SATHYABAMA UNIVERSITY

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE MATERIAL

Subject Name : C# AND .NET UNIT II Subject Code : SCSX1008

2.1 Assemblies
2.2 Versioning
2.3 Attributes
2.4 Reflection
2.5 Viewing metadata
2.6 Type discovery – Reflecting on a type
2.7 Marshaling
2.8 Remoting – Understanding server object types – Specifying a server with an interface
2.8.1 Building a server
2.8.2 Building the client
2.9 Exception handling
2.10 Garbage collector.

2.1 ASSEMBLIES

Assemblies form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions for a .NET-based application. Assemblies take the form of an executable (.exe) file or dynamic link library (.dll) file, and are the building blocks of the .NET Framework. They provide the common language runtime with the information it needs to be aware of type implementations.

Assemblies can contain one or more modules. For example, larger projects may be planned in such a way that several individual developers work on separate modules, all coming together to create a single assembly.

Assemblies have the following properties:

- Assemblies are implemented as .exe or .dll files.
- Assembly can be shared between applications by putting it in the global assembly cache. Assemblies must be strong-named before they can be included in the global assembly cache.
- Assemblies are only loaded into memory if they are required. If they are not used, they are not loaded. This means that assemblies can be an efficient way to manage resources in larger projects.
- Obtain information about an assembly by using reflection. For more information, see Reflection.
- If you want to load an assembly only to inspect it, use a method such as ReflectionOnlyLoadFrom.

Two Types : **Private Assemblies**, used for single programs, and
Global Assemblies shared among several applications.

Private Assemblies

Intended use by single applications. Building modules to group common functionality.

- Location is specified at compile time
- PATH is not checked while looking up files, neither set by Control Panel 'System' configuration nor set in a Console Window.
- Identified by name and version if required. But only one version at a time.
- Digital signature possible to ensure that it can't be tampered.
- Get smaller EXE files.
- Dynamic linking, i.e. loading on demand.

Global Assemblies

Publicly sharing functionality among different application.

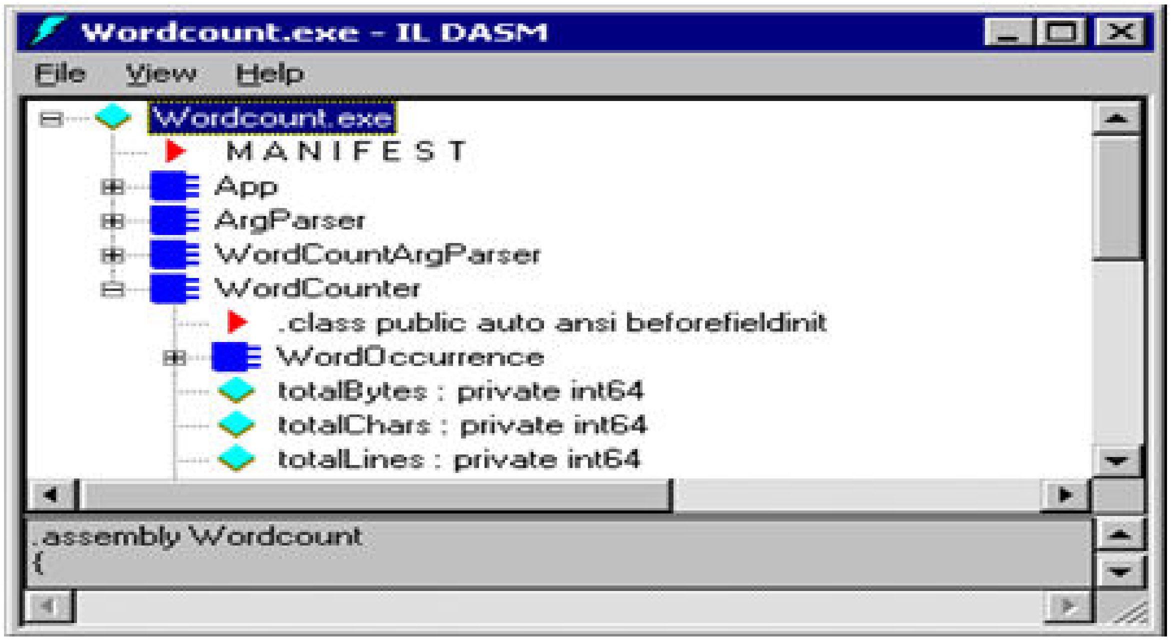
- Located in Global Assembly Cache (GAC).
- Identified by globally unique name and version.
- Digital signature to ensure that it can't be tampered.
- Get smaller EXE files.
- Dynamic linking, i.e. loading on demand.

View Assemblies - The Intermediate Language Disassembler (ILDASM)

Display metadata of one of your .NET programs or libraries by the use of the ILDASM tool:

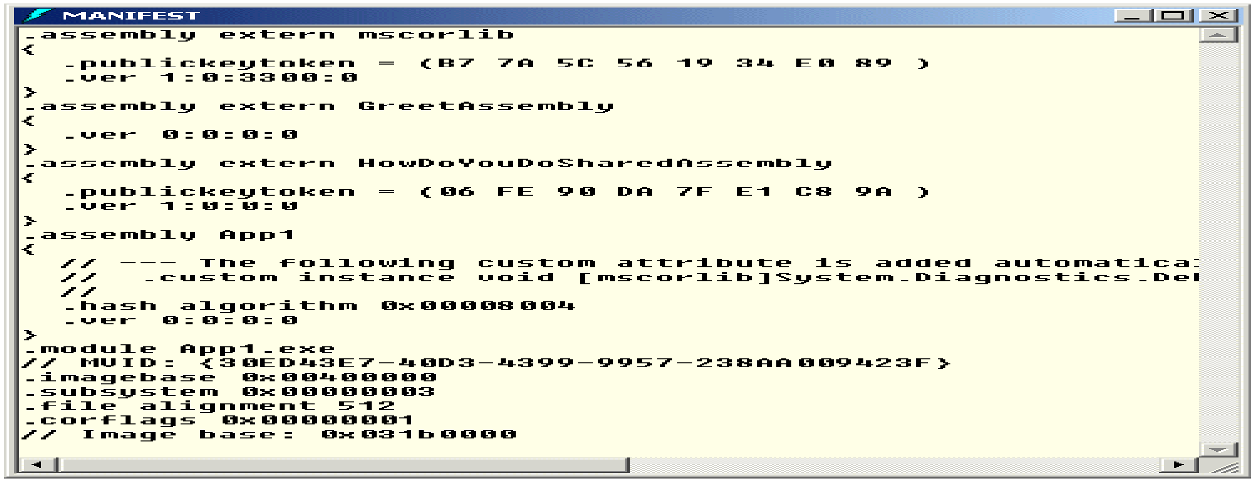
```
C:\SS\> ildasm app1.exe
```

We can see the assembly's metadata with all the methods and types in a tree representation. If we click on ' M A N I F E S T ' we get a window, which shows the manifest information whcich is shown in Fig 2.1 and 2.2.


