Summary for exam 70-320 Developing XML Web Services

1. Understanding the .NET Framework
2. Creating and Managing Windows Services
   - 2.1. Windows services run in Windows 2000, Windows XP, and Windows NT
   - 2.2. You can also specify a Windows service to run in the security context of a specific user account that is different from the logged on user account or the default computer account.
   - 2.3. The Windows service architecture consists of three components:
     - 2.3.1. Service application. An application that consists of one or more services that provide the desired functionality
     - 2.3.2. Service controller application. An application that enables you to control the behavior of a service.
     - 2.3.3. Service Control Manager. A utility that enables you to control the services that are installed on a computer.
   - 2.4. Use the System.ServiceProcess namespace to create Windows services
     - 2.4.1. Use the ServiceBase class to create a Windows service. Override the following methods:
       - 2.4.1.1. OnStart, OnPause, OnStop, OnContinue, OnShutDown, OnCustomCommand, On PowerEvent.
     - 2.4.2. Use the ServiceInstaller class to install the class that extends the ServiceBase class to implement the service.
     - 2.4.3. Use the ServiceProcessInstaller classes to create and register the process in which your system runs.
     - 2.4.4. Use the ServiceController class (and the SCM) to start and stop a service. The class also controls the service through the following methods.
       - 2.4.4.1. Close. Disconnects the instance of the ServiceController class from the service and releases all the resources allocated for the application
       - 2.4.4.2. Continue. Resumes a service after it is paused
       - 2.4.4.3. ExecuteCommand. Enables you to execute a custom command on a service
       - 2.4.4.4. Pause. Pauses the service
       - 2.4.4.5. Refresh. Refreshes the values of all the properties
       - 2.4.4.6. Start. Starts the service
       - 2.4.4.7. Stop. Stops the service and all dependent services.
   - 2.5. Check the registry to see the registered services HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services.
   - 2.6. After a service is started it can exist in running, paused, or stopped state. Also can be in pending state that indicated that a command was issued but not completed.
   - 2.7. The services that use a single process space are called Win32OwnProcess services, whereas the services that share a process with other services are called Win32ShareProcess services.
   - 2.8. Use the System.Diagnostics namespace to monitor the status and debug the service when it is running.
     - 2.8.1. EventLog class. Reports errors in the event log
     - 2.8.2. PerformanceCounter class. To monitor resource utilization through performance counters.
     - 2.8.3. Trace, Debug classes. To debug and trace the execution of the code
   - 2.9. Steps to create a Windows Service
     - 2.9.1. Create a project with a Windows Service template
     - 2.9.2. In the service design windows change the following properties:
       - ServiceName. Name that the SCM uses to identifiy the service:

```
this.ServiceName = "MyWindowsService_CSharp2";
```

       - Name. Specifies the name of the class that extends the System.ServiceProcess.ServiceBase class
     - 2.9.3. Change the service name in the Main method.

```
ServicesToRun = new System.ServiceProcess.ServiceBase[] { new
MyWindowsService_CSharp() };
```

     - 2.9.4. In the service design window optionally change the following properties:
       - Autolog. Make it True to make entries in the system event log
       - CanStop. To enable the user to stop the service (it call the OnStop method)
       - CanShutdown. To make the SCM inform the service when the system in shutting down.
       - CanPauseAndContinue. To allow the user pause and continue the service.
       - CanHandlePowerEvent. To enable the service to handle computer power status.
     - 2.9.5. Change the file name which contains the class
     - 2.9.6. To add functionality, override the OnStart and OnStop methods.
   - 2.10. Windows service events

2.10.1. Start event. Trigered when starting the service with the SCM, which locates the .exe file of the event and call the OnStart method. Set the StartType property of the service using the ServiceStartMode enumeration values (Automatic, Manual, Disabled)
2.10.2. Stop event. Trigered by the SCM when stoping the service; the service's OnStop method is called. If the CanStop property is set to False, the SCM doesn't pass the stop command to the service.
2.10.3. Pause event. The SCM checks the value of the CanPauseAndContinue property, if it is set to true is passes the pause command and call the OnPause method.
2.10.4. OnContinue Event. The SCM checks the value of the CanPauseAndContinue property and send a Continue command to the service and call the OnContinue method.

2.11. System event logs.
Set the AutoLog property to access the system event logs. Whe the property is set to True the service is registered in the Application log.
- System lo
- Security log
- Application log

```
protected override void OnStart(string[] args)
{
   // TODO: Add code here to start your service.
   EventLog.WriteEntry("Starting MyWindowsServices");
}

protected override void OnStop()
{
   // TODO: Add code here to perform any tear-down
   // necessary to stop your service.
   EventLog.WriteEntry("Stopping MyWindowsServices");
}
```

2.12. Custom Event logs
2.12.1. Set the AutoLog property of tye service to False
2.12.2. Add an instance of the EventLog component from the Components toolbox in the Service application
2.12.3. Specify a value for the Source property and the name of the log file that you want to create in the Log property
2.12.4. To create an event log programmatically:
2.12.4.1. Add a reference to System.Diagnostics namespace
2.12.4.2. Use the CreateEventSource method of the EventLog class

```
private System.Diagnostics.EventLog eventLog1;
.
.
private void CreateEventLog()
{
   //Initialize the instance of the EventLog component
   eventLog1 = new System.Diagnostics.EventLog();

   /* Create an event log for your service. The code also checks if an
      event log already exists for your service application */
   if (!System.Diagnostics.EventLog.SourceExists("Transaction Service"))
   {
      System.Diagnostics.EventLog.CreateEventSource("Transaction Service",
         "Transaction Log");
   }
   //Specify the event log to use your service application as source
   eventLog1.Source = "Transaction Service";
}
```

You can call the CreateEventLog() method in the OnStart method. After you create a custom event log, you can direct your service to create entries in it. The following code shows how to write entries in a custom event log.

```
protected override void OnStart(string[] args)
```

```
{
   CreateEventLog();
   eventLog1.WriteEntry("MyWindowsService_CSharp Started",
      EventLogEntryType.Information);
}
```

2.13. Performance Counters

Use the PerformanceCounter class in the System.Diagnostics namespace.

