplLib.RemoteImpl, RemoteImplLib"
                   url="tcp://localhost:8467/dac"/>
      </client>
      <channels>
        <channel ref="tcp" port="0"/>
      </channels>
    </application>
  </system.runtime.remoting>
```

Client Class

```
using System.Runtime.Remoting;
using InterfaceLib;
namespace Client
{  class Client
    {
        static void Main(string[] args)
        {
         RemotingConfiguration.Configure("Client.exe.config", true);
         RemoteInterface obj;
         obj = (RemoteInterface)Activator.GetObject(
                         typeof(RemoteInterface),
                        "tcp://localhost:8467/dac");
            Console.WriteLine("Remote Client Ready!\n");
            Console.ReadLine();
            int iRes = obj.Sum(12, 4);
            Console.WriteLine("Sum : " + iRes);
            Console.ReadLine();
            string hello = obj.Hello("DAC");
            Console.WriteLine("Result : " + hello);
        }
    }
}
```