Wordcount.exe - IL DASM
File
View
Help
Wordcount.exe
MANIFEST
App
ArgParser
WordCountArgParser
WordCounter
.class public auto ansi beforefieldinit
WordOccurrence
totalBytes : private int64
totalChars : private int64
totalLines : private int64

```
.assembly Wordcount
{
```

Figure 2.1:Manifest Information



```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 1:0:3300:0
}
.assembly extern GreetAssembly
{
  .ver 0:0:0:0
}
.assembly extern HowDoYouDoSharedAssembly
{
  .publickeytoken = (06 FE 90 DA 7F E1 C8 9A )
  .ver 1:0:0:0
}
.assembly App1
{
  // --- The following custom attribute is added automatica
  //  .custom instance void [mscorlib]System.Diagnostics.De
  //
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module App1.exe
// MVID: {30ED43E7-40D3-4399-9957-238AA009423F}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x031b0000
```

Figure 2.2:Manifest Information

Table 2.1 Ggraphic symbols used in Assembly

Symbol	Meaning
🛡	Namespace
🟦	Class
I	Interface
E	Enum
🟪	Method
S	Static method
◆	Field
▼	Event
▲	Property

CREATE PRIVATE ASSEMBLY
Consider :
Hello.cs
There is a simple class providing a method to print out a 'Hello':

```
namespace csharp.test.app.greet
 {
            public class Hello {
                        public void SayHello() {
                                        System.Console.WriteLine("Hello my friend, I am a DLL");
                                                }
                                   }
}
```

GoodBye.cs
Similar to Hello.cs but prints a 'Good bye':

```
         namespace csharp.test.app.greet
         {
                           public class GoodBye {
                                    public void SayGoodBye() {
                                        System.Console.WriteLine("Good bye, I am a DLL too");
                                                            }
                                                }
      }
```

Hello.cs and GoodBye.cs will be put into a single Private Assembly. They must be in the same namespace.

HowDoYouDo.cs

We are going to implement this source file as a Global Assembly:

```
using System.Reflection;
[assembly:AssemblyKeyFile("app.snk")]              //attributes
[assembly:AssemblyVersion("1.0.0.0")]              //attributes
      namespace csharp.test.app
      {
            public class HowDoYouDo {
                  public void SayHowDoYouDo() {
                  System.Console.WriteLine("How do you do, I am a Global Assembly");
                                          }
                                    }
      }
```

With the Attributes at the top we specify the key file used to generate a hash code and to declare the version.

Compile Classes to DLLs - The CSharp Compiler (CSC)

To compile our source files we use the C# Compiler (csc):

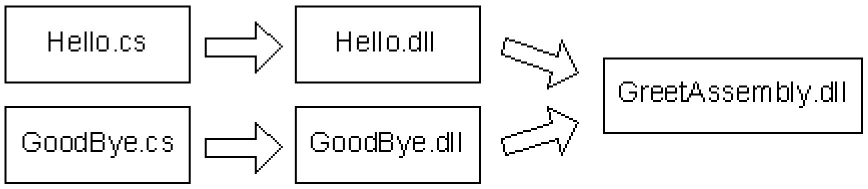
```
DotNet> csc /debug /t:module /out:bin\Hello.dll Hello.cs
DotNet> csc /debug /t:module /out:bin\GoodBye.dll GoodBye.cs
DotNet> csc /debug /t:module /out:bin\HowDoYouDo.dll HowDoYouDo.cs
```

- the /debug includes debug information.
- the /t (target) switch lets us create a DLL.
- We are writing all our DLLs into a bin folder.

Group DLLs in a Private Assembly - The Assembly Linker (AL)

Combine Hello.dll with GoodBye.dll and put them into a Private Assembly i.e GreetAssembly.dll, this is Step 1 in the Sample Application.

```
DotNet> al /t:library /out:bin\GreetAssembly.dll bin\Hello.dll bin\GoodBye.dll
```

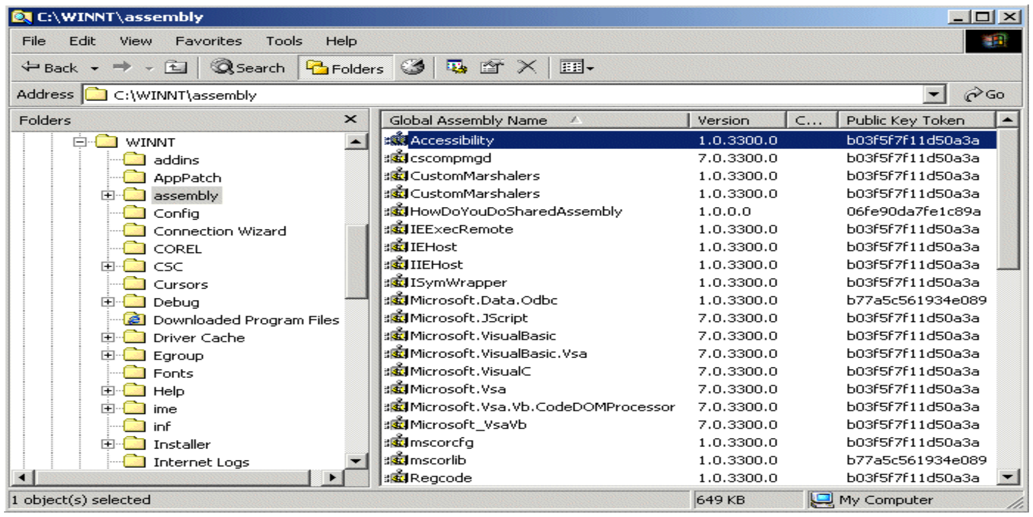


With /t (target) generate here a library referencing the two other DLLs. This is also called a Multi-Module Assembly. Again, we store all the binaries in a bin folder.