2.13.1. To use existing performace counters.

2.13.1.1. Open your Windows service in design view.

2.13.1.2. Open Server Explorer.

2.13.1.3. Expand the Servers node, locate your server, and expand the Performance Counters node.

2.13.1.4. Select a performance counter category and expand it.

2.13.1.5. Locate the performance counter that you want to use. For example, you can use the Global # Of IL Bytes Jitted counter from the .NET CLR Jit category, which specifies the number of bytes JIT-compiled.

2.13.1.6. Drag and drop the counter.

2.13.1.7. Set the MachineName property to blank to make the performance-counter machine independent. Visual Studio adds the following code

```
private System.Diagnostics.PerformanceCounter performanceCounter1;

private void InitializeComponent()
{
   this.performanceCounter1 = new System.Diagnostics.PerformanceCounter();
   ((System.ComponentModel.ISupportInitialize)
      (this.performanceCounter1)).BeginInit();
   this.performanceCounter1.CategoryName = ".NET CLR Jit";
   this.performanceCounter1.CounterName = "Total # of IL Bytes Jitted";
   this.performanceCounter1.InstanceName = "_Global_";
}
```

2.13.2. To create custom performance counters.

2.13.2.1. Right-click the Performance Counter node in Server Explorer.

2.13.2.2. Choose Create New Category from the shortcut menu.

2.13.2.3. In the Performance Counter Builder window, specify a name for the performance counter category you create.

2.13.2.4. Click New to create a new counter. Specify the Name and Type properties of the counter.

2.13.2.5. Add it to your application by dragging the performance counter from Server Explorer to your application. You can then add code to increment, decrement, or set the value of the performance counter as shown in the following code.

```
// Increments the value of counter by 1
performanceCounter1.Increment();
// Increments the value of counter by 5
performanceCounter1.IncrementBy(5);
// Decreases the value of counter by 1
performanceCounter1.Decrement();
// Sets the value of counter to 10
performanceCounter1.RawValue = 10;
// Gets the value of counter
int val = performanceCounter1.RawValue;
```

2.14. Predefined installer classes.

- System.Diagnostics.EventLogInstaller. To install and configure default and custom event logs
- System.Diagnostics.PerformanceCounterInstaller. To install and configure the system and custom performance counter
- System.ServiceProcess.ServiceInstaller. One instance for each service within the service application. It performs task specific to a service
- System.ServiceProcess.ServiceProcessInstaller. One instance for the service application. It perfoms task common to all the service (like writing to the registry)
- System.Messaging.MessageQueueInstaller. Install and configures the message queues used by the service.

2.15. Adding installers
  2.15.1. You can either click the Add Installer link in the properties windows of the Component designer window, or rightclick the the component for which you want to create an installer and then chose Add Installer from the Shortcut menu.
  2.15.2. The system creates an installer class which derives from the System.Configuration.Install.Installer class. This class is marked with the attribute [RunInstaller(true)], which requires you to install its assembly by using the Installutil.exe tool (which makes sure all the either all the components are installed or not)
  2.15.3. That class containg the Installers collection, to which the installation components are added (EventLogInstallers, PerformanceCounterInstallers, Etc.)
2.16. To specify the security context for a service use the Account property of the ServiceProcessInstaller class.
  - LocalService. Account with extensive local privileges providing credentials to a remote server
  - LocalSystem. Account with nonprivileged user on the local computer. Anonymous to a remote server
  - NetworkService. Nonprivilieged local account providing credentials to a remote server
  - User. Use the context of a user. You need to provide username and password
2.17. To install a Windows service use Installutil <.exe filename>. To uninstall it use Installutil /u </exe filename>
2.18. By using the properties window of the service you can:
  - Start, stop, and pause a service
  - Specify the Startup type of a service
  - Specify the recover actions (fist, second, and subsquent failures) in case of a service failure
  - Specify a Lon on user account for a service
  - View services dependencies
2.19. Use the ServiceController class to connect and control a windows service. You can Start, Stop, Pause, Resume and perform custom commands on a service.
  2.19.1. The serviceController class passes requests to the SCM which then passes requests to the service
  2.19.2. There are 3 ways to create an instance of the ServiceController class
    2.19.2.1. From the Component tab (in the toolbox), drag a ServiceController component on the form of a Windows application and then specify the service to control (ServiceName property) and the Machine with the service (MachineName property).
    2.19.2.2. In the Server explorer locate the server with the service, right-click the service and chose Add to Designet from the shortcut menu. The component is added to the window form with the properties set.
    2.19.2.3. Programatically.

```
private System.ServiceProcess.ServiceController serviceController1;
this.serviceController1 = new System.ServiceProcess.ServiceController();
this.serviceController1.MachineName = " NancyD";
this.serviceController1.ServiceName = " DBWriterCS";
```

