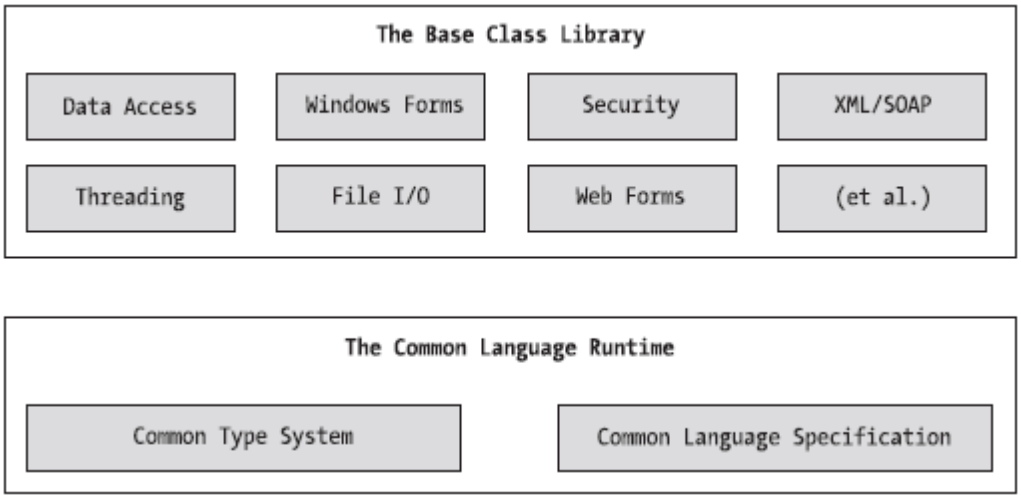
Q.2 Explain the following:
(i) The role of the Base class libraries
(ii) Compiling CIL to platform-specific instructions
(iii) The role of the Assembly Manifest
(iv) CTS Enumeration types

Answer:

i) The role of the Base class libraries

In addition to the CLR and CTS/CLS specifications, the .NET platform provides a base class library that is available to all .NET programming languages. Not only does this base class library encapsulate various primitives such as threads, file input/output (I/O), graphical rendering, and interaction with various external hardware devices, but it also provides support for a number of services required by most real-world applications.
For example, the base class libraries define types that facilitate database access, XML manipulation, programmatic security, and the construction of web-enabled (as well as traditional desktop and console-based) front ends. From a high level, you can visualize the relationship between the CLR, CTS, CLS, and the base class library, as shown in figure below.



The CLR, CTS, CLS and base class libraries relationship

ii) Compiling CIL to platform-specific instructions

Due to the fact that assemblies contain CIL instructions, rather than platform-specific instructions, CIL code must be compiled on the fly before use. The entity that compiles CIL code into meaningful CPU instructions is termed a *just-in-time* (*JIT*) *compiler*, which sometimes goes by the friendly name of *Jitter*. The .NET runtime environment leverages a JIT compiler for each CPU targeting the runtime, each optimized for the underlying platform.
For example, if you are building a .NET application that is to be deployed to a handheld device (such as a Pocket PC), the corresponding Jitter is well equipped to run within a low memory environment. On the other hand, if you are deploying your assembly to a back-end server (where memory is seldom an issue), the Jitter will be optimized to function in a high memory environment. In this way,

developers can write a single body of code that can be efficiently JIT-compiled and executed on machines with different architectures.
Furthermore, as a given Jitter compiles CIL instructions into corresponding machine code, it will cache the results in memory in a manner suited to the target operating system. In this way, if a call is made to a method named PrintDocument(), the CIL instructions are compiled into platform specific instructions on the first invocation and retained in memory for later use. Therefore, the next time PrintDocument() is called, there is no need to recompile the CIL.

iii) The role of the Assembly Manifest

A .NET assembly contains metadata that describes the assembly itself (technically termed a *manifest*). Among other details, the manifest documents all external assemblies required by the current assembly to function correctly, the assembly's version number, copyright information, and so forth. Like type metadata, it is always the job of the compiler to generate the assembly's manifest. Here are some relevant details of the CSharpCalculator.exe manifest:

```
.assembly extern mscorlib
{
      .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
      .ver 2:0:0:0
}
.assembly CSharpCalculator
{
      .hash algorithm 0x00008004
      .ver 0:0:0:0
}
.module CSharpCalculator.exe
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
```

In a nutshell, this manifest documents the list of external assemblies required by CSharpCalculator.exe (via the .assembly extern directive) as well as various characteristics of the assembly itself (version number, module name, and so on).

iv) CTS Enumeration types

Enumerations are a handy programming construct that allows you to group name/value pairs. For example, assume you are creating a video-game application that allows the player to select one of three character categories (Wizard, Fighter, or Thief). Rather than keeping track of raw numerical values to represent each possibility, you could build a custom enumeration using the enum keyword:

```
// A C# enumeration type.
public enum CharacterType {
      Wizard = 100,
      Fighter = 200,
      Thief = 300
}
```

By default, the storage used to hold each item is a 32-bit integer; however, it is possible to alter this storage slot if need be. Also, the CTS demands that enumerated types derive from a common base class, System.Enum. This base class defines a number of interesting members that allow you to extract, manipulate, and transform the underlying name/value pairs programmatically.

