Programming in C#
Unit-4

21-12,2020, 23-12-2020, 26-12-2020,
28-12-2020, 04-01-2021,06-1-2021

Dr.M.Paul Arokiadass Jerald
Assistant Professor
Department of Computer Science
Periyar Arts College, Cuddalore-1

Syllabus Overview

UNIT - IV: REFLECTION AND REMOTING

- Life Cycle of threads
- Using Reflection – Reflecting the Members of a Class - Dynamic Loading and Reflection
- .NET Remoting – Architecture – Hosting of Objects – Single Ton and Single Call – Remoting Server – Remoting Client.

4.1 Multi-Threading

- **Threading** means parallel code execution.
- **Multi-threading** is the most useful feature of C# which allows **concurrent programming(execution)** of two or more parts of the program for maximizing the utilization of the CPU.
- Each part of a program is called Thread.

- **Namespace** : System.Threading
- **Assembly** : System.Threading.Thread.dll

Types of Thread

C# supports two types of threads :

1. Foreground Thread

- thread which keeps on running to complete its work even if the *Main* thread leaves its process.
- Foreground thread does not care whether the main thread is alive or not, it completes only when it finishes its assigned work.
- Life of the foreground thread does not depend upon the main thread.

Types of Thread

2. Background Thread

- A thread which leaves its process when the Main method leaves its process.
- Life of the background thread depends upon the life of the main thread.
- If the main thread finishes its process, then background thread also ends its process.
- **Note:** To use a background thread in your program, then set the value of *IsBackground* property of the thread to *true*.

Life Cycle of a Thread

- A thread in C# at any point of time exists in any one of the following states :
  - **Unstarted**
  - **Runnable**
  - **Running**
  - **Not Runnable**
  - **Dead**