    2.19.2.4. After crating an instance of the serviceController class, you can use its methods to control the service:

```
Using System.ServiceProcess

private void StartService()
{
   /* Check the current status of the service before you
   try to change the current status of the service */
   if (DBWriterController.Status == ServiceControllerStatus.Stopped)
   {
      DBWriterController.Start();
      MessageBox.Show("Service Started");
   }
   else
   {
      MessageBox.Show("Service Running");
   }
}

private void PauseService()
{
```

```
    /* Check the current status of the service before you try to
       change the current status of the service*/
    if (DBWriterController.Status == ServiceControllerStatus.Running)
    {
        //Check whether you can pause the service
        if(DBWriterController.CanPauseAndContinue==true)
        {
            DBWriterController.Pause();
            MessageBox.Show("Service Paused");
        }
        else
            MessageBox.Show("You Can not Pause the Service");
        }
        else
            MessageBox.Show("Service not Running");
}
```

2.19.2.5. To run custom commands on the service appplication, use the following steps:

- Create a method that calls the ServiceController.ExecuteCommand method in the application that you use to control your service application.
- Override the OnCustomCommand method in your service application to specify the tasks that you want your service application to perform.

```
//The method in the controller application that calls ExecuteCommand method
//The paremater is a numeric value between 128 and 256.
void RunCommand()
{
    if(System.DateTime.Now.Hour <= 12 )
        MyServiceController.ExecuteCommand(200);
    else
        if(System.DateTime.Now.Hour <= 24 )
            MyServiceController.ExecuteCommand(220);
}

//Overriden the method in the service application
protected override void OnCustomCommand(int command)
{
    FileStream FS = new FileStream(@"c:\Temp\MyWindowsService_CSharp.txt",
        FileMode.OpenOrCreate, FileAccess.Write);
    StreamWriter SR = new StreamWriter(FS);
    if(command == 200)
    {
        SR.WriteLine("Good Morning");
        SR.Flush();
    }
    else
    {
        if (command==220)
        {
            SR.WriteLine("Good Evening");
            SR.Flush();
        }
    }
}
```

2.19.3. To retrieve a list of services using the ServiceController class, use:

```
using System;
using System.ServiceProcess;
using System.Collections;

namespace ConsoleApplication5
{
```

```
class Class1
{
    [STAThread]
    static void Main(string[] args)
    {
        ServiceController MyServiceController = new ServiceController();
        ServiceController[] services= ServiceController.GetServices();
        IEnumerator enumerator   = services.GetEnumerator();
        while (enumerator.MoveNext())
        {
            Console.WriteLine(((ServiceController)enumerator.Current).ServiceName);
        }
        Console.WriteLine("Press <Enter> to exit");
        Console.Read();
    }
}
}
```

2.20. To change resources such as event logs, performance counters, and database connections in Windows services without needing to recompile use the Dynamic Properties area of the properties window at design time. After setting those values, Visual Studio created code like this:

```
System.Configuration.AppSettingsReader configurationAppSettings = new
   System.Configuration.AppSettingsReader();
this.performanceCounter1.CategoryName = ((string)
   (configurationAppSettings.GetValue("performanceCounter1.CategoryName",
   typeof(string))));
this.performanceCounter1.CounterName = ((string)
   (configurationAppSettings.GetValue("performanceCounter1.CounterName",
   typeof(string))));
```

And also a configuration file like this:

```
<?xml version="1.0" encoding="Windows-1252"?>
<configuration>
   <appSettings>
      <!-- User application and configured property settings go here.-->
      <!-- Example: <add key="settingName" value="settingValue"/> -->
      <add key="PerformanceCounter1.CategoryName" value=".NET CLR Jit" />
      <add key="PerformanceCounter1.CounterName" value="Total # of IL Bytes
         Jitted" />
   </appSettings>
</configuration>
```

2.21. To debug a service:
- 2.21.1. Choose Processes from the Debug menu
- 2.21.2. Select Show system Processes
- 2.21.3. Select the process for the service and click Attach
- 2.21.4. Select Common Language Runtime
- 2.21.5. You only can debug the OnPause, OnContinue, and OnStop methods. To debug OnStart and Main methods, create a test harness service to run the service to debug.

3. Creating and Consuming Serviced Components
   - 3.1. Serviced components enable access to COM+ services such as automatic transaction mangement, object pooling, security, loosely coupled events (LCE), and just-in-time (JIT) activation.
   - 3.2. COM+ provides the infrastructure to create and deploy distributed multitier applications.
   - 3.3. The component-based approach of COM allowed you to create small, logical, reusable, and stand-alone modules that you could integrate into a single application.
   - 3.4. Com+ Services overview

COM+ services	Used to
Automatic transaction processing	Apply declarative transaction-processing features

XA Interoperability	Support the X/Open transaction-processing model
Synchronization (Activity)	Manage concurrency
Shared properties	Share a state among multiple objects within a server process
Role-based security	Apply role-based security permissions for authentication and access control. Security can be Declarative and Programmatic
Queued components	Provide asynchronous message queuing
Object pooling	Provide a pool of ready-made objects easy to be reused.
Object construction	Pass a persistent string value to a class instance on the construction of the instance
Loosely coupled events	Manage object-based events. Events act as *Publishers* sending messages about changes in their state to an event store. Other applications act as *Suscribers* by searching the event store for the event information.
Just-in-time activation	Activate a COM+ object on a method call and deactivate the object after the call returns (the client has only references to the object). A COM+ object can have the states: Exists/Activated, Exist/NonActivated, NonExistent.
Compensating resource managers (CRMs)	Apply atomicity and durability properties to nontransactional resources
COM transaction integrator (COMTI)	Encapsulates access to IBM's CICS and IMS applications in automation objects

3.5. Serviced Components

They are .NET components that use available component services of COM+. These components run within the managed execution environment of the .NET Framework and share their context with COM+ applications. To create them create a class that derives from the ServicedComponent base class.

```
using System.EnterpriseServices;
public class Account : ServicedComponent
{
   static void Main()
   {}
}
```

3.5.1. Setting attributes to Serviced components.

A serviced based component can utilize COM+ services by using service-related attributes. The following service-related attributes help to configure serviced components.