Q.3a. Explain the process of configuring the C# Command-line compiler.

Answer:

There are a number of techniques you may use to compile C# source code. In addition to Visual Studio 2005 (as well as various third-party .NET IDEs), you are able to create .NET assemblies using the C# command-line compiler, csc.exe (where csc stands for *C-Sharp Compiler*). This tool is included with the .NET Framework 2.0 SDK. While it is true that you may never decide to build a large-scale application using the command-line compiler, it is important to understand the basics of how to compile your *.cs files by hand. I can think of a few reasons you should get a grip on the process:

• The most obvious reason is the simple fact that you might not have a copy of Visual Studio 2005.

• You plan to make use of automated build tools such as MSBuild or NAnt.

• You want to deepen your understanding of C#. When you use graphical IDEs to build applications, you are ultimately instructing csc.exe how to manipulate your C# input files. In this light, it's edifying to see what takes place behind the scenes.

Another nice by-product of working with csc.exe in the raw is that you become that much more comfortable manipulating other command-line tools included with the .NET Framework 2.0 SDK. As you will see throughout this book, a number of important utilities are accessible only from the command line.

Configuring the C# Command-Line Compiler

Before you can begin to make use of the C# command-line compiler, you need to ensure that your development machine recognizes the existence of csc.exe. If your machine is not configured correctly, you are forced to specify the full path to the directory containing csc.exe before you can compile your C# files.

To equip your development machine to compile *.cs files from any directory, follow these steps (which assume a Windows XP installation; Windows NT/2000 steps will differ slightly):

1. Right-click the My Computer icon and select Properties from the pop-up menu.

2. Select the Advanced tab and click the Environment Variables button.

3. Double-click the Path variable from the System Variables list box.

4. Add the following line to the end of the current Path value (note each value in the Path variable is separated by a semicolon):

C:\Windows\Microsoft.NET\Framework\v2.0.50215

Of course, your entry may need to be adjusted based on your current version and location of the .NET Framework 2.0 SDK (so be sure to do a sanity check using Windows Explorer). Once you have updated the Path variable, you may take a test run by closing any command windows open in the background (to commit the settings), and then opening a new command window and entering csc /?

If you set things up correctly, you should see a list of options supported by the C# compiler.

b.With the help of an example, illustrate the process of compiling an application that make use of types defined in a separate .NET assembly.

Answer

Compiling an application that makes use of types defined in a separate .NET assembly is referred as **referencing external assemblies.**

To illustrate the process of referencing external assemblies, let's take an example of TestApp application to display aWindows Forms message box as below:

```
using System;

using System.Windows.Forms;
class TestApp {
      public static void Main() {
            Console.WriteLine("Testing! 1, 2, 3");
            MessageBox.Show("Hello...");
      }
}
```

In this example we have made a reference to the System.Windows.Forms namespace via the C# using keyword. When you explicitly list the namespaces used within a given *.cs file, you avoid the need to make use of fully qualified names (which can lead to hand cramps).

At the command line, you must inform csc.exe which assembly contains the "used" namespaces.

Given that you have made use of the MessageBox class, you must specify the System.Windows.Forms.dll assembly using the /reference flag (which can be abbreviated to /r):

```
csc /r:System.Windows.Forms.dll testapp.cs
```

If you now run your application, you should see what appears in following figure in addition to the console output.

c.Which class in C# allows you to obtain a number of details regarding the context of operating system hosting .Net application. Write a small code in C# to illustrate.

Answer: Page 88, Chapter 2 – Reference book

Q.4a. What do you mean by static methods and static classes in C#? List some of the properties of static constructor.

Answer:

Static Methods

Assume a class named Teenager that defines a static method named Complain(), which returns a random string, obtained in part by calling a private helper function named GetRandomNumber():

```
class Teenager {
      private static Random r = new Random();
      private static int GetRandomNumber(short upperLimit) {
            return r.Next(upperLimit);
      }
      public static string Complain() {
            string[] messages = new string[5]{ "Do I have to?",
                  "He started it!", "I'm too tired...", "I hate school!",
            "You are sooo wrong." } ;
            return messages[GetRandomNumber(5)];
      }
}
```

Notice that the System.Random member variable and the GetRandomNumber() helper function method have also been declared as static members of the Teenager class, given the rule that static members can operate *only* on other static members.

Thus, Static members can operate* only *on static class members. If you attempt to make use of non static class members (also called *instance data*) within a static method, you receive a compiler error.

Like any static member, to call Complain(), prefix the name of the defining class:

```
// Call the static Complain method of the Teenager class.
static void Main(string[] args) {
      for(int i = 0; i < 10; i++)
            Console.WriteLine("-> {0}", Teenager.Complain());
}
```

And like any nonstatic method, if the Complain() method was *not* marked static, you would need to create an instance of the Teenager class before you could hear about the gripe of the day:

```
// Nonstatic data must be invoked at the object level.
      Teenager joe = new Teenager();
      joe.Complain();
```

Static Classes

C# 2005 has widened the scope of the static keyword by introducing *static classes*. When a class has been defined as static, it is not creatable using the new keyword, and it can contain only static members or fields (if this is not the case, you receive compiler errors).