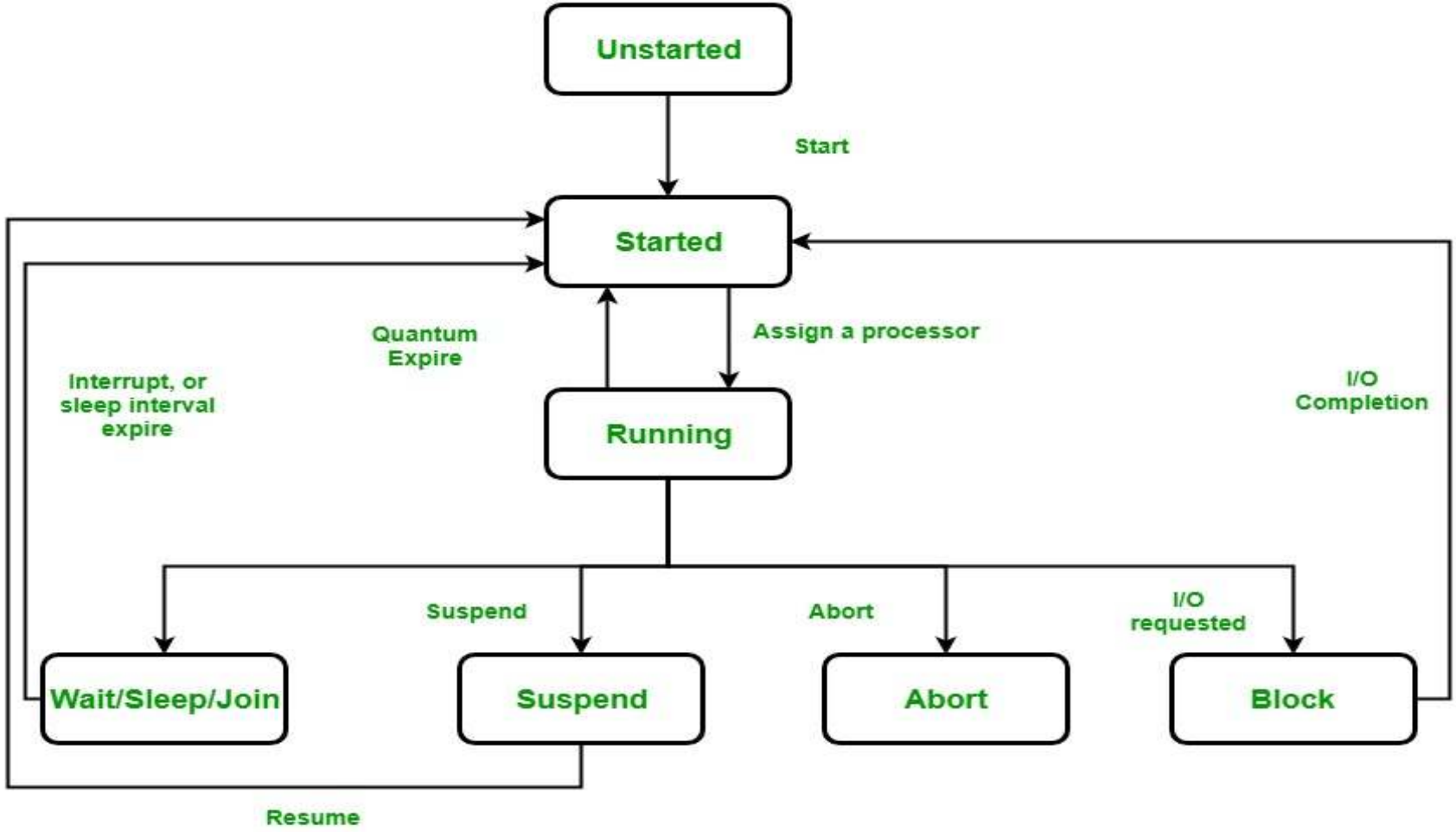
Life Cycle of a Thread

- A thread in C# at any point of time exists in any one of the following states.
- A thread lies only in one of the shown states :
  - **Unstarted**
  - **Runnable**
  - **Running**
  - **Not Runnable**
  - **Dead**

Life Cycle of a Thread

- A thread in C# at any point of time exists in any one of the following states.
- A thread lies only in one of the shown states :
  - **Unstarted**
  - **Runnable**
  - **Running**
  - **Not Runnable**
  - **Dead**

Flowchart of a Thread Life cycle



Life Cycle of a Thread …

- **Unstarted state:** When an instance of a Thread class is created, it is in the unstarted state, means the thread has not yet started to run when the thread is in this state.
- Or in other words Start() method is not called.

  Thread thr = new Thread();

  Here, thr is at unstarted state.
- **Runnable State:** A thread that is ready to run is moved to runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time.
- It is the responsibility of the thread scheduler to give the thread, time to run. the Start() method is called.

Life Cycle of a Thread …

- **Running State:** A thread that is running. Or in other words, the thread gets the processor.
- **Not Runnable State:** A thread that is not executable because
  - Sleep() method is called.
  - Wait() method is called.
  - Due to I/O request.
  - Suspend() method is called.
- **Dead State:** When the thread completes its task, then thread enters into dead, terminates, abort state.

Methods to implement Thread

- Thread class provides different types of methods to implement the states of the threads :
- **Sleep()** method is used to temporarily suspend the current execution of the thread for specified milliseconds, so that other threads can get the chance to start the execution
- **Join()** method is used to make all the calling thread to wait until the main thread, i.e. joined thread complete its work.
- **Abort()** method is used to abort the thread.

Methods to implement Thread

- **Suspend()** method is called to suspend the thread.
- **Resume()** method is called to resume the suspended thread.
- **Start()** method is used to send a thread into runnable State.

Example – Life Cycle

```
// C# program for states of thread
using System;
using System.Threading;
public class MyThread
{
      public void thread()
     {
      for (int x = 0; x < 2; x++)
     {
      Console.WriteLine("My Thread");
      }
      }
}
public class ThreadExample {
      // Main method
      public static void Main()
      {       // Creating instance for
              // mythread() method
              MyThread obj = new MyThread();
// Creating and initializing
// threads Unstarted state
Thread thr1 = new Thread(new
      ThreadStart(obj.thread));
Console.WriteLine("ThreadState: {0}",
              thr1.ThreadState);
// Running state
thr1.Start();
Console.WriteLine("ThreadState: {0}",
      thr1.ThreadState);
// thr1 is in suspended state
thr1.Suspend();
Console.WriteLine("ThreadState: {0}",
      thr1.ThreadState);
// thr1 is resume to running state
              thr1.Resume();
              Console.WriteLine("ThreadState: {0}",
      thr1.ThreadState);
      }
}
```

4.2 Reflection

- **Reflection** objects are used for obtaining metadata (information) at runtime.
- The classes that give access to the metadata of a running program are in **System.Reflection** namespace.
- The **System.Reflection** namespace contains classes that allow you to obtain information about the application and to dynamically add types, values, and objects to the application.
- Reflection in C# is similar to RTTI (Runtime Type Information) of C++.