Attribute	Use	Scope	Unconf. Def.	Config. Def
ApplicationAccessControlAttribute	configure security	assembly	False	True
ApplicationActivationAttribute	Service runs either in the system's or the creator's process	assembl	Library	-
ApplicationIDAttribute	Specifies the globally unique identifier (GUID)	assembly	GUID	-
ApplicationNameAttribute	Specifies the name of the COM+ application that hosts the serviced components	assembly	the assembly name	
ApplicationQueuingAttribute	defines whether the serviced component can read messages from message queues	assembly	-	-
AutoCompleteAttribute	Specify whether the method automatically calls SetComplete or SetAbort	method	False	True
ComponentAccessControlAttribute	enables security checking on the method calls	class	False	True
COMTIIntrinsicsAttribute	enables you to pass the context properties from the COM transaction integrator (COMTI) to the COM+ context	class	False	True
ConstructionEnabledAttribute	to specify the initialization information externally so that you do not have to hard code the configuration information that is located inside a class	class	False	True
DescriptionAttribute	informational attribute	Assembly class Method interface	-	-
EventClassAttribute	specifies the class as an event class	class	-	-
EventTrackingEnabledAttribute	enables event tracking	class	False	True

TransactionsAttribute	specifies the type of transaction that is available (1) TransactionOption (2) TransactionIsolationLevel	Class	False	1.Required 2.Serializable Timeout
JustInTimeActivationAttribute	enables JIT activation	class	False	True
ObjectPoolingAttribute	to enable and configure object pooling	Class	False	True
LoadBalancingSupportedAttribute	determines whether the component participates in load balancin	class	False	True
MustRunInClientContextAttribute	enables the objects of the class to be created in the context of the creator	class	False	True
PrivateComponentAttribute	makes the objects of the class accessible only within the application	class	-	Private
SecureMethodAttribute	ensures secure calls to a method or to the methods within a class or assembly	Assembly Class method	-	-
SecurityRoleAttribute	to add security roles to an application and associate the security roles with components	Assembly Class method	-	-

Example of using attributes:

```
using System;
using System.Reflection ;
using System.EnterpriseServices;

[assembly: ApplicationName("MyServicedComponents")]
[assembly: AssemblyDescription("This app contains serviced components")]

   [ObjectPooling(MinPoolSize=1, MaxPoolSize=5, CreationTimeout=20000),
      Transaction(TransactionOption.Supported)]
   public class Account : System.EnterpriseServices.ServicedComponent
   {

      [AutoComplete]
      public void Debit()
      {
      }

      [AutoComplete]
      public void credit()
      {
      }
   }
```

3.5.2. After a component is created it is hosted in a COM+ application by performing the following tasks:

3.5.2.1. Assign a Strong Name to the Assembly.

3.5.2.1.1. Use the Strong Name Tool, Sn.exe, to create a file containing the public key information.

3.5.2.1.2. Use AssemblyKeyFileAttribute to assign a strong name to an assembly.

```
using System.Reflection;
[assembly: AssemblyKeyFile("MyKey.snk")]
public class Account : System.EnterpriseServices.ServicedComponent
{
}
```

3.5.2.2. Register the Serviced Component

- Register the assembly in the Windows registry
- Register and install the type library definitions in a COM+ application

3.5.2.2.1. Attributes in Serviced components.

During the process of registration, the serviced components are added to a COM+ application and configured according to the attributes used. COM+ applies default attributes to the components that are not configured with explicit values. You can apply the following attributes to serviced components so that they can access COM+ services

3.5.2.2.1.1. Application identity attribute.

It is estableshid by the ApplicationName and /or ApplicationID attributes. The default for the ApplicationName is the full name of the assembly. The ApplicationID server as an index for application searches made duing the registration process.

```
using System.EnterpriseServices;
[assembly: ApplicationName("MyServicedComponents")]
[assembly: ApplicationID("8fb2d46f-efc8-4643-bcd0-4e5bfa6a174c")]
public class Account : ServicedComponent
{
   static void Main()
   {}
}
```

3.5.2.2.1.2. Application activation type attribute.

Use this attribute to specify the activation typeof a COM+:

- Library of the caller process (Default)
- New process of the server.

  Here the dependent assemblies must be added to the Global Assembly Cache (GAC), and the parameters of the component must be marked as serializable to avoid exceptions.

```
using System.EnterpriseServices;
[assembly: ApplicationName("MyServicedComponents")]
[assembly: ApplicationID("8fb2d46f-efc8-4643-bcd0-4e5bfa6a174c")]
[assembly: ApplicationActivation(ActivationOption.Server)]
public class Account : ServicedComponent
{
   static void Main()
   {}
}
```

3.5.2.2.1.3. Description attribute.

With this attribute you can add a descriptive field specifying the assembly, class, interface, and method used in a particular component. The same value can be described in the Description field of the General Properties tab in the Component Services tool.

```
using System.EnterpriseServices;
using System.Reflection;

[assembly: ApplicationName("MyServicedComponents")]
[assembly: ApplicationID("8fb2d46f-efc8-4643-bcd0-4e5bfa6a174c")]
[assembly: ApplicationActivation(ActivationOption.Server)]
[assembly: AssemblyDescription("This app contains serviced components
")]
public class Account : ServicedComponent
{
   static void Main()
   {}
}
```

3.5.2.2.2. Type of registration.

COM+ services use the following three types of registrations for a serviced component:

3.5.2.2.2.1. Manual registration

It is utilized for design-time testing to learn about the types of errors that may occur during the execution of the application. Use the command line *RegSvcs.exe* tool, which perform the following taks during registration:

- Loads the assembly
- Registers the assembly
- Generates the type library
- Calls the LoadTypeLibrary method to register the type library
- Installs the type library into the specified COM+ application
- Configures the class

RegSvcs Options

/appname	To add the serviced components of MyFirstAssembly.dll to the COM+ application MyFirstApp, type the following at the command prompt: RegSvcs.exe /appname:MyFirstApp MyFirstAssembly.dll
/c	Create an application in a particular assembly. To create an application called MyFirstApp in MyFirstAssembly.dll, type the following at the command prompt: RegSvcs.exe /c MyFirstApp MyFirstAssembly.dll
/fc	Locate a COM+ application with a name and create it if not found. To locate an application called MyFirstApp in MyFirstAssembly.dll, type the following at the command prompt: RegSvcs.exe /fc MyFirstApp MyFirstAssembly.dll
/reconfig	Reconfigure an existing assembly version. To reconfigure the version of MyFirstAssembly.dll, type the following at the command prompt: RegSvcs.exe /reconfig /fc MyFirstApp MyFirstAssembly.dll

3.5.2.2.2.2. Dynamic registration

3.5.2.2.2.2.1. Utilized by clients (like ASP.NET) and they don't need to be place in the GAC (Global assembly cache). However for COM clients use manual registration

3.5.2.2.2.2.2. The assemby with the services component is copied into the COM+ directory and remains unregistred .

3.5.2.2.2.2.3. At the first call, the CLR creates an instance of it, register the assemby an the type library and configures the COM+ catalof accordingly.

3.5.2.2.2.3. Programmatic registration

3.5.2.2.2.3.1. Create an instance of the RegistrationHelper class.

Definition of the class

```
[Guid("")]
public sealed class RegistrationHelper : MarshalByRefObject,
    IRegistrationHelper
```

3.5.2.3. Example of using a serviced component

The Server component

```
using System.EnterpriseServices;
using System.Runtime.CompilerServices;
using System.Reflection;

// Specify a name for the COM+ application.
[assembly: ApplicationName("MyServicedComponents")]
// Specify a strong name for the assembly.
[assembly: AssemblyKeyFile("MyKey.snk")]
//Similar to using the COM+ Explorer to set the transaction support on a
//COM+ component
[Transaction(TransactionOption.Required)]
public class Account : ServicedComponent
{
   // Specify that the CLR calls the SetAbort method if an exception
  //  is generated during the execution of the method
// The SetComplete function is called otherwise.
    [AutoComplete]
    public bool Post(int accountNum, double amount)
    {
        // Calls SetAbort if an exception occurs
        // Calls SetComplete automatically if no exception is Generated.
        return false;
    }
}
```

The client component

```
public class Client
```

```
{
   public static int Main()
   {
      Account act = new Account();
      // Post money into the account.
      act.Post(5, 100);
      return 0;
   }
}
```

3.6. Utilizing COM+ Services

3.6.1. Features related with transactions

3.6.1.1. At the end of its execution, a transaction exists in one of the following states:

3.6.1.1.1. Committed.
Indicates the successful execution of all the tasks in a ?transaction.

3.6.1.1.2. Disable commit.
Indicates that the component has not finished its task, and the transactional updates are in an inconsistent state.

3.6.1.1.3. Enable commit.
Indicates that the task of the object is not finished, but its transactional updates are in a consistent state.

3.6.1.1.4. Aborted.
Indicates the unsuccessful execution of one of the tasks of the transaction, which resulted in the failure of the transaction.

3.6.1.2. A transaction processing system consists of:

3.6.1.2.1. Transaction manager (TM).
Implemented by the Microsoft Distributed Transaction Coordinator (MSDTC), ensuring that all parties in the transaction are notified of the outcome (commit or rollback). The TM also coordinates the recovery of one or more of the systems from failure.

3.6.1.2.2. Resource manager.
Microsoft Message Queue (MSMQ) is often used as a COM+ resource manager. It manages the ACID properties of a particular resource.

3.6.1.2.3. Transaction Processing Monitor (TP Monitor).
The TP Monitor is an environment present between the clients and the server resources to manage transactions, manage resources, provide load balancing and fault tolerance.

3.6.1.2.4. Resource dispensers.
Application components use resource dispensers to access shared information. Resource dispensers can also be used to load the auxiliary components.

3.6.2. Transactions attributes.
Use them to specify the requirements of a transaction.

Table 3-4. Characteristics of the COM+ Transaction Attribute Values

Transaction attribute	**Start a new transaction**	**Use a client's transaction**	**Transaction root**
Required	Maybe	Yes, if client has an existing transaction	Yes, if client does not have an existing a transaction
Requires New	Yes	Maybe ?	Maybe ?
Disabled	Never	Maybe	Never
Not Supported	Never	Never	Never

Table 3-4. Characteristics of the COM+ Transaction Attribute Values

Supported	Never	Yes, if client has an existing transaction	Never

3.6.3. Transaction Time-out

It is measured in seconds, and by default uses the systemwide time-out value set using the Components Services tool.

```
[Transaction(TransactionOption.Required, Isolation=
   TransactionIsolationLevel.Serializable, Timeout=25)]
```

3.6.4. JIT activation

Utilized to create an object that is a nonactive, context-only object. When a client invokes a method on the object, the run time creates the full object. COM+ deactivates the object when the method call returns, but leaves the context in memory. Its default value is True.

```
[JustInTimeActivation]
public class Account : ServicedComponent
```