At first glance, this might seem like a very *useless* feature, given that a class that cannot be created does not appear all that helpful. However, if you create a class that contains nothing but static members and/or constant data, the class has no need to be allocated in the first place. Consider the following type:

```
// Static classes can only
// contain static members and constant fields.
static class UtilityClass {
      public static void PrintTime() {
```

```
            Console.WriteLine(DateTime.Now.ToShortTimeString());
      }
      public static void PrintDate() {
            Console.WriteLine(DateTime.Today.ToShortDateString());
      }
}
```

Given the static modifier, object users cannot create an instance of UtilityClass:

```
static void Main(string[] args) {
      UtilityClass.PrintDate();
      // Compiler error! Can't create static classes.
      UtilityClass u = new UtilityClass();
      ...
}
```

Prior to C# 2005, the only way to prevent an object user from creating such a type was to either redefine the default constructor as private or mark the class as an abstract type using the C# abstract keyword.

```
class UtilityClass {
      private UtilityClass(){}
      ...
}
abstract class UtilityClass {
      ...
}
```

While these constructs are still permissible, the use of static classes is a cleaner solution and more type-safe, given that the previous two techniques allowed non static members to appear within the class definition.

Following are the properties of **static constructors:**

- A given class (or structure) may define only a single static constructor.
- A static constructor executes exactly one time, regardless of how many objects of the type are created.
- A static constructor does not take an access modifier and cannot take any parameters.
- The runtime invokes the static constructor when it creates an instance of the class or before accessing the first static member invoked by the caller.
- The static constructor executes before any instance-level constructors.

b.With the help of syntax and an example, explain "foreach" statement used in C# programming.

Answer:

The **foreach** statement is similar to the **for** statement but implemented differently. It enables us to iterate the elements in arrays and collection classes such as **List** and **HashTable**. The general form of the **foreach** statement is:

```
      foreach (type variable in expression) {
            // Body of the loop
      }
```

The *type* and *variable* declares the *iteration* variable. During execution, the iteration variable represents

the array element (or collection element in case of collections) for which an iteration is currently being performed. **in** is a keyword. The *expression* must be an *array* or *collection* type and an explicit conversion must exist from the element type of the collection to the type of the iteration variable. Example is as follows:

```
public static void Main (string[] args) {
      foreach (string s in args) {
            Console.WriteLine(s);
      }
}
```

This program segment displays the command line arguments. The same may be achieved using the **for** statement as follows:

```
public static void Main (string[] args) {
      for (int i=0; i<args.Length; i++) {
            Console.WriteLine(args[i]);
      }
}
```

The following program illustrates the use of **foreach** statement for printing the contents of a numerical array.