Applications of Reflection

- Reflection has the following applications −
  - It allows view attribute information at runtime.
  - It allows examining various types in an assembly and instantiate these types.
  - It allows late binding to methods and properties
  - It allows creating new types at runtime and then performs some tasks using those types.

Types defined in Metadata

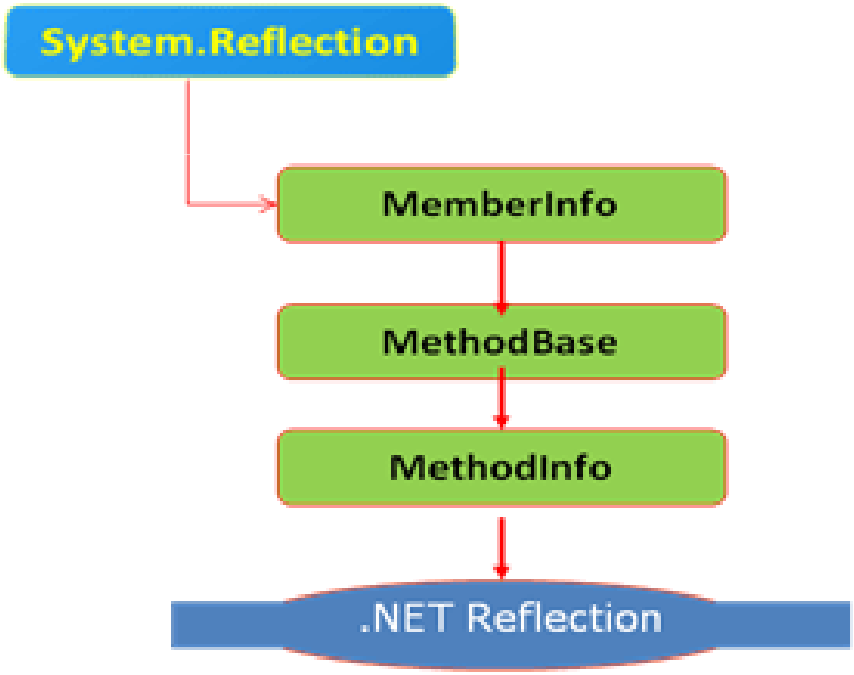
- classes (class);
- interfaces (interface);
- structures (struct);
- enumerations (enum);
- delegates (delegate).

Applications of Reflection

- Reflection has the following applications −
  - It allows view attribute information at runtime.
  - It allows examining various types in an assembly and instantiate these types.
  - It allows late binding to methods and properties
  - It allows creating new types at runtime and then performs some tasks using those types.