3.6.5. Loosely Coupled Events Service.

You can make late-bound event or method calls to the publishers and subscribers within an event system. The event system provides the publisher and the subscriber with information as and when it is made available. This technique saves the task of repeatedly polling the server for information.

```
using System;
using System.IO;
using System.Reflection;
using System.EnterpriseServices;
using System.Runtime.InteropServices;
using System.Windows.Forms;

[assembly: ApplicationName("DemoLCE")]
[assembly: ApplicationActivation(ActivationOption.Library)]
[assembly: AssemblyKeyFile("DemoSvrLCE.snk")]

namespace DemoLCE
{
   public interface IEvent
   {
      void EvntMethod(string mess);
   }
//***Event Class
   [EventClass]
   public class LCEClass : ServicedComponent, IEvent
   {
      public void EvntMethod(string mess){}
   }
//*** Subscriber to the event Class
   public class LCESink : ServicedComponent, IEvent
   {
      public void EvntMethod(string mess)
      {
         MessageBox.Show(mess, "Event sink");
      }
   }
}

//Windows applications, The EvntMethod is called from the button handler
//***Publisher class
public class LCEClient : System.Windows.Forms.Form
{
   private System.Windows.Forms.Button FireButtonEvent;
```

```
    protected void FireEventButton_Click(object Mysender, System.EventArgs e)
    {
        IEvent MyEvent = (IEvent) new LCEClass();
        MyEvent.EvntMethod("This is a welcome message for events ");
    }
}
```

3.6.6. Using the Object Pooling Service.

The object is extracted from the pool on activation. Similarly, on deactivation, the object is returned to the pool for later use. To configure object pooling, you apply ObjectPoolingAttribute to a class that is derived from the System.?EnterpriseServices.ServicedComponent class.

```
[ObjectPooling(Enabled=true, MinPoolSize=1, MaxPoolSize=10,
    CreationTimeout=25000)]
public class MyPooledObject : ServicedComponent
{}
```

3.6.7. Using the Queue Components Service

- Utilised to invoke and execute COM+ components asynchronously.
- Apply the ApplicationQueuingAttribute class, which is derived from the System.Enterprise-Services.ServicedComponent class.
- Set the maximum number of queued components listener threads (0 – 1000) utilizing the MaxListenerThreads attribute.

```
[assembly:ApplicationQueuingAttribute(QueueListenerEnabled = true,
    MaxListenerThreads =  35 )]
```

3.6.8. Using object construction

Use object construction to initialize configuration information externally. Apply a ConstructionEnabled Attribute to a class that is derived from the System.EnterpriseServices.ServicedComponent class.

```
[ConstructionEnabled(Default="Hello")]
```

3.6.9. Implementing Security for serviced components

4. Creating and Consuming .NET Remoting Objects

4.1. *Communication channels* are the objects that transport messages between the remote objects.

4.2. *Serialization formatters* are the objects that help you to encode and decode messages that are sent to or received from a remote object. Two kinds of encoding are possible for all messages: binary and XML encoding.

4.3. *Transport channel*. Technology that performs functions such as opening a network connection, formatting messages, writing messages into streams, and sending bytes to the receiving application.

4.4. *Nonremotable objects* that do not provide the remoting system with a method to either copy them or use them in another application domain. Therefore, you can access these objects only in their own application domain.

4.5. *Remotable objects* can either be accessed outside their application domain or context using a proxy or copied and passed outside their application domain or context. Types of remotable objects:

4.5.1. Marshal-by-value objects. Copied and passed by value out of the application domain.

4.5.1.1. They implement the *ISerializable* interface or are marked with the *SerializableAttribute* attribute.

4.5.1.2. The copy in the client application domain handles any method call.

4.5.1.3. When marshal-by-value objects are passed as arguments, a copy of the object is passed to the method.

4.5.2. Marshal-by-reference objects. The client needs a proxy to access the object remotely.

4.5.2.1. They extend the *System.MarshalByRefObject* class.

4.5.2.2. If the client is in the same application domain as the marshal-by-reference object, the infrastructure returns a direct reference to the marshal-by-reference object, avoiding the overhead of marshaling.

4.5.3.

4.6. Remote Object Activation.

4.6.1. Server Activation.

Object are created on the server when you call a method in the server class (not when using the *new* keyword).

```
using System;
using System.Runtime.Remoting;
namespace SSNComponentCSharp
```

```
{
   public class SSNServer : MarshalByRefObject
   {
      public SSNServer()
      {
      }
      public String ValidateSSN(long number)
      {
         // Return the address
         String address;
         // Do some work here to validate the SSN
         return address;
      }
   }
}

//Calling program
SSNServer serverInstance = new SSNServer();
// Remote Object on the server is created in the next line
Console.WriteLine(serverInstance.ValidateSSN(242990307));
```

4.6.1.1. They can be created as Singleton or SingleCall objects (getting a default time).

- Singleton objects can have only one instance regardless of the number of clients they have. These objects also have a default lifetime.
- SingleCall object, here the remoting system creates an object each time a client method invokes a remote object.

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

namespace SSNComponentCSharp
{
   public class SSNServer
   {
      public SSNServer()
      {
         RemotingConfiguration.ApplicationName = "testService";
         RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(testService),
            "MyUri",WellKnownObjectMode.Singleton); //or SingleCall
         Console.WriteLine("Press enter to stop.");
         Console.ReadLine();
      }
   }
}

class testService : MarshalByRefObject
{
   // Service Object Registered in the SSNServer class above.
}
```

4.6.2. Client Activation.

- Created on the server when you create an instance using the new keyword.
- The client application domain defines the lifetimes of client-activated objects.
  - The instance of the remote object serves a particular client until its lease expires.
  - Then the object contacts the client and asks whether it should continue to exist and for how long.

- If the client is not currently available, a default time limit is specified for which the server object waits while trying to contact the client before marking itself available for garbage collection.

4.6.3. Lifetime Leases

They determine the duration of a marshal-by-reference object which is remoted outside an application domain.

- Each application domain contains a *lease manger* that administers the leases in its domain
- When a lease expires, the lease manager checks with the sponsors for renewals before deleting it.