CREATE GLOBAL ASSEMBLY

Global Assemblies, also called Shared Assemblies, are used to provide globally accessible libraries for different applications. These applications may be used by a single vendor or may be publicly available to other vendors and companies. We do not have control who is using our Global Assemblies.

Global Assemblies are presented in the Global Assembly Cache (GAC) which can be displayed by Windows Explorer like in Fig 2.3.



Global Assembly Name	Version	C...	Public Key Token
Accessibility	1.0.3300.0		b03f5f7f11d50a3a
cscompmgd	7.0.3300.0		b03f5f7f11d50a3a
CustomMarshalers	1.0.3300.0		b03f5f7f11d50a3a
CustomMarshalers	1.0.3300.0		b03f5f7f11d50a3a
HowDoYouDoSharedAssembly	1.0.0.0		06fe90da7fe1c89a
IEExecRemote	1.0.3300.0		b03f5f7f11d50a3a
IEHost	1.0.3300.0		b03f5f7f11d50a3a
IIEHost	1.0.3300.0		b03f5f7f11d50a3a
ISymWrapper	1.0.3300.0		b03f5f7f11d50a3a
Microsoft.Data.Odbc	1.0.3300.0		b77a5c561934e089
Microsoft.JScript	7.0.3300.0		b03f5f7f11d50a3a
Microsoft.VisualBasic	7.0.3300.0		b03f5f7f11d50a3a
Microsoft.VisualBasic.Vsa	7.0.3300.0		b03f5f7f11d50a3a
Microsoft.VisualC	7.0.3300.0		b03f5f7f11d50a3a
Microsoft.Vsa	7.0.3300.0		b03f5f7f11d50a3a
Microsoft.Vsa.Vb.CodeDOMProcessor	7.0.3300.0		b03f5f7f11d50a3a
Microsoft_VsaVb	7.0.3300.0		b03f5f7f11d50a3a
mscorcfg	1.0.3300.0		b03f5f7f11d50a3a
mscorlib	1.0.3300.0		b77a5c561934e089
Regcode	1.0.3300.0		b03f5f7f11d50a3a

Figure 2.3 : Global Assembly

The GAC can be found always in the 'assembly' sub folder inside %SystemRoot%, i.e. WINNT for Windows 2000 and Windows NT.

2.2 VERSIONING

Version information for an assembly consists of the following four values:

Major Version
Minor Version
Build Number
Revision

We can specify all the values or can default the Build and Revision Numbers
by using the '*' as shown below:

```
[assembly: AssemblyVersion("1.0.*")]
```

•The build number is the number of days since 01/01/2000
•The revision number if the number of 2 seconds periods since 00:00 of this day

This feature is great if we need some atomic versionning of a particular library for instance. Each program using this library has its own version and don't risk to break working feature in future release of this library. And if we want to force the use of a newer version of a library within a particular application, we can use some assembly redirection.

```
using System;
using System.Reflection;
[assembly:AssemblyVersion("1.1.0.0")]
class Example
{
   static void Main()
   {
      Console.WriteLine("The version of the currently executing assembly is: {0}",
         Assembly.GetExecutingAssembly().GetName().Version);

      Console.WriteLine("The version of mscorlib.dll is: {0}",
         typeof(String).Assembly.GetName().Version);
   }
}
```

Output :

```
The version of the currently executing assembly is: 1.1.0.0
The version of mscorlib.dll is: 2.0.0.0
```

2.3 ATTRIBUTES

An attribute is a mechanism to add declarative information to code elements (types, members, assemblies or modules) beyond the usual predefined keywords. They are saved with the metadata of the object and can be used to describe the code at runtime or to affect application behaviour at run time through the use of reflection.

An *attribute* is an object that represents data you want to associate with an element in your program. The element to which you attach an attribute is referred to as the *target* of that attribute.

Intrinsic Attributes

Attributes come in two flavors: *intrinsic* and *custom*. *Intrinsic* attributes are supplied as part of the Common Language Runtime (CLR), and they are integrated into .NET. *Custom* attributes are attributes you create for your own purposes.

Most programmers will use only intrinsic attributes, though custom attributes can be a powerful tool when combined with reflection.

Attribute Targets

If you search through the CLR, you'll find a great many attributes. Some attributes are applied to an assembly, others to a class or interface, and some, such as [WebMethod], to class members. These are called the *attribute targets*. Possible attribute targets are given in Table 2.2.

Table 2.2 Possible attribute targets

Member Name	Usage
All	Applied to any of the following elements: assembly, class, class member, delegate, enum, event, field, interface, method, module, parameter, property, return value, or struct
Assembly	Applied to the assembly itself
Class	Applied to instances of the class
ClassMembers	Applied to classes, structs, enums, constructors, methods, properties, fields, events, delegates, and interfaces
Constructor	Applied to a given constructor
Delegate	Applied to the delegated method
Enum	Applied to an enumeration
Event	Applied to an event
Field	Applied to a field

Interface	Applied to an interface
Method	Applied to a method
Module	Applied to a single module
Parameter	Applied to a parameter of a method
Property	Applied to a property (both get and set, if implemented)
ReturnValue	Applied to a return value
Struct	Applied to a struct

Applying Attributes

You apply attributes to their targets by placing them in square brackets immediately before the target item. You can combine attributes, either by stacking one on top of another:

```
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile(".\\keyFile.snk")]
```

or by separating the attributes with commas:

```
[assembly: AssemblyDelaySign(false),
   assembly: AssemblyKeyFile(".\\keyFile.snk")]
```

The key fact about intrinsic attributes is that you know when you need them; the task will dictate their use.

Custom Attributes

You are free to create your own custom attributes and use them at runtime as you see fit. Suppose, for example, that your development organization wants to keep track of bug fixes. You already keep a database of all your bugs, but you'd like to tie your bug reports to specific fixes in the code.