System.Reflection Namespace

System.Reflection Namespace hierarchy



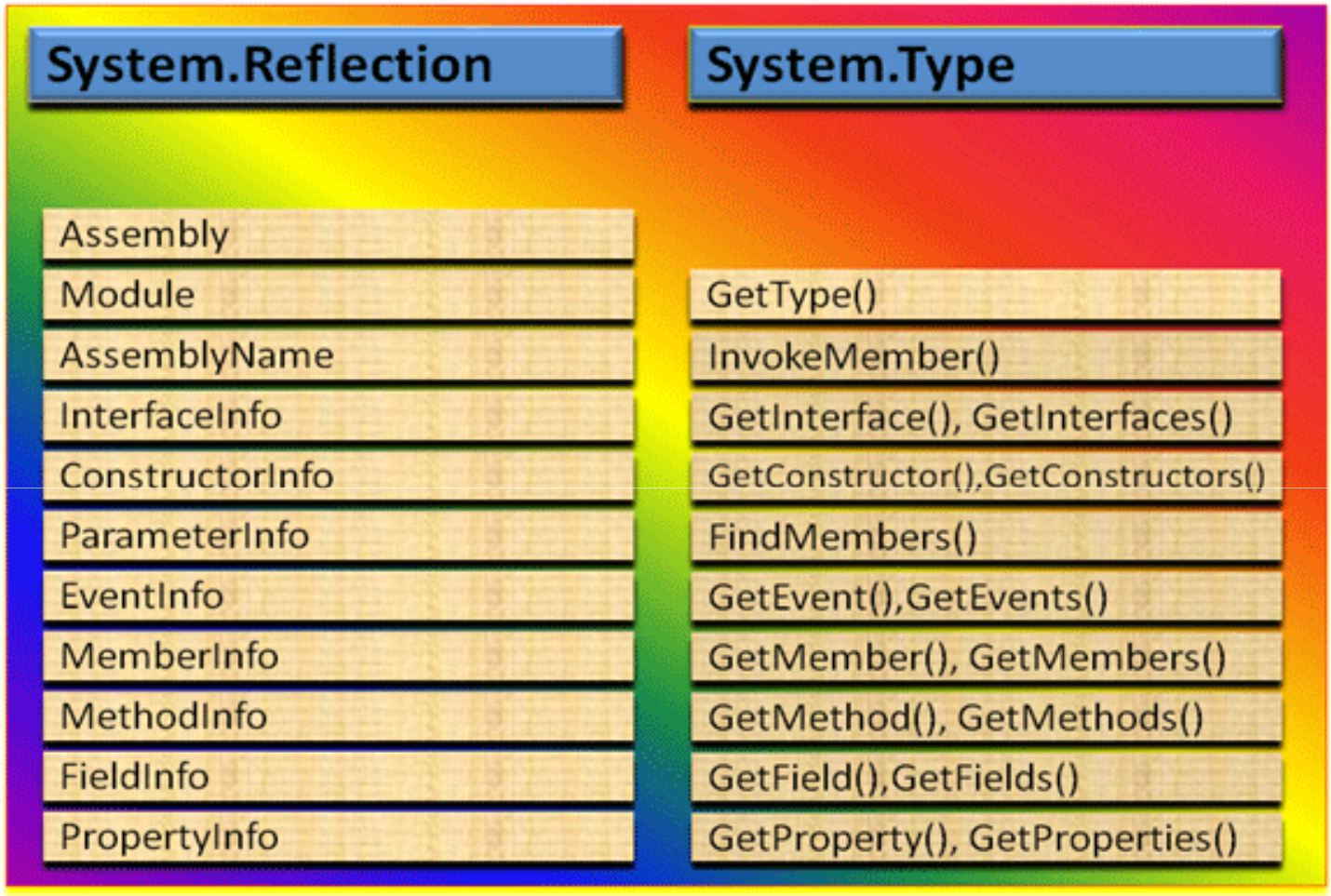
Types in System.Reflection namespace

- **Assembly** – abstract class. It contains static methods for working with the assembly.
- **AssemblyName** – this is a class that contains information that is used to identify the assembly. For example: the version number of the assembly, information about the culture, etc.;
- **EventInfo** – abstract class. Contains information about a specified event;
- **FieldInfo** – abstract class. It can contain information about the specified data members of the class;

Types in System.Reflection namespace

- **MemberInfo** – abstract class. Contains general behavior information for classes
(types) EventInfo, FieldInfo, MethodInfo and PropertyInfo;
- **MethodInfo** – abstract class. Contains information about specified method;
- **Module** – abstract class. It allows to get information about a given module in the case of a multi-file assembly;
- **ParameterInfo** – a class that contains information about a given parameter in a given method;
- **PropertyInfo** – abstract class. It contains information about the specified property.

.NET Reflection Road Map



https://www.c-sharpcorner.com/UploadFile/keesari_anjaiah/reflection-in-net/

Example for Reflection

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
 // connect the System.Reflection namespace
using System.Reflection;

namespace TrainReflection1
{    class MathFunctions
    {
        public int a, b, c; // data members of class

        // minimum between two values
        public int Min2(int a, int b)
        {           if (a < b) return a;
            return b;
        }
```

```
// minimum between three values
        public int Min3(int a, int b, int c)
        {
            int min = a;
            if (min > b) min = b;
            if (min > c) min = c;
            return min;
        }
         // maximum between two values
        public int Max2(int a, int b)
        {
            if (a < b) return b;
            return a;
        }
    }
```

```
class Program
   {   static void Main(string[] args)
       {
           // get the value of the type
           Type tp = null;
            tp = Type.GetType("TrainReflection1.MathFunctions");
            // get a list of methods from the MathFunctions
         class
           MethodInfo[] mi = tp.GetMethods();
            // get the names of methods
           string m1 = mi[0].Name; // m1 = "Min2"
           string m2 = mi[1].Name; // m2 = "Min3"
           string m3 = mi[2].Name; // m3 = "Max2"
            // get a list of internal class data
           FieldInfo[] fi = tp.GetFields();
            string f1 = fi[0].Name; // f1 = "a"
           string f2 = fi[1].Name; // f2 = "b"
           string f3 = fi[2].Name; // f3 = "c"
```