```
using System;
using System.Runtime;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Lifetime;

public class MyLifetimeControlObject: MarshalByRefObject
{
   //Overrride this method to initialize lifetime lease
   public override object InitializeLifetimeService()
   {
      ILease lease = (ILease)base.InitializeLifetimeService();
      if (lease.CurrentState == LeaseState.Initial)
      {
         lease.InitialLeaseTime = TimeSpan.FromMinutes(2);
         lease.SponsorshipTimeout = TimeSpan.FromMinutes(3);
         lease.RenewOnCallTime = TimeSpan.FromSeconds(3);
      }
      return lease;
   }
}
```

- Renew the lease time by changing the *CurrentLeaseTime* property. Two ways to do it;
  - A client application call the Ilease.Renew method
  - A sponsor renews the lease

```
RemoteType obj = new RemoteType();
ILease lease = (ILease)RemotingServices.GetLifetimeService(obj);
TimeSpan expireTime = lease.Renew(TimeSpan.FromSeconds(30));
```

4.7. Channels

4.7.1. Enable communication between two applications running in different domain, process, or computer using protocols such as TCP and HTTP

4.7.2. A Channel implement the *Ichannel* interface of the *System.Runtime.Remoting.Channels* namespace

4.7.3. Channel are categorized in:

4.7.3.1. Receiver or Server. (implements IchannelReceiver).
Example: HttpServerChannel, TcpServerChannel.

4.7.3.2. Sender or Client. (implements IchannelSender).
Example: HttpClientChannel, TcpClientChannel
Note: HttpChannel and TcpChannel classes implement both.

4.7.4. Use the *RegisterChannel* static method of the *ChannelServices* class to register a channel with the remoting infrastructure.

```
TcpServerChannel channel = new TcpServerChannel(8010);
ChannelServices.RegisterChannel(channel);
```

4.7.5. General rules for selecting channels

4.7.5.1. Register at least one client channel (TCPClientChannel or HttpClientChannel) on the client computer and a server channel (TcpServerChannel or HttpServerChannel) in the remote computer

4.7.5.2. Register a server channel on the client computer if a remote object calls back a method on the client

4.7.5.3. You can have several channels of the same type but with different names

4.7.5.4. Two channels cannot listen on the same port

4.7.6. Marshaling. It us the process by which the parameters and call-related information are bundled into a message when calling a remote object.

4.7.7. Sinks. The remoting infrastructure send the marhsaled messages to the Sink object, which is an object that allows a client to establish a connection with the channel registered by the remote object.

4.7.8. HTTP Channels (System.Runtime.Remoting.Channels.Http)

- Utilized when interoperability between remote components is the main objective
- Use the SoapFormatter class to serialize messages into the XML format using the SOAP protocol.

4.7.8.1. Use the *HttpClientChannel* to transport messages from a client to a remote object.
The port number is optional (takes any available)

```
ChannelServices.RegisterChannel(new HttpClientChannel());
```

4.7.8.2. The *HttpServerChannel* allows a remote object t listen to the remote calls from clients.
The port number is required, (0 takes an avaialble port)

```
HttpServerChannel channel=new HttpServerChannel(8080);
ChannelServices.RegisterChannel(channel);
```

4.7.8.3. Use the *HttpChannel* class to transport messages to and from remote objects.

```
HttpChannel channel = new HttpChannel(8010);
ChannelServices.RegisterChannel(channel);
```

4.7.9. TCP Channels (System.Runtime.Remoting.Channels.Tcp)
Use the BinaryFormatter class to serialize messages into a ninary stream

4.7.9.1. The TcpClientChannel class allows a client to send messages to a remote object
ChannelServices.RegisterChannel(new TcpClientChannel());

4.7.9.2. The TcpServerChannel class allows a remote object to receive messages from clients.

```
TcpServerChannel channel=new TcpServerChannel(8070);
ChannelServices.RegisterChannel(channel);
```

4.7.9.3. The TcpChannel class allows you to transport messages to and from a remote object.

```
TcpChannel tcpchannel = new TcpChannel(8010);
ChannelServices.RegisterChannel(tcpchannel);
```

4.7.10. Sinks and Sinks Chains

4.7.10.1. The messages that channels send to or receive from remote objects pass through a chain of objects called channel sinks.

4.7.10.2. A channel sink performs certain functions on the message before forwarding the message to the next channel sink in the chain.

4.7.10.3. Within a channel sink chain, you can perform tasks, such as logging, applying filters, encrypting the message, and imposing security restrictions.

```
// The following code shows how to create a channel sink chain
private IClientChannelSinkProvider CreateSinkChain()
{
    IClientChannelSinkProvider chain = new FormatterSinkProvider_1();
    IClientChannelSinkProvider sink = chain;
    sink.Next = new FormatterSinkProvider_2();
    sink = sink.Next;
    return chain;
}
```

4.8. Implementing Events and Delegates

4.8.1. A *delegate* is a class that holds a reference to the method that is called when an event is fired.