You might add comments to your code along the lines of:

```
// Bug 323 fixed by Jesse Liberty 1/1/2005.
```

This would make it easy to see in your source code, but there is no enforced connection to Bug 323 in the database. A custom attribute might be just what you need. You would replace your comment with something like this:

```
[BugFixAttribute(323,"Jesse Liberty","1/1/2005")
Comment="Off by one error"]
```

You could then write a program to read through the metadata to find these bug-fix notations and update the database. The attribute would serve the purposes of a comment, but would also allow you to retrieve the information programmatically through tools you'd create.

Declaring an Attribute

Attributes, like most things in C#, are embodied in classes. To create a custom attribute, you derive your new custom attribute class from System.Attribute:

```
public class BugFixAttribute : System.Attribute
```

You need to tell the compiler with which kinds of elements this attribute can be used (the attribute target). You specify this with (what else?) an attribute:

```
[AttributeUsage(AttributeTargets.Class |
   AttributeTargets.Constructor |
   AttributeTargets.Field |
   AttributeTargets.Method |
   AttributeTargets.Property,
   AllowMultiple = true)]
```

AttributeUsage is an attribute applied to attributes: a meta-attribute. It provides, if you will, meta-metadata--that is, data about the metadata. For the AttributeUsage attribute constructor, you pass two arguments. The first argument is a set of flags that indicate the target--in this case, the class and its constructor, fields, methods, and properties. The second argument is a flag that indicates whether a given element might receive more than one such attribute. In this example, AllowMultiple is set to true, indicating that class members can have more than one BugFixAttribute assigned.

Naming an Attribute

The new custom attribute in this example is named BugFixAttribute. The convention is to append the word Attribute to your attribute name. The compiler supports this by allowing you to call the attribute with the shorter version of the name. Thus, you can write:

```
[BugFix(123, "Jesse Liberty", "01/01/05", Comment="Off by one")]
```

The compiler will first look for an attribute named BugFix and, if it does not find that, will then look for BugFixAttribute.

Constructing an Attribute

Every attribute must have at least one constructor. Attributes take two types of parameters, *positional* and *named*. In the BugFix example, the programmer's name and the date are positional parameters, and comment is a named parameter. Positional parameters are passed in through the constructor and must be passed in the order declared in the constructor:

```
public BugFixAttribute(int bugID, string programmer,
string date)
{
   this.bugID = bugID;
   this.programmer = programmer;
   this.date = date;
}
```

Named parameters are implemented as properties:

```
public string Comment
{
    get
    {
       return comment;
    }
    set
    {
       comment = value;
    }
}
```

It is common to create read-only properties for the positional parameters :

```
public int BugID
{
    get
    {
       return bugID;
    }
}
```

Using an Attribute

Once you have defined an attribute, you can put it to work by placing it immediately before its target. To test the BugFixAttribute of the preceding example, the following program creates a simple class named MyMath and gives it two functions. You'll assign BugFixAttributes to the class to record its code-maintenance history:

```
[BugFixAttribute(121,"Jesse Liberty","01/03/05")]
[BugFixAttribute(107,"Jesse Liberty","01/04/05",
   Comment="Fixed off by one errors")]
public class MyMath
```

These attributes will be stored with the metadata. Example 18-1 shows the complete program.

Example 18-1: Working with custom attributes

```
namespace Programming_CSharp
{
  using System;
  using System.Reflection;

  // create custom attribute to be assigned to class members
  [AttributeUsage(AttributeTargets.Class |
     AttributeTargets.Constructor |
     AttributeTargets.Field |
     AttributeTargets.Method |
     AttributeTargets.Property,
     AllowMultiple = true)]
  public class BugFixAttribute : System.Attribute
  {
    // attribute constructor for
    // positional parameters
    public BugFixAttribute
      (int bugID,
      string programmer,
      string date)
    {
      this.bugID = bugID;
      this.programmer = programmer;
      this.date = date;
    }
```

```
// accessor
public int BugID
{
  get
  {
    return bugID;
  }
}

// property for named parameter
public string Comment
{
  get
  {
    return comment;
  }
  set
  {
    comment = value;
  }
}

// accessor
public string Date
{
  get
  {
    return date;
  }
}

// accessor
public string Programmer
{
  get
  {
    return programmer;
  }
}

// private member data
private int     bugID;
```

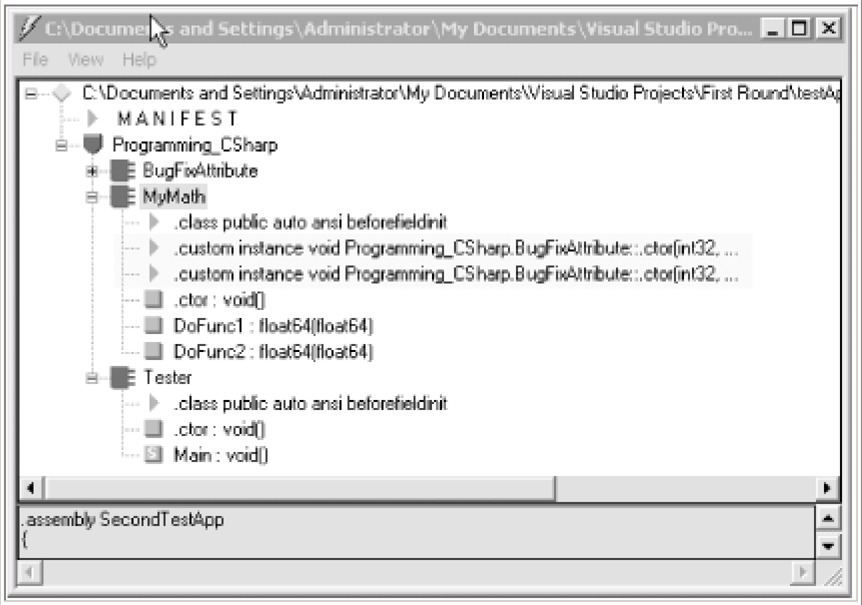
```
    private string  comment;
    private string  date;
    private string  programmer;
  }


  // ********* assign the attributes to the class ********

  [BugFixAttribute(121,"Jesse Liberty","01/03/05")]
  [BugFixAttribute(107,"Jesse Liberty","01/04/05",
     Comment="Fixed off by one errors")]
  public class MyMath
  {

    public double DoFunc1(double param1)
    {
      return param1 + DoFunc2(param1);
    }

    public double DoFunc2(double param1)
    {
      return param1 / 3;
    }

  }

  public class Tester
  {
    public static void Main( )
    {
      MyMath mm = new MyMath( );
      Console.WriteLine("Calling DoFunc(7). Result: {0}",
        mm.DoFunc1(7));
    }
  }
}
```

Output :

```
Calling DoFunc(7). Result: 9.3333333333333339
```

As you can see, the attributes had absolutely no impact on the output. In fact, for the moment, you have only my word that the attributes exist at all. A quick look at the metadata using ILDasm does reveal that the attributes are in place, however, as shown in Figure 2.4.