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

namespace foreachloopdemo
{
    class ForeachTest {
        public static void Main() {
            int[] arrayInt = { 32, 56, 76, 21 };
            foreach (int m in arrayInt)
            {
                Console.Write(" " + m);
            }
            Console.WriteLine();
        }
    }
}
```

Q.5 a. Explain the three core principles of object-oriented programming, often called the famed "pillars of OOP".

Answer:

All object-oriented languages contend with three core principles of object-oriented programming, often called the famed "pillars of OOP."

- *Encapsulation*: How does this language hide an object's internal implementation?
- *Inheritance*: How does this language promote code reuse?
- *Polymorphism*: How does this language let you treat related objects in a similar way?

Encapsulation

The first pillar of OOP is called *encapsulation*. This trait boils down to the language's ability to hide unnecessary implementation details from the object user. For example, assume you are using a class named DatabaseReader that has two methods named Open() and Close():

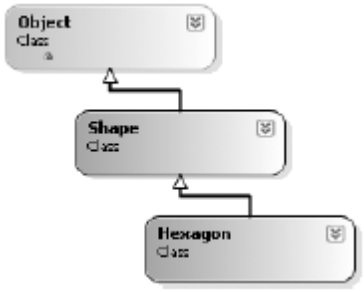
```
// DatabaseReader encapsulates the details of database manipulation.
DatabaseReader dbObj = new DatabaseReader();
dbObj.Open(@"C:\Employees.mdf");
// Do something with database...
dbObj.Close();
```

The fictitious DatabaseReader class has encapsulated the inner details of locating, loading, manipulating, and closing the data file. Object users love encapsulation, as this pillar of OOP keeps programming tasks simpler. There is no need to worry about the numerous lines of code that are working behind the scenes to carry out the work of the DatabaseReader class. All you do is create an instance and send the appropriate messages (e.g., "open the file named Employees.mdf located on my C drive").

Another aspect of encapsulation is the notion of data protection. Ideally, an object's state data should be defined as *private* rather than *public* (as was the case in previous chapters). In this way, the outside world must "ask politely" in order to change or obtain the underlying value.

Inheritance

The next pillar of OOP, inheritance, boils down to the language's ability to allow you to build new class definitions based on existing class definitions. In essence, inheritance allows you to extend the behavior of a base (or *parent*) class by enabling a subclass to inherit core functionality (also called a *derived class* or *child class*). Following figure illustrates the "is-a" relationship.



The "is-a" relationship

This diagram can be read as "A hexagon is-a shape that is-an object." When you have classes related by this form of inheritance, you establish "is-a" relationships between types. The "is-a" relationship is often termed *classical inheritance*.

The System.Object is the ultimate base class in any .NET hierarchy. Here, the Shape class extends Object. We can assume that Shape defines some number of properties, fields, methods, and events that are common to all shapes. The Hexagon class extends Shape and inherits the functionality defined by Shape and Object, in addition to defining its own set of members (whatever they may be).

There is another form of code reuse in the world of OOP: the containment/delegation model (also known as the "has-a" relationship). This form of reuse is not used to establish base/subclass relationships. Rather, a given class can define a member variable of another class and expose part or all of its functionality to the outside world.

For example, if you are modeling an automobile, you might wish to express the idea that a car "has-a" radio. It would be illogical to attempt to derive the Car class from a Radio, or vice versa.
(A Car "is-a" Radio? I think not!) Rather, you have two independent classes working together, where the containing class creates and exposes the contained class's functionality:

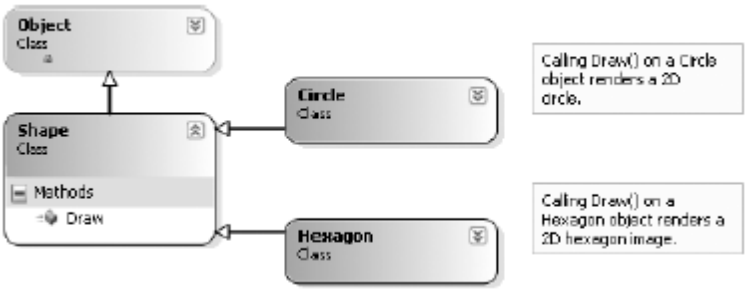
```
public class Radio {
      public void Power(bool turnOn) {
            Console.WriteLine("Radio on: {0}", turnOn);
      }
}
public class Car {
      // Car "has-a" Radio.
      private Radio myRadio = new Radio();
      public void TurnOnRadio(bool onOff) {
      // Delegate to inner object.
            myRadio.Power(onOff);
      }
}
```

Here, the containing type (Car) is responsible for creating the contained object (Radio). If the Car wishes to make the Radio's behavior accessible from a Car instance, it must extend its own public interface with some set of functions that operate on the contained type. Notice that the object user has no clue that the Car class is making use of an inner Radio object:

```
static void Main(string[] args) {
      // Call is forward to Radio internally.
      Car viper = new Car();
      viper.TurnOnRadio(true);
}
```

Polymorphism

The final pillar of OOP is *polymorphism*. This trait captures a language's ability to treat related objects the same way. This tenet of an object-oriented language allows a base class to define a set of members (formally termed the *polymorphic interface*) to all descendents. A class type's polymorphic interface is constructed using any number of *virtual* or *abstract* members. In a nutshell, a virtual member *may* be changed (or more formally speaking, *overridden*) by a derived class, whereas an abstract method *must* be overriden by a derived type. When derived types override the members defined by a base class, they are essentially redefining how they respond to the same request.
To illustrate polymorphism, let's revisit the shapes hierarchy. Assume that the Shape class has defined a method named Draw(), taking no parameters and returning nothing. Given the fact that every shape needs to render itself in a unique manner, subclasses (such as Hexagon and Circle) are free to override this method to their own liking (see figure below).



Classical Polymorphism

Once a *polymorphic interface* has been designed, you can begin to make various assumptions in your code. For example, given that Hexagon and Circle derive from a common parent (Shape), an array of Shape types could contain any derived class. Furthermore, given that Shape defines a polymorphic interface to all derived types (the Draw() method in this example), we can assume each member in the array has this functionaltiy. Ponder the following Main() method, which instructs an array of Shape-derived types to render themselves using the Draw() method:

```
static void Main(string[] args) {
        // Create an array of Shape derived items.
        Shape[] myShapes = new Shape[3];
        myShapes[0] = new Hexagon();
        myShapes[1] = new Circle();
        myShapes[2] = new Hexagon();
        // Iterate over the array and draw each item.
        foreach (Shape s in myShapes)
                s.Draw();
        Console.ReadLine();
}
```

b.How we prevent a class that cannot be further subclassed i.e. no further derivation of that class is possible? Explain.

Answer:

To prevent others from extending a class, make use of the C# sealed keyword.