```
//Declaring a delegete
public delegate void RetirementHandler(object sender, RetireEventArgs e);
//Declaring a method with the signature of the delegate
public class Action
{
    // RetireEvent has the same signature as RetirementHandler.
    public void RetireEvent (object sender, RetireEventArgs e)
    {}
}
//Create an instance of the delegate and store the address of the method whose reference the
delegate holds
```

```
Action a = new Action();
NewEventHandler handler = new RetirementHandler(a.RetireEvent);
```

4.8.2. Use delegates to implement:

4.8.2.1. Callback functions. To enable callback functions to the client in remoting applications, enabling the client and the remote application to function as domain servers.

4.8.2.2. Event programming

4.8.2.3. Asynchronous programming.

```
using System;
// Implement an EventArgs class
public class RetirementEventArgs : EventArgs
{
   private int ret_age;

   public RetirementEventArgs(int age)
{
   ret_age = age;
}

public int RetirementAge
{
   get
   {
      return ret_age;
   }
}
}

//Define Delegate to handle the RetirementEvent
public delegate void RetirementEventHandler(Object sender,
   RetirementEventArgs e);

//Implement a class that raises the retirement event
public class Employee : MarshalByRefObject
{
   public event RetirementEventHandler retirement;

   public void Submit()
   {
      RetirementEventArgs e = new RetirementEventArgs(58);
      retirement(this, e);
   }
}

// Implement class that handles the retirement event
public class HR
{
   public static void Main()
   {
      Employee emp = new Employee();
      HR hr = new HR();
      emp.retirement+= new RetirementEventHandler(hr.RetirementHandler);
   }

   public void RetirementHandler(Object sender, RetirementEventArgs e)
   {
      Console.WriteLine("Retirement Age is " + (int)e.RetirementAge);
   }
}
```

4.9. Implementing Asynchronous methods

4.9.1. Similar to the way it is implemented for single application domain or context

4.9.2. Implement asynchronous programming using the following ways:

- Use callbacks.
  You supply the callback delegate when you begin asynchronous calls.
- Poll completed.
  You poll the IAsyncResult.IsCompleted property to determine the completion of asynchronous calls.
- BeginInvoke, EndInvoke.
  You use these methods when you need to complete the operation prematurely.
- BeginInvoke, WaitHandle, EndInvoke.
  You use these methods to wait on IAsyncResult.

4.9.3. To implement asynchronous programming in a remoting application, complete the following steps:

1. Create an instance of an object that can receive a remote call to a method.
2. Wrap that instance method with an AsyncDelegate method.
3. Wrap the remote method with another delegate.
4. Call the BeginInvoke method on the second delegate, passing any arguments, the AsyncDelegate method, and some object to hold the state.
5. Wait for the server object to call your callback method.

4.9.4. How to call methods asynchronously in a .NET remoting application

```
// Create an instance of the class that can receive remote calls
MyRemoteObject remoteObj = new MyRemoteObject();

// Create delegate to a method that is executed when async method
// finishes execution.
AsyncCallback remoteMethod = new AsyncCallback(remoteObj.CallBackMethod);

// Create a delegate to the method that will be executed asynchronously
MyAsyncDelegate remoteDel = new MyAsyncDelegate(obj.LongCall);

// Begin the invocation of the asynchronous method.
remoteDel.BeginInvoke(remoteMethod,nothing);
```

4.10. Remote Object Configuration

4.10.1. Remote objects can be configured programatically or manually. In both cases the information to be provided includes:

- The activation type for the remote object
- The channels that the remote object will use to receive messages from clients
- The URL of the remote object
- The type metadata that describes the type of your remote object

4.10.2. The .NET Framework provides the *RemotingConfiguration* class in the *System.Runtime.Remoting* namespace that allows you to configure your remote components programmatically.

```
ChannelServices.RegisterChannel(new TcpChannel(8020));
WellKnownServiceTypeEntry myservice =
   new WellKnownServiceTypeEntry(typeof(MyService),"TcpService",
   WellKnownObjectMode.SingleCall);
RemotingConfiguration.ApplicationName = " TcpService";
RemotingConfiguration.RegisterWellKnownServiceType(myservice);
```

4.10.3. To configure manually an object, add the remoting configuration section to the application configuration file (or the Machine.config file).

```
<configuration>
   <system.runtime.remoting>
      <application>
         <service>
            <wellknown
               mode = "SingleCall"
               type = "MyRemoteClass,RemoteAssembly"
               objectUri = "MyApp.rem"
            />
```

```
            <activated
               type="ClientActivatedType, TypeAssembly"
            />
         </service>
      </application>
   </system.runtime.remoting>
</configuration>
```

- Use the RemotingConfiguration.Configure method to load the settings. Except when using the Machine.Config file, because it is loaded automatically.

```
RemotingConfiguration.Configure("MyApp.exe.config");
```

- Configuration Attributes Element

  <application>
  This element contains information about the remote objects that are exposed or consumed by an application.

  <service>
  This element contains the objects that are exposed by an application. The <service> element can occur one or more times in the <application> element.

  <wellknown>
  This element contains information about server-activated objects exposed by the application. The <wellknown> element can occur one or more times in the <service> element. The <wellknown> element has three required attributes: mode, type, and objectUri. Mode can be Singleton or SingleCall. Type specifies the type name and the name of the assembly that contains the type implementation. ObjectUri specifies the endpoint for the URI of an object.

  <activated>
  This element contains information about the client-activated objects that are exposed by the application. The <activated> element can occur one or more times in the <service> element. This element consists of one attribute, type, which specifies the type of the object and the assembly that implements the remote object type.

4.11. Securing .NET remoting object

4.11.1. Use code-access security.

4.11.2. If the remote object are host in IIS, use security features of IIS and SSL. Also you can use Windows authenticaion or Kerbeos.

4.11.3. Whenever you have a choice between using the HttpChannel and the TcpChannel class, you should use the HttpChannel class and host your remote objects in IIS.

5. 44