Figure 2.4. The metadata in the assembly



```
C:\Documents and Settings\Administrator\My Documents\Visual Studio Pro...
File  View  Help

C:\Documents and Settings\Administrator\My Documents\Visual Studio Projects\First Round\testAp
  MANIFEST
  Programming_CSharp
    BugFixAttribute
    MyMath
      .class public auto ansi beforefieldinit
      .custom instance void Programming_CSharp.BugFixAttribute::.ctor(int32, ...
      .custom instance void Programming_CSharp.BugFixAttribute::.ctor(int32, ...
      .ctor : void()
      DoFunc1 : float64(float64)
      DoFunc2 : float64(float64)
    Tester
      .class public auto ansi beforefieldinit
      .ctor : void()
      Main : void()

.assembly SecondTestApp
{
```

2.4 REFLECTION

The classes in the Reflection namespace, along with the **System.Type** and **System.TypedReference** classes, provide support for examining and interacting with the metadata.

Reflection is generally used for any of **four tasks**:

Viewing metadata :

This might be used by tools and utilities that wish to display metadata.

Performing type discovery :

This allows you to examine the types in an assembly and interact with or instantiate those types. This can be useful in creating custom scripts. For example, you might want to allow your users to interact with your program using a script language, such as JavaScript, or a scripting language you create yourself.

Late binding to methods and properties :
This allows the programmer to invoke properties and methods on objects dynamically instantiated based on type discovery. This is also known as dynamic invocation.

Creating types at runtime (Reflection Emit) :
The ultimate use of reflection is to create new types at runtime and then to use those types to perform tasks. You might do this when a custom class, created at runtime, will run significantly faster than more generic code created at compile time.

2.5 VIEWING METADATA :

```
public static void Main( )
{

   // get the member information and use it to
   // retrieve the custom attributes
   System.Reflection.MemberInfo inf = typeof(MyMath);
   object[] attributes;
   attributes =
     inf.GetCustomAttributes(
         typeof(BugFixAttribute), false);

   // iterate through the attributes, retrieving the
   // properties
   foreach(Object attribute in attributes)
   {
      BugFixAttribute bfa = (BugFixAttribute) attribute;
      Console.WriteLine("\nBugID: {0}", bfa.BugID);
      Console.WriteLine("Programmer: {0}", bfa.Programmer);
      Console.WriteLine("Date: {0}", bfa.Date);
      Console.WriteLine("Comment: {0}", bfa.Comment);
   }
}
```

Output:

```
BugID: 121 Programmer: Jesse Liberty Date: 01/03/05 Comment:
BugID: 107 Programmer: Jesse Liberty Date: 01/04/05 Comment: Fixed off by one errors
```

2.6 TYPE DISCOVERY

REFLECTING ON AN ASSEMBLY

```
namespace Programming_CSharp
{
   using System;
   using System.Reflection;

   public class Tester
   {
      public static void Main( )
      {
         // what is in the assembly
         Assembly a = Assembly.Load("Mscorlib.dll");
         Type[] types = a.GetTypes( );
         foreach(Type t in types)
         {
               Console.WriteLine("Type is {0}", t);
         }
         Console.WriteLine(
            "{0} types found", types.Length);
      }
   }
}
```

Output :

```
Type is System.TypeCode
Type is System.Security.Util.StringExpressionSet
Type is System.Runtime.InteropServices.COMException
Type is System.Runtime.InteropServices.SEHException
Type is System.Reflection.TargetParameterCountException
Type is System.Text.UTF7Encoding
Type is System.Text.UTF7Encoding+Decoder
Type is System.Text.UTF7Encoding+Encoder
Type is System.ArgIterator
1426 types found   ……………………………..
```

This example obtained an array filled with the types from the Core Library and printed them one by one. The array contained 1,426 entries on my machine.

REFLECTING ON A TYPE :

```
namespace Programming_CSharp
{
   using System;
   using System.Reflection;

   public class Tester
   {
      public static void Main( )
      {
         // examine a single object
          Type theType =
            Type.GetType(
               "System.Reflection.Assembly");
          Console.WriteLine(
            "\nSingle Type is {0}\n", theType);
      }
   }
}
```

Output:
```
Single Type is System.Reflection.Assembly
```

2.7 MARSHALLING

Marshaling is the process of creating a bridge between managed code and unmanaged code; it is the homer that carries messages from the managed to the unmanaged environment and reverse. It is one of the core services offered by the CLR (Common Language Runtime.).NET code are called "managed" because it is controlled (managed) by the CLR. Other code that is not controlled by the CLR is called unmanaged.

Marshalling Types During Platform Invoke (P/Invoke) on the Microsoft .NET Compact Framework.

Data type representations in the Microsoft® .NET Compact Framework differ from those in unmanaged code. Converting between the managed and unmanaged representations is called marshaling and is automatic for most simple data types.

Marshaling is the act of taking data from the environment you are in and exporting it to another environment. In the context of .NET, marhsaling refers to moving data outside of the app-domain you are in, somewhere else.

Marshalling Value and Reference Types :
Value Types : STACK
Reference Types : HEAP

Value types are marshaled to unmanaged code on the stack. Reference types are passed by address. This means a pointer is passed on the stack, and the pointer contains the address of the marshaled data on the heap.

BLITTABLE : A type is considered blittable if it has a common representation in managed and unmanaged code memory.

NON-BLITTABLE : Non-blittable types require custom marshaling to convert between the unmanaged and managed representations.