```
Console.WriteLine("Method1 = {0}", m1); // Method1 = Min2
Console.WriteLine("Method2 = {0}", m2); // Method2 = Min3
Console.WriteLine("Method3 = {0}", m3); // Method3 = Max2
  Console.WriteLine("Field1 = {0}", f1); // Field1 = a
  Console.WriteLine("Field2 = {0}", f2); // Field2 = b
  Console.WriteLine("Field3 = {0}", f3); // Field3 = c
        Console.WriteLine("\nDetails about Object");
        Type t = typeof(string);
        Console.WriteLine("Name : {0}", t.Name);
        Console.WriteLine("Full Name : {0}", t.FullName);
        Console.WriteLine("Namespace : {0}", t.Namespace);
        Console.WriteLine("Base Type : {0}", t.BaseType);
        Console.ReadKey();}
    }
}
```

Reflection of Methods

- Reflection of methods allows to obtain information about the list of public methods of a given type. Information about methods can be obtained for a class, structure, or interface.
- To get the list of methods of a given type instance, GetMethods() method is used.
- GetMethods() returns an array of type MethodInfo, which contains all the necessary information about the methods.

Reflection on fields of a class, structure or enumeration

- To get information about a field (property) of a particular type (class, structure, enumeration), use the Type.GetFields() method.
- **Type.GetFields()** method returns an array of type FieldInfo.
- The FieldInfo type contains all the necessary information about the fields and properties that are declared as public.

Example – Reflection of Methods

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;
namespace ConsoleApp2
{   // class, which defines a date
  public class Date
  {  int number;
    int month;
    int year;
    // access methods
    public int GetNumber() { return number; }
    public int GetMonth() { return month; }
    public int GetYear() { return year; }
    public void SetNumber(int nnumber) { number = nnumber; }
    public void SetMonth(int nmonth) { month = nmonth; }
    public void SetYear(int nyear) { year = nyear; }
  }
```

Example – Reflection of Methods

```
class Program
{  static void Main(string[] args)
   {
      // get information about methods
      // get an instance of the type by its name
      Type tp = Type.GetType("ConsoleApp2.Date");
// class name "Date" in the assembly ConsoleApp2
      // get an array of class Date methods
      MethodInfo[] methods = tp.GetMethods();
      // display the method names
      int i = 0;
      foreach (MethodInfo mi in methods)
      {   i++;
          Console.WriteLine("Method[{0}] = {1}", i, mi.Name);
      }
    }   }
}
```

4.3 DYNAMIC LOADING

- The act of loading external assemblies on demand is known as Dynamic Loading.
- Using the Assembly class, we can dynamically load both private and shared assemblies from the local location to a remote location as well as, explore its properties.

DYNAMIC LOADING INTRO 1/2

- An external assembly can be **connected to the program in two ways**:
- in static way by using special tools from Microsoft Visual Studio. This is done, for example, with the commands “Add Reference …” or “Add Service Reference …”. In this case, the assembly manifest contains the relevant information about the external assembly that was connected; be obtained dynamically.

DYNAMIC LOADING INTRO 1/2

- An external assembly can be **connected to the program in two ways**:
- in static way by using special tools from Microsoft Visual Studio. This is done, for example, with the commands “Add Reference …” or “Add Service Reference …”. In this case, the assembly manifest contains the relevant information about the external assembly that was connected; be obtained dynamically.

DYNAMIC LOADING INTRO 2/2

- dynamically using the Assembly class which is located in the **System.Reflection** namespace.
- In this case, the information about the external assembly is not placed in the manifest of the current assembly.
- This information is obtained during runtime, that is, dynamically. There are a number of tasks in which information about an assembly in a program should be obtained dynamically.

Dynamic loading of assembly

- A program that is hosted in an assembly can include other assemblies in order to use their capabilities (classes, interfaces, methods, etc.).
- *Dynamic loading of assembly* is the process of loading and retrieving information about external assemblies on demand during program execution.
- When dynamically loading an external assembly, there is no information in the manifest about this assembly. The information is obtained programmatically.

Load() and LoadFrom() Methods

- The System.Assembly class contains tools for dynamically loading assemblies and viewing their properties.
- The System.Assembly class is located in the System.Reflection namespace.
- To load an assembly, invoke one of the methods: method Load() for private assemblies.