```
public sealed class PTSalesPerson : SalesPerson {
      public PTSalesPerson(string fullName, int age, int empID,
            float currPay, string ssn, int numbOfSales)
            : base(fullName, age, empID, currPay, ssn, numbOfSales){
                  // Interesting constructor logic...
      }
      // Other interesting members...
}
```

Because PTSalesPerson is sealed, it cannot serve as a base class to any other type. Thus, if you attempted to extend PTSalesPerson, you receive a compiler error:

```
// Compiler error!
```

```
public class ReallyPTSalesPerson : PTSalesPerson
{ ... }
```
The sealed keyword is most useful when creating stand-alone utility classes. As an example, the String class defined in the System namespace has been explicitly sealed:
```
public sealed class string : object,
      IComparable, ICloneable, IConvertible, IEnumerable {...}
```
Therefore, you cannot create some new class deriving from System.String:
// **Another error!**
public class MyString : string
{...}
If you wish to build a new class that leverages the functionality of a sealed class, the only option is to forego classical inheritance and make use of the containment/delegation model.

c.Explain the meaning and use of C# keywords: virtual and override.

Answer
Polymorphism provides a way for a subclass to customize how it implements a method defined by its base class. If a base class wishes to define a method that *may be* overridden by a subclass, it must specify the method as virtual:
```
public class Employee {
      // GiveBonus() has a default implementation, however
      // child classes are free to override this behavior.
      public virtual void GiveBonus(float amount) {
            currPay += amount; }
            ...
}
```
When a subclass wishes to redefine a virtual method, it does so using the override keyword. For example, the SalesPerson and Manager could override GiveBonus() as follows (assume that PTSalesPerson overrides GiveBonus() in manner similar to SalesPerson):
```
public class SalesPerson : Employee {
      // A salesperson's bonus is influenced by the number of sales.
      public override void GiveBonus(float amount) {
            int salesBonus = 0;
            if(numberOfSales >= 0 && numberOfSales <= 100)
                  salesBonus = 10;
            else if(numberOfSales >= 101 && numberOfSales <= 200)
                  salesBonus = 15;
            else
                  salesBonus = 20; // Anything greater than 200.
            base.GiveBonus (amount * salesBonus);
      }
      ...
}

public class Manager : Employee {
```

```
      // Managers get some number of new stock options, in addition to
raw cash.
      public override void GiveBonus(float amount) {
            // Increase salary.
            base.GiveBonus(amount);
            // And give some new stock options...
            Random r = new Random();
            numberOfOptions += (ulong)r.Next(500);
      }
      ...
}
```

Notice how each overridden method is free to leverage the default behavior using the base keyword. In this way, you have no need to completely reimplement the logic behind GiveBonus(), but can reuse (and possibly extend) the default behavior of the parent class.

Also assume that Employee.DisplayStats() has been declared virtual, and has been overridden by each subclass to account for displaying the number of sales (for salespeople) and current stock options (for managers). Now that each subclass can interpret what these virtual methods means to itself, each object instance behaves as a more independent entity:

```
static void Main(string[] args) {
      // A better bonus system!
      Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-
2322", 9000);
      chucky.GiveBonus(300);
      chucky.DisplayStats();
      SalesPerson fran = new SalesPerson("Fran", 43, 93, 3000, "932-32-
3232", 31);
      fran.GiveBonus(200);
      fran.DisplayStats();
}
```

Q.6a. Differentiate between System-Level and Application-Level exceptions. What are the different ways to build your own custom exceptions?

Answer

System-level Exception

The .NET base class libraries define many classes derived from System.Exception. For example, the System namespace defines core error objects such as ArgumentOutOfRangeException, IndexOutOfRangeException, StackOverflowException, and so forth. Other namespaces define exceptions that reflect the behavior of that namespace (e.g., System.Drawing.Printing defines printing exceptions, System.IO defines IO-based exceptions, System.Data defines database-centric exceptions, and so forth).

Exceptions that are thrown by the CLR are (appropriately) called *system exceptions*. These exceptions are regarded as nonrecoverable, fatal errors. System exceptions derive directly from a base class named System.SystemException, which in turn derives from System.Exception (which derives from System.Object):

```
      public class SystemException : Exception {
            // Various constructors.
```

```
}
```

Given that the System.SystemException type does not add any additional functionality beyond a set of constructors, you might wonder why SystemException exists in the first place. Simply put, when an exception type derives from System.SystemException, you are able to determine that the .NET runtime is the entity that has thrown the exception, rather than the code base of the executing application.

Application-level Exception

Given that all .NET exceptions are class types, you are free to create your own application-specific exceptions. However, due to the fact that the System.SystemException base class represents exceptions thrown from the CLR, you may naturally assume that you should derive your custom exceptions from the System.Exception type. While you could do so, best practice dictates that you instead derive from the System.ApplicationException type:

```
public class ApplicationException : Exception {
      // Various constructors.
}
```

Like SystemException, ApplicationException does not define any additional members beyond a set of constructors. Functionally, the only purpose of System.ApplicationException is to identify the source of the (nonfatal) error. When you handle an exception deriving from System.ApplicationException, you can assume the exception was raised by the code base of the executing application, rather than by the .NET base class libraries.

Building Custom Exceptions, Take One

While you can always throw instances of System.Exception to signal a runtime error , it is sometimes advantageous to build a *strongly typed exception* that represents the unique details of your current problem. For example, assume you wish to build a custom exception (named CarIsDeadException) to represent the error of speeding up a doomed automobile. The first step is to derive a new class from System.ApplicationException (by convention, all exception classes end with the "Exception" suffix).