Value Types
The data types outlined in the table below are automatically marshaled by value using P/Invoke. The table also shows the unmanaged equivalents in C/C++.

Common Value Types that are Automatically Marshaled

C#	**Visual Basic .NET**	**Native C/C++**	**Size (bits)**
int	Integer	int	32
short	Short	short	16
bool	Boolean	BYTE	8
char	Char	WCHAR	16

Take the following C/C++ function as an example. The function accepts three integer parameters and calculates their arithmetic mean:

```
extern "C" _declspec(dllexport) int mean(int x, int y, int z)
{
   return (x + y + z) / 3;
}
```

This method can be declared and called in a Smart Device application. The following code calculates the mean of 1, 3 and 5 (that is, 3) and displays it in a Message Box:

```
using System.Runtime.InteropServices;
.
.
.
[DllImport("MarshalByValueDemo.dll")]
extern static int mean(int x, int y, int z);
.
.
.
int avg = mean(1, 3, 5);
MessageBox.Show(String.Format("The mean is {0}", avg));
```

It is worth noting; only value types of 32 bits or less can be marshaled automatically. Long types (64-bit integer) and floating-point types (float and double) cannot be marshaled by value into unmanaged code. You should pass these values by reference.

Reference Types
In the .NET Compact Framework, reference types are, by default, passed by reference.When parameters are passed by reference, a pointer to the parameters on the managed heap is passed to the unmanaged code.
Since the unmanaged code receives a pointer, it is possible for the method to modify the data held on the managed heap. The .NET Compact Framework also supports passing value types by reference, using the **ref** keyword in **C#** and **ByRef in Visual Basic .NET**.

The following example takes three double parameters, passed by reference, and returns the arithmetic mean through a fourth parameter, also passed by reference.

```
extern "C" _declspec(dllexport) void mean(double* x, double* y, double* z,
 double* mean)
{
   *mean = (*x + *y + *z) / 3.0;
}
```

To call this from managed code:

```
[DllImport("MarshalByRefDemo.dll")]
extern static void mean(ref double x, ref double y, ref double z, ref
  double mean);
.
.
.
double x = 1.0;
double y = 3.0;
double z = 5.0;
double avg = 0.0;
mean(ref x, ref y, ref z, ref avg);
MessageBox.Show(String.Format("The mean is {0}", avg));
```

MARSHALLING ARRAYS

In C/C++, arrays are representations as pointers to a contiguous region of memory with the array elements addressed as offsets from this pointer, starting with zero.

The marshaler in the .NET Compact Framework Common Language Runtime (CLR) ensures that managed arrays adhere to this format when passed to unmanaged code.

In this C/C++ example, an array is passed into a function and is searched to locate the smallest value:

```
extern "C" _declspec(dllexport) int MinArray(int* pData, int length)
{
   // Initialise minData to the first element of the pData Array
   int minData = pData[0];
   int pos;

   // Loop through the array
   for(pos = 1; pos < length; pos++)
   {
      // If the current element is less than minData,
      // set minData to the value of current element
      if(pData[pos] < minData)
         minData = pData[pos];
   }

   return minData;
```

}

The following code declares the above method and calls it from C#:

```
[DllImport("MarshalArray.dll")]
extern static int MinArray(int[] pData, int length);
.
.
.
int[] sampleData = int[] {5, 1, 3 };
int result = MinArray(sampleData, sampleData.Length);
MessageBox.Show(String.Format("Smallest integer is {0}, result));
```

2.8 NET REMOTING OVERVIEW

.NET Remoting is a Microsoft application programming interface (API) for interprocess communication

.NET remoting enables you to build widely distributed applications easily, whether application components are all on one computer or spread out across the entire world. You can build client applications that use objects in other processes on the same computer or on any other computer that is reachable over its network. You can also use .NET remoting to communicate with other application domains in the same process.

To use .NET remoting to build an application in which two components communicate directly across an application domain boundary, you need to build only the following:

- A remotable object.
- A host application domain to listen for requests for that object.
- A client application domain that makes requests for that object.

Remoting is a framework built into Common Language Runtime (CLR) in order to provide developers classes to build distributed applications and wide range of network services.

Remoting provides various features : Object Passing, Proxy Objects,Activation, Stateless and Stateful Object, Lease Based LifeTime and Hosting of Objects in IIS.

The namespaces that one typically uses in C# distributed object applications are the following:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
```

When using one of the major IDE's (Visual Studio or C# Builder), it is important to add the reference system.remoting.dll to the build if it is not already present.

In C#, using distributed objects does not require stubs or interfaces as in Java. The CLR provides full support for remote object calls. Using distributed objects does not depend on the system registry for information about the remote classes. This information is encapsulated in a .DLL file that must be added as a reference when compiling the client code.

Classes derived from System.MarshalByRefObject cause the distributed object system to generate proxy objects on the client that encapsulate the low-level socket protocol. When the client sends a message to a remote object, it is the proxy that processes this message and sends serialized information across the network. The same works in reverse when proxy objects de-serialize information that is returned from the server.

Channel objects are the mechanism used to transfer messages between client and server. The .NET framework provides two bidirectional channels:

System.Runtime.Remoting.Channels.http.HttpChannel and
System.Runtime.Remoting.Channels.Tcp.TcpChannel.

The http channel uses SOAP (Simple Object Access Protocol) and the tcp channel uses a binary stream. This latter method is more efficient because it avoids the need to encode and decode SOAP messages.

A channel must be registered before it can be used. The ChannelServices class is used to accomplish this as follows:

```
ChannelServices.RegisterChannel(someChannel);
```

The general steps involved in writing a distributed application are summarized below :

Writing the Server

1. Construct the server class.
2. Select a method for hosting the server object(s) on the server. Typically a short application is created that launches the server and makes the server object available to the client(s).
3. The server object typically waits for one or more client objects to communicate with it.

Writing the Client

1. Identify the remote server object to the client.
2. Connect the server to the client through a channel.
3. The client must activate the remote object and create a reference to it.
4. Communication to the remote object(s), once activated, is similar to sending messages to local objects.

Let us consider a simple client server application.

Remoting Object
This is the object to be remotely access bynetwork applications. The object to be accessed remotely must bederived by MarshalByRefObject and all the objects passed by value mustbe serializable.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace RemotingSamples
{
public class RemoteObject : MarshalByRefObject
{

///constructor
public RemoteObject()
{       Console.writeline("Remote object activated");      }

///return message reply
public String ReplyMessage(String msg)
                              {
          Console.WriteLine("Client : "+msg);//print given message on console
          return "Server : Yeah! I'm here";       }
}
}
```