Load() and LoadFrom() Methods

- Private assemblies are located in the same directory as the program that uses them (explores); method LoadFrom() for shared assemblies.
- Shared assemblies are libraries that can be used by different applications on the same machine. Shared assemblies are deployed in a special GAC (Global Assembly Cache) directory.
- **LoadFrom() Example:**

```
Assembly asm = Assembly.LoadFrom(@"E:\TestLib.dll");
```

- **Load() Example:**

```
Assembly asm = Assembly.Load(typName);
```

Example – Dynamic Loading

```
using System;
using System.Reflection;
namespace Reflection
{   class Program
    {  static void Main(string[] args)
       {
          Console.Write("Enter External Assembly:");
          string typName = Console.ReadLine();
          try
          {   Assembly asm = Assembly.Load(typName);

             DisplayAssembly(asm);  }
          catch
          {    Console.WriteLine("Can't Load Assembly");
          }
          Console.ReadKey();
       }
```

Example – Dynamic Loading….

```
        static void DisplayAssembly(Assembly a)
        {
            Console.WriteLine("*****Contents in Assembly*******");
            Console.WriteLine("Information:{0}",a.FullName);
            Type[] asm = a.GetTypes();
            foreach (Type tp in asm)
            {
                Console.WriteLine("Type:{0}", tp);
            }
        }
    }
}
```

4.4 Late Binding

- The .NET framework can create an instance of a given type using early binding or late binding.
- In early binding, we typically set the external assembly reference in the project and allocate the type using the new operator. Early binding allows us to determine errors at compile time rather than at runtime.
- In late binding, an instance of a given type and invoke its methods at runtime can be created without having knowledge at compile time.
- A late binding instance of an external assembly can be created using the CreateInstance() method of the System.Activator static class.

Example – Late Binding

```
using System;
using System.Reflection;
 namespace Reflection
{   class Program
    {   static void Main(string[] args)
        {   try
            {  Console.WriteLine("****Assembly Late Binding*****");
               Type t = Type.GetType("TestLib.utility,TestLib");
               object obj = Activator.CreateInstance(t);
               Console.WriteLine("Create a {0} using late binding", obj);
               MethodInfo mth = t.GetMethod("Test");
               mth.Invoke(obj, null);
               Console.WriteLine("Method Invoked");
            }
            catch
            {   Console.WriteLine("Can't Create Assembly Instance");
            }
            Console.ReadKey();
        }  }
}
```