```
// This custom exception describes the details of the car-is-dead condition.
public class CarIsDeadException : ApplicationException
{}
```

Like any class, you are free to include any number of custom members that can be called within the catch block of the calling logic. You are also free to override any virtual members defined by your parent classes. For example, we could implement CarIsDeadException by overriding the virtual Message property:

```
public class CarIsDeadException : ApplicationException {
      private string messageDetails;
      public CarIsDeadException(){ }
      public CarIsDeadException(string message) {
            messageDetails = message;
      }
// Override the Exception.Message property.
      public override string Message {
            get {
                  return   string.Format("Car   Error   Message:   {0}",
```

```
            messageDetails);
            }
      }
}
```

Here, the CarIsDeadException type maintains a private data member (messageDetails) that represents data regarding the current exception, which can be set using a custom constructor. Throwing this error from the Accelerate() is straightforward. Simply allocate, configure, and throw a CarIsDeadException type rather than a generic System.Exception:

// **Throw the custom CarIsDeadException.**

```
public void Accelerate(int delta) {
      ...
      CarIsDeadException ex = new CarIsDeadException(string.Format("{0}
has overheated!",
            petName));
      ex.HelpLink = "http://www.CarsRUs.com";
      ex.Data.Add("TimeStamp", string.Format("The car exploded at {0}",
DateTime.Now));
      ex.Data.Add("Cause", "You have a lead foot.");
      throw ex;
      ...
}
```

To catch this incoming exception explicitly, your catch scope can now be updated to catch a specific CarIsDeadException type (however, given that CarIsDeadException "is-a" System.Exception, it is still permissible to catch a generic System.Exception as well):

```
static void Main(string[] args) {
      ...
      catch (CarIsDeadException e) {
            // Process incoming exception.
      }
      ...
}
```

So, now that you understand the basic process of building a custom exception, you may wonder when you are required to do so. Typically, you only need to create custom exceptions when the error is tightly bound to the class issuing the error (for example, a custom File class that throws a number of file-related errors, a Car class that throws a number of car-related errors, and so forth). In doing so, you provide the caller with the ability to handle numerous exceptions on an error-by-error basis.

Building Custom Exceptions,Take Two

The current CarIsDeadException type has overridden the System.Exception.Message property in order to configure a custom error message. However, we can simplify our programming tasks if we set the parent's Message property via an incoming constructor parameter. By doing so, we have no need to write anything other than the following:

```
public class CarIsDeadException : ApplicationException {
      public CarIsDeadException(){ }
      public CarIsDeadException(string message)
            : base(message){ }
```

```
}
```

Notice that this time you have *not* defined a string variable to represent the message, and have *not* overridden the Message property. Rather, you are simply passing the parameter to your base class constructor.

With this design, a custom exception class is little more than a uniquely named class deriving from System.ApplicationException, devoid of any member variables (or base class overrides).

Don't be surprised if most (if not all) of your custom exception classes follow this simple pattern.

Many times, the role of a custom exception is not necessarily to provide additional functionality beyond what is inherited from the base classes, but to provide a *strongly named type* that clearly identifies the nature of the error.

Building Custom Exceptions,Take Three

If you wish to build a truly prim-and-proper custom exception class, you would want to make sure your type adheres to the exception-centric .NET best practices. Specifically, this requires that your custom exception:

- Derives from Exception/ApplicationException
- Is marked with the [System.Serializable] attribute
- Defines a default constructor
- Defines a constructor that sets the inherited Message property
- Defines a constructor to handle "inner exceptions"
- Defines a constructor to handle the serialization of your type

Now, based on your current background with .NET, you may have no idea regarding the role of attributes or object serialization, which is just fine. I'll address these topics at this point later in the text. However, to finalize our examination of building custom exceptions, here is the final iteration of CarIsDeadException:

```
[Serializable]
public class CarIsDeadException : ApplicationException {
      public CarIsDeadException() { }
      public CarIsDeadException(string message) : base( message ) { }
      public CarIsDeadException(string message,
            System.Exception inner) : base( message, inner ) { }
      protected CarIsDeadException(
            System.Runtime.Serialization.SerializationInfo info,
            System.Runtime.Serialization.StreamingContext context)
                  : base( info, context ) { }
}
```

b.Explain the following:

(i) Generic catch Statements

(ii) Rethrowing Exceptions

(iii) The finally Block

Answer

i) Generic catch Statements

C# also supports a "generic" catch scope that does not explicitly receive the exception object thrown by a given member:

```
// A generic catch.
```

```
static void Main(string[] args) {
      ...
      try {
            for(int i = 0; i < 10; i++)
                  myCar.Accelerate(10);
      } catch {
            Console.WriteLine("Something bad happened...");
      }
      ...
}
```

Obviously, this is not the most informative way to handle exceptions, given that you have no way to obtain meaningful data about the error that occurred (such as the method name, call stack, or custom message). Nevertheless, C# does allow for such a construct.

ii) Rethrowing Exceptions

Be aware that it is permissible for logic in a try block to *rethrow* an exception up the call stack to the previous caller. To do so, simply make use of the throw keyword within a catch block. This passes the exception up the chain of calling logic, which can be helpful if your catch block is only able to partially handle the error at hand:

```
// Passing the buck.
static void Main(string[] args) {
      ...
      try {
            // Speed up car logic...
      } catch(CarIsDeadException e) {
            // Do any partial processing of this error and pass the
            buck.
            // Here, we are rethrowing the incoming CarIsDeadException
            object.
            // However, you are also free to throw a different
            exception if need be.
            throw e;
      }
      ...
}
```

Be aware that in this example code, the ultimate receiver of CarIsDeadException is the CLR, given that it is the Main() method rethrowing the exception. Given this point, your end user is presented with a system-supplied error dialog box. Typically, you would only rethrow a partial handled exception to a caller that has the ability to handle the incoming exception more gracefully.

iii) The Finally Block

A try/catch scope may also define an optional finally block. The motivation behind a finally block is to ensure that a set of code statements will *always* execute, exception (of any type) or not. To illustrate, assume you wish to always power down the car's radio before exiting Main(), regardless of any handled exception:

```
static void Main(string[] args) {
      ...
```

```
      Car myCar = new Car("Zippy", 20);
      myCar.CrankTunes(true);
      try {
            // Speed up car logic.
      } catch(CarIsDeadException e) {
            // Process CarIsDeadException.
      }
      catch(ArgumentOutOfRangeException e) {
            // Process ArgumentOutOfRangeException.
      }
      catch(Exception e) {
            // Process any other Exception.
      }
      finally {
            // This will always occur. Exception or not.
            myCar.CrankTunes(false);
      }
      ...
}
```

If you did not include a finally block, the radio would not be turned off if an exception is encountered (which may or may not be problematic). In a more real-world scenario, when you need to dispose of objects, close a file, detach from a database (or whatever), a finally block ensures a location for proper cleanup.

Q.7 a. What is an Interface? What do you mean by "Extending" an interface and "Implementing" interface? Explain with the help of example.

Answer

An **interface** can contain one or more methods, properties, indexers and events but none of them are implemented in the interface itself. It is responsibility of the class that implements the interface to define the code for implementation of these members.

The syntax for defining an interface is very similar to that used for defining a class. The general form of an interface definition is:

```
interface InterfaceName {
      member declarations;
}
```

Here, **interface** is keyword and ***InterfaceName*** is a valid C# identifier. Member declarations will contain only a list of members without implementation code. For example, below is a simple interface that defines a single method:

```
interface Show {
      void Display();
}
```

In addition to methods, interfaces can declare properties, indexers and events. Example is as follows:

```
interface ABC {
      int ABCproperty {
            get;
      }
      event someEvent Changed;
      void Display();
}
```

The accessibility of interface can be controlled by using the modifiers **public, protected, internal, and private.** The use of a particular modifier depends on the contact in which the interface declaration occurs.

Extending An Interface

Like classes, interfaces can also be extended. That is, an interface can be subinterfaced from other interfaces. The new suninterface will inherit all the members of the superinterface in the manner similar to subclasses. This is achieved as follows:

```
interface I2 : I1 {
      members of I2;
}
```

For example, we can put all members of particular behaviour category in one interface and the members of another category in the other. Consider the code below:

```
interface Addition (
      int Add(int x, int y);
}

interface Compute : Addition {
      int Sub(int x, int y);
}
```

The interface **Compute** will have both the methods and any class implementing the interface **Compute** should implement both of them; otherwise, it is an error.
We can also combine several interfaces together into a single interface. Following declarations are valid:

```
interface I1 {
      --------
}
interface I2 {
      --------
}

interface I3 : I1, I2 {          // multiple inheritance
      ----------
}
```

While interfaces are allowed to extend other interfaces, subinterfaces cannot define the methods declared in the superinterfaces. After all, subinterfaces are still interfaces, not classes. It is the responsibility of the class that implements the derived interface to define all the methods.
It is important to remember that an interface cannot extend classes. This would violate the rule that an

interface can have only abstract members.

Implementing Interface

Interfaces are used as 'superclasses' whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. this is done as follows:

```
class classname : interfacename {
      // body of class
}
```

Here the class **classname** 'implements' the interface **interfacename**. A more general for of implementation may look like this:

```
class classname : superclass, interface1, interface2 ... {
      // body of class
}
```

This shows that a class can extend another class while implementing interfaces.
In C#, we can drive from a single class and, in addition, implement as many interfaces as the class needs. When a class inherits from a superclass, the name of each interface to be implemented must appear after the superclass name. Examples:

```
class A : B, I1, I2 {
      // body of class
}
```

where B is the base class and I1, I2 are interfaces. The base class and interfaces are separated by commas.

b. Explain with the help of program code, how interfaces can be passed to methods as parameters and can also be used as method return values.

Answer

Interfaces as parameters

Given that interfaces are valid .NET types, you may construct methods that take interfaces as parameters. To illustrate, assume you have defined another interface named IDraw3D:

```
// Models the ability to render a type in stunning 3D.
public interface IDraw3D {
      void Draw3D();
}
```

Next, assume that two of your three shapes (Circle and Hexagon) have been configured to support this new behavior:

```
// Circle supports IDraw3D.
public class Circle : Shape, IDraw3D {
...
      public void Draw3D() {
            Console.WriteLine("Drawing Circle in 3D!");
      }
}

// Hexagon supports IPointy and IDraw3D.
// This interface defines the behavior of "having points."
```

```
public interface IPointy {
// Implicitly public and abstract.
      byte GetNumberOfPoints();
}

public class Hexagon : Shape, IPointy, IDraw3D {
      ...
      public void Draw3D() {
            Console.WriteLine("Drawing Hexagon in 3D!");
      }
}
```

If you now define a method taking an IDraw3D interface as a parameter, you are able to effectively send in *any* object implementing IDraw3D (if you attempt to pass in a type not supporting the necessary interface, you receive a compile-time error). Consider the following:

```
// Make some shapes. If they can be rendered in 3D, do it!
public class Program {
// I'll draw anyone supporting IDraw3D.
      public static void DrawIn3D(IDraw3D itf3d) {
            Console.WriteLine("-> Drawing IDraw3D compatible type");
            itf3d.Draw3D();
      }
      static void Main() {
            Shape[] s = { new Hexagon(), new Circle(),
                  new Triangle(), new Circle("JoJo")} ;
            for(int i = 0; i < s.Length; i++) {
                  ...
                  // Can I draw you in 3D?
                  if(s[i] is IDraw3D)
                  DrawIn3D((IDraw3D)s[i]);
            }
      }
}
```

Interfaces As Return Values

Interfaces can also be used as method return values. For example, you could write a method that takes any System.Object, checks for IPointy compatibility, and returns a reference to the extracted interface:

```
// This method tests for IPointy-compatibility and,
// if able, returns an interface reference.
static IPointy ExtractPointyness(object o) {
      if (o is IPointy)
            return (IPointy)o;
      else
            return null;
}
```

We could interact with this method as follows:

```
static void Main(string[] args) {
      // Attempt to get IPointy from Car object.
```

```
      Car myCar = new Car();
      IPointy itfPt = ExtractPointyness(myCar);
      if(itfPt != null)
            Console.WriteLine("Object has {0} points.", itfPt.Points);
      else
            Console.WriteLine("This object does not implement IPointy");
}
```

Q.8 a. What is .NET delegate type? Explain the concept and syntax of delegate in C#.

Answer

In the .NET Framework, callbacks are still possible, and their functionality is accomplished in a much safer and more object-oriented manner using *delegates*. In essence, a delegate is a type-safe object that points to another method (or possibly multiple methods) in the application, which can be invoked at a later time. Specifically speaking, a delegate type maintains three important pieces of information:

- The *name* of the method on which it makes calls
- The *arguments* (if any) of this method
- The *return value* (if any) of this method

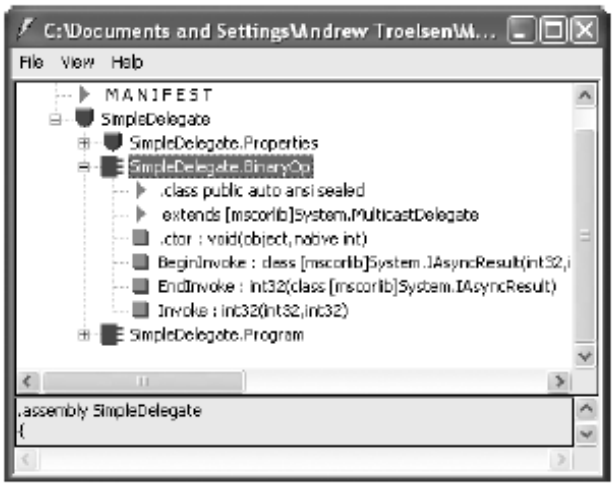
Once a delegate has been created and provided the aforementioned information, it may dynamically invoke the method(s) it points to at runtime. As you will see, every delegate in the .NET Framework (including your custom delegates) is automatically endowed with the ability to call their methods *synchronously* or *asynchronously*. This fact greatly simplifies programming tasks, given that we can call a method on a secondary thread of execution without manually creating and managing a Thread object.

When you want to create a delegate in C#, you make use of the delegate keyword. The name of your delegate can be whatever you desire. However, you must define the delegate to match the signature of the method it will point to. For example, assume you wish to build a delegate named BinaryOp that can point to any method that returns an integer and takes two integers as input parameters:

```
// This delegate can point to any method,
// taking two integers and returning an integer.

public delegate int BinaryOp(int x, int y);
```

When the C# compiler processes delegate types, it automatically generates a sealed class deriving from System.MulticastDelegate. This class (in conjunction with its base