2.8.1 Server
This is the server application used toregister remote object to be access by client application. First, of all choose channel to use and register it, supported channels are HTTP,TCP and SMTP. I have used here TCP. Then register the remote object specifying its type.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
```

```
namespace RemotingSamples
{
public class Server
{
 ///constructor
public Server()
        {               }
 ///main method
public static int Main(string [] args)
        {
        //select channel to communicate
                TcpChannel chan = new TcpChannel(8085);
        //register channel
                ChannelServices.RegisterChannel(chan);
        //register remote object
                RemotingConfiguration.RegisterWellKnownServiceType(
                                Type.GetType("RemotingSamples.RemoteObject,object"),
                                        "RemotingServer",
                                        WellKnownObjectMode.SingleCall);
        //inform console
                Console.WriteLine("Server Activated");
        return 0;
        }
}       }
```

2.8.2 Client

This is the client application and it willcall remote object method. First, of all client must select the channelon which the remote object is available, activate the remote object andthan call proxy’s object method return by remote object activation.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using RemotingSamples;

namespace RemotingSamples
{
public class Client
{
        ///constructor
        public Client()
                {       }
///main method
public static int Main(string [] args)
        {
        //select channel to communicate with server
                TcpChannel chan = new TcpChannel();
                ChannelServices.RegisterChannel(chan);
                RemoteObject remObject = (RemoteObject)Activator.GetObject(
                                        typeof(RemotingSamples.RemoteObject),
                                        "tcp://localhost:8085/RemotingServer");
        if (remObject==null)
        Console.WriteLine("cannot locate server");
        else
        remObject.ReplyMessage("You there?");
return 0;
        }
}  }
```

Deployment :

To deploy this distributed application, the following sequence of steps must be followed:

1.The remote object must be compiled as follows to generate remote object.dll which is used to generate server and client executable.

```
        csc /t:library /debug /r:System.Runtime.Remoting.dll remoteobject.cs
```

2.The server must be compiled as follows to produce server.exe.

```
        csc /debug /r:remoteobject.dll /r:System.Runtime.Remoting.dll server.cs
```

3. The client must be compiled as follows in order to produce client.exe

```
        csc /debug /r:remoteobject.dll /r:System.Runtime.Remoting.dll client.cs
```

2.9 EXCEPTION HANDLING :

An exception handling is an error that occurs at runtime. Using c# exception handling, you can handle runtime errors in a structured and controlled manner. Exception handling streamlines error-handling by allowing your program to define a block of code, called an exception handler, that is executed automatically when an error occurs. It is not necessary to check the success or failure of each specific operation or method call manually. If an error occurs, it will be processed by the exception handler.

The System.Exception Class:

In c#, exception are represented by classes. All exception classes must be derived from the built-in class Exception,which is part of System namespace. Thus all exception are subclasses of Exception.
From Exception are derived **SystemException** and **ApplicationException**. These support the two general catagories of exception:

- Those generated by the c# runtime system
- Those generated by application program

C# defines built-in exception that are derived from SystemException. For example, when division-by-zero is attempted, a **DivideByZero** Exception is generated.

Exception Handling Fundamentals:

C# exception handling is managed via four keywords: **try, catch, throw** and **finally.**
Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch exception using **catch** and handle it in some rational manner. To manually throw an exception, use the keyword **throw**. Any code that absolutely must be executed upon exiting from a **try** block is put in a **finally** block.

Using try and catch:

At the core of exception handling are **try** and **catch**. These keywords work together. You can't have a try without a **catch**, or a catch without a **try.** The general form is:

```
try{
//block of code to monitor for errors
}
catch(ExcepType1 exOb) {
//handler for ExcepType1
}
catch(ExcepType2 exOb) {
//handler for ExcepType2
}
```

Here when the exception is thrown, it is caught by its corresponding **catch** statement, which then processes the exception. As the general form shows, there can be more than one **catch** statement associated with a **try.** The type of exception determines which catch statement is executed. If no exception is thrown, then a try block ends normally and all the catch statements are bypassed. The catch statements are executed only if an exception is thrown.

Exception Example :

Following is a simple example that illustrates how to watch for and catch an exception:

```
using System;
class Except
{
public static void Main()
{
int x=Int 32 Parse(Console.ReadLine());
int y=Int 32 Parse(Console.ReadLine());
int[] a ={10,5,3,4};
try
{
int z=x\y;
Console.WriteLine(z);
int b=a[3]+a[4];
Console.WriteLine(b);
}
}
catch(Exception e)
{
Console.WriteLine("error");
}
}
```

Output:

```
x=15
y=3
z=5
```

Using Multiple Catch Statements:

You can associate more than one **catch** statement with a **try.** However each **catch** statement must catch a different type of exception. The general form of **multiple catch** statement is:

```
catch(arithmetic Exception e1)
{
}
catch(ArrayIndexOutOfBoundException e2)
{
}
catch(Exception e3)
{
}
```

Following is a simple example that illustrates how to use multiple catch statements:

```
using System;
class Except
{
public static void Main()
{
int x=Int 32 Parse(Console.ReadLine());
int y=Int 32 Parse(Console.ReadLine());
int[] a ={10,15,20,3};
try
{
int z=x\y;
Console.WriteLine(z);
int b=a[3]+a[4];
Console.WriteLine(b);
}
catch(arithmetic Exception e1)
{
Console.WriteLine("Arithmetic Exception);
}
catch(ArrayIndexOutOfBoundException e2)
{
Console.WriteLine("ArrayOutOfBoundException");
}
}
}
```

Output:

```
x=10
y=2
z=5
ArrayOutOfBoundException
```

Throwing An Exception:

It is possible to manually throw an exception by using the throw statement.its general form is shown here:

Throw expectOb;

The expectOb must be an object of an exception class derived from **Exception.**

Following is a simple example that illustrates the **throw** statement by manually throwing a **DivideByZeroException:**

```
using System;
class Throwdemo
{
public static void Main()
{
```

```
try
{
Console.WriteLine("Before throw");
throw new DivideByZeroException();
}
catch (DivideByZeroException)
{
Console.WriteLine("Exception caught");
}
Console.WriteLine("After try/catch block");
}
}
}
```

Output:

```
Before throw
Exception caught
After try/catch block
```

Rethrowing an Exception:

An exception caught by one **catch** statement can be rethrown so that it can be caught by an outer **catch**. The most likely reason to rethrow an exception is to allow multiple handlers access to the exception. To rethrow an exception, you simply specify **throw,** without specifying an exception. That is, you use the form of **throw:**

```
throw;
```

If you rethrow an exception,it will not be recaught by the same **catch** statement. It will propagate to the next **catch** statement.

Following is a simple example that illustrates the rethrowing of an exception:

```
using System;
class rethrow
{
Int x=0,y=0;
{
public void div()
{
try
{
int z=x/y;
throw new DivideByZeroException();
}
catch
{
throw;  //rethrow the exception
}
}
}
class excep
{
public static void Main()
{
rethrow r=new rethrow();
try
{
r.div();
}
catch(DivideByZeroException e)
{
Console.WriteLine("Exception received");
}
}
}
```

Output:

```
Exception received
```

Using finally:

Sometimes you will want to define a block of code that will execute when a **try/catch** block is left. Such types of circumtances are common in programming, and c# provides a convenient way to handle them using **finally** keyword. To specify a block of code execute when a **try/catch** block is exited, include a **finally** block at the end of a **try/catch** sequence. The general form of a **try/catch** that includes **finally** is shown here:

```
try
{
catch(ExcepType1 exOb)
{
}
Catch(ExcepType2 exOb)
{
}
.
.
.
finally
{
}
```

The **finally** block will be exceuted whenever exceution leaves a **try/catch** block, no matter what conditions cause it. That is, whether the **try** block ends normally or because of an exception, the last code executed is that defined by **finally**. The finally block is also executed if any code within the **try** block or any of its **catch** statements returns from the method.
Following is a simple example that illustrates the use of finally keyword:

```
using System;
class finally
{
public static void Main()
{
int x=10,y=0;
int[]a={10,15};
try
{
int z=x/y;
int b=a[1]+a[2];
}
catch(Arithmetic Exception e1)
{
Console.WrieLine("Divide by Zero");
}
finally
{
Console.WrieLine("Error");
}
}
}
```

<u>Output :</u> Divide by Zero Error

Commonly Used Exception:

Exception	**Meaning**
ArrayTypeMismatchExecption	Type of value being stored is incompatiable with the type of the array.
DivideByZeroException	Division by zero attempted
IndexOutOfRangeException	Array index is out of bounds
InvalidCastExecption	A runtime cast is invalid
OutOfMemoryException	A call to new fails because insufficient free memory exists
OverflowException	An arithmetic overflow occurred
NullReferenceException	An attempt was made to operate on a null reference that is,a reference that does not refer to an object
StackOverflowException	The stack was overrun

2.10 Garbage Collection

Object lifetime in C#

 Memory allocation for an object should be made using the “new” keyword
 Objects are allocated onto the managed heap, where they are automatically deallocated by the runtime at “some time in the future”
 Garbage collection is automated in C#

Note : Allocate an object onto the managed heap using the new keyword and forget about it

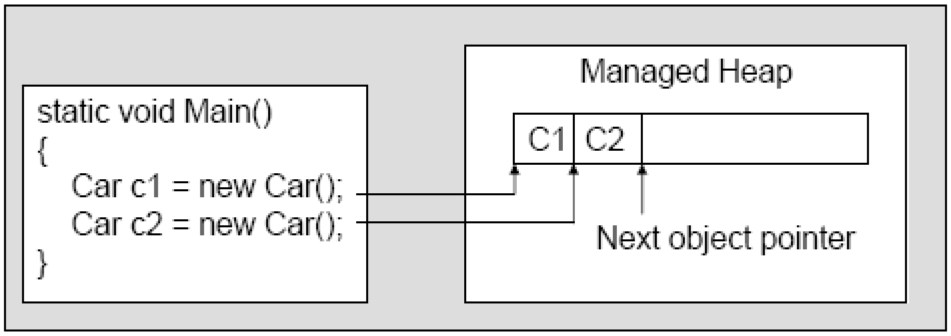
Object creation

 When a call to new is made, it creates a CIL “newobj” instruction to the code module

```
public static int Main (string[] args)
{
        Car c = new Car(“Viper”, 200, 100); }
```

Tasks taken by CIL newobj instruction

 Calculate the total amount of memory required for the object.
 Examine the managed heap to ensure enough room for the object.
 Return the reference to the caller, advance the next object pointer to point to the next available slot on the managed heap.



Rule: If the managed heap does not have sufficient memory to allocate a requested object, a garbage collection will occur.

Garbage collection steps

1. The garbage collector searches for managed objects that are referenced in managed Code - mark
2. The garbage collector attempts to finalize objects that are unreachable - Sweep
3. The garbage collector frees objects that are unmarked and reclaims their memory - Sweep

Building finalizable objects

```
//System.Object
   public class Object
   {
   ...
       protected virtual void Finalize() { }
   }
```

- Override Finalize() to perform any necessary memory cleanup for your type
- A call to Finalize () occurs:
  - natural garbage collection
  - GC.Collect()
  - Application domain is unloaded from the memory

The System.GC type

- Provide a set of static method for interacting with garbage collection
- Use this type when you are creating types that make use of unmanaged resource.

When to override System.Object.Finalize()
The only reason to override Finalize() is if your C# class is making use of unmanaged resources via PInvoke or complex COM interoperability tasks (typically via the System.Runtime,InteropServices.Marshal type). It is illegal to override Finalize() on structure types.

Building Disposable Objects

- ✓ Another approach to handle an object's cleanup.
- ✓ Implement the IDisposable interface
- ✓ Object users should manually call Dispose() before allowing the object reference to drop out of scope
- ✓ Structures and classes can both support Idisposal (unlike overriding Finalize())

.