.NET REMOTING

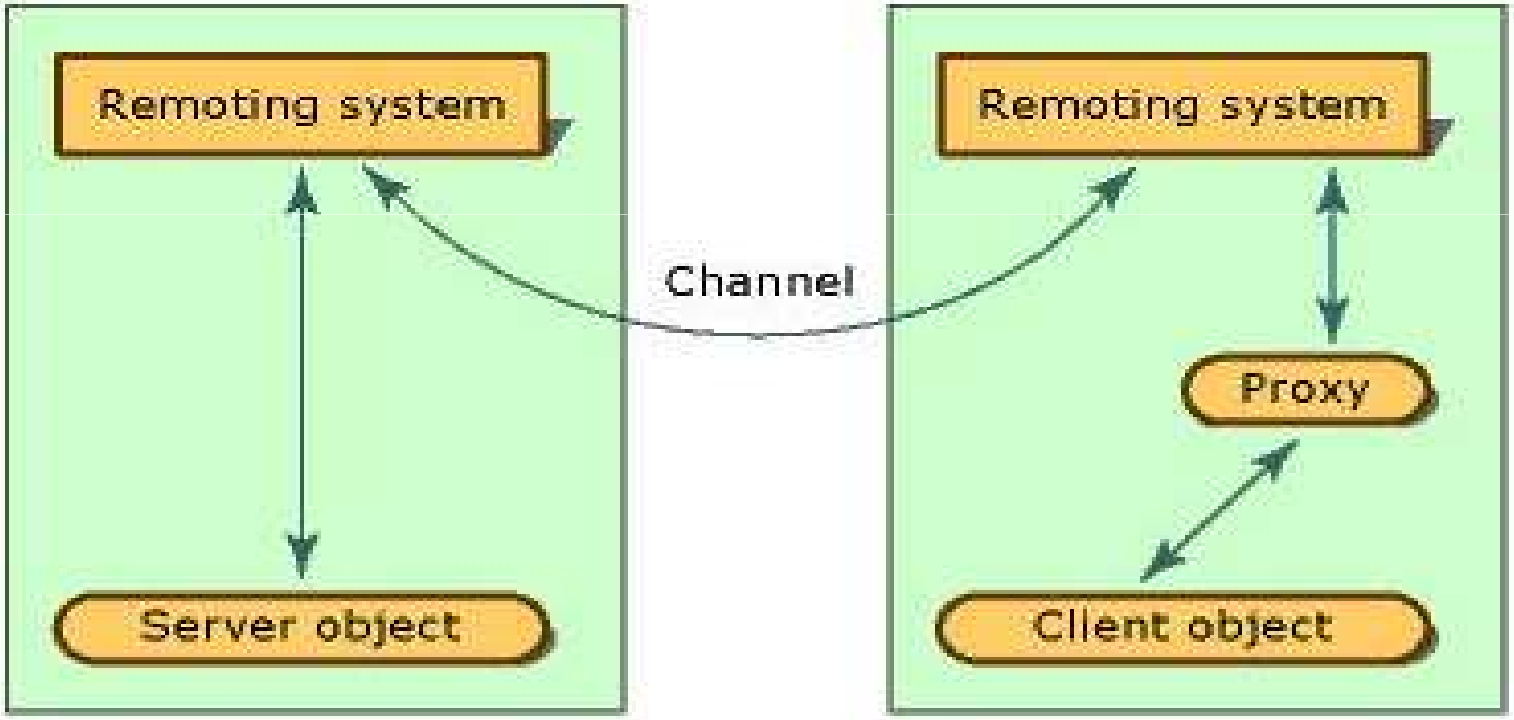
4.5 .NET REMOTING…Intro

- .NET Remoting provides an infrastructure for distributed objects.
- It exposes the full object semantics of .NET to remote processes using plumbing that is both flexible and extensible.
- .NET Remoting offers complex functionality, including support for passing objects by value or by reference, callbacks, and multiple-object activation and lifecycle management policies.
- In order to use .NET Remoting, a client needs to be built using .NET

.NET REMOTING…INTRO

- The **.NET Remoting** provides an inter-process communication between Application Domains by using Remoting Framework.
- The applications can be located on the same computer , different computers on the same network, or on computers across separate networks.
- The .NET Remoting supports distributed object communications over the TCP and HTTP channels by using Binary or SOAP formatters of the data stream.

.NET REMOTING ARCHITECTURE



.NET FORMATTERS FOR REMOTING

- .NET Framework provides two standard formatters
- System.Runtime.Serialization.Formatters.Binary.BinaryFormatter
- System.Runtime.Serialization.Formatters.Soap.SoapFormatter
- The BinaryFormatter and SoapFormatter as the name suggest marshal types in binary and SOAP format respectively.

.NET FORMATTERS FOR REMOTING

- For metadata .NET Remoting relies on the CLR assemblies, which contain all the relevant information about the data types they implement and expose it via reflection. The reliance on the assemblies for metadata makes it easy to preserve the full runtime type-system reliability. As a result, when the .NET Remoting marshals data, it includes all of a class's **public** and **private** members.

.NET Remoting Objects

- There are three types of objects that can be configured to serve as .NET remote objects. Choose the type of object depending on the requirement of the application.

1. **Single Call :** Single Call objects service one and only one request coming in. Single Call objects are useful in scenarios where the objects are required to do a limited amount of work. Single Call object are not required to store state information, in fact they cannot hold state information between method calls.

.NET Remoting Objects

2. Singleton Objects : These objects service multiple clients and hence share data by storing state information between client invocations. They are useful in cases in which data needs to be shared explicitly between clients.

.NET Remoting Objects

3. Client-Activated Objects : These objects are server-side objects that are activated upon request from the client.

- When the client submits a request for a server object using "new" operator, an activation request message is sent to the remote application.
- The server then creates an instance of the requested class and returns an ObjRef back to the client by using which proxy is then created. These objects can store state information between method calls for its specific client.
- Each invocation of "new" returns a proxy to an independent instance of the server type.

COMPONENTS OF REMOTING FRAMEWORK

- The main three components of a Remoting Framework are :

1. C# Remotable Object
2. C# Remote Listener Application - (listening requests for Remote Object)
3. C# Remote Client Application - (makes requests for Remote Object)

The Remote Object is implemented in a class that derives from System.MarshalByRefObject .

C# Remotable Object

- Any object outside the application domain of the caller application should be considered as **Remote Object** .
- A Remote Object that should be derived from **MarshalByRefObject** Class.
- Any object can be changed into a Remote Object by deriving it from MarshalByRefObject . Objects without inheriting from MarshalByRefObject are called Non-remotable Objects.

C# Remote Listener

- Choose and register a channel for handle the networking protocol and serialization formats and register the Type with the .NET Remoting System , so that it can use the channel to listen for requests for the Type.
- C# Remote Channels are Objects that responsible of handling the network protocols and serialization formats.

C# Remote Client

- The Client application for calling Remote Object's method in C# is simple and straight forward.
- The **.NET Remoting System** will intercept the client calls, forward them to the remote object, and return the results to the client. The Client Application have to register for the Remote Type also.

Steps for creating .NET Remoting

1. Create and register the channel of transport, the object used for marshalling,
   with ChannelServices. E.g. Use TCP, HTTP or SMTP channels.
2. Register the object with RemotingServices.

Hosting a Remoting Application (IS Server)

1. Client object registers a channel.
2. Creation of Proxy object (Client activated or Server Activated)
3. Calling the method of a remote object via proxy.
4. Client-side formatter formats the message and transmits it via the appropriate channel.
5. Server-side formatter reformats the message.
6. The specified function on a remote object is executed and the result is returned.
7. Above the process of formatting and reformatting is reversed and the result is returned to the client object.

Terminologies in Remoting

- **Proxy:** To avoid conjunction in networking. Main work is task Distributing. Two types of proxy.
  - Transparent proxy (There is no physical existence , Created by IIS server)
  - Real Proxy (Physical Existence)
- **Channel:** Channel provides the medium for transfer data from one location to another location. There are two types of channel.
  - TCP(work with Predefined root Connection oriented)
  - HTTP (No need predefined root)
  - Formatters: Change the data in an appropriate format that it can traverse through channels.

Terminologies in Remoting

- **Formatters:** Change the data in an appropriate format that it can traverse through channels.

  Two types of formatters
  - Binary
  - SOAP(Simple Object Access Protocol)
- **Sink:** Sink is used for security point of view. Before sending the data, the Data will be encrypted. Some additional bit will be added with the data to secure the data. Two types of sink
  - Envoy sink
  - Server Context Sink
- **Object Mode On Server:** SingleCall, Singleton

Example for Remoting

Create 3 applications:

1. class Library (Of which Remote Object will be created)
2. Server Application (Console Application)
3. Client Application (Window Application)

Example for Remoting
Class Library - remoteclass

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

namespace remoteclass
{
    public class xx:MarshalByRefObject
    {
        public int sum(int a, int b)
        {
            return a + b;
        }
    }
}
```

Example for Remoting Server Application

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
namespace remoteserver
{     class Program
   {          static voidMain(string[] args)
      {               TcpChannel ch=new TcpChannel(8085);
         ChannelServices.RegisterChannel(ch);
         RemotingConfiguration.RegisterWellKnownServiceType(typeof
                  (remoteclass.xx),"rahul",WellKnownObjectMode.Singleton);
         Console.Write("Sever is  Ready.....");
         Console.Read();
      }
   } }
```

Example for Remoting Client Application

```
using System;  using System.Collections.Generic;
using System.ComponentModel;  using System.Data;
using System.Drawing;  using System.Text;
using System.Windows.Forms; using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;  using System.Runtime.Remoting.Channels.Tcp;

namespace remoteclient
{
   public partial class Form1 : Form
   {
      //TcpChannel ch = new TcpChannel();
      remoteclass.xx obj = new remoteclass.xx();
     public Form1()
      {
         InitializeComponent();
      }
```

Example for Remoting Client Application

```
        private void button1_Click(object sender, System.EventArgs e)
        {
            //ChannelServices.RegisterChannel(ch);
            obj = (remoteclass.xx)Activator.GetObject(typeof(remoteclass.xx),
                        "tcp://localhost:8085/IMSC_CSl");
            int x = Int32.Parse(textBox1.Text);
            int y = Int32.Parse(textBox2.Text);
            textBox3.Text = (obj.sum(x, y)).ToString();
        }
    }
}
```

References

- http://msdn.microsoft.com
- http://www.csharphelp.com/
- http://www.csharp-station.com/
- http://www.csharpindex.com/
- http://www.c-sharpcorner.com/
- https://www.w3schools.com/cs/
- https://www.javatpoint.com/c-sharp-tutorial
- https://www.onlinebuff.com/article_understand-threading-and-types-of-threading-in-c-using-an-example_56.html
- https://www.geeksforgeeks.org/types-of-threads-in-c-sharp/
- https://www.bestprog.net/en/2018/11/03/reflection-of-types-getting-type-metadata-the-system-reflection-namespace-class-system-type-ways-to-get-information-about-type/
- https://www.bestprog.net/en/site-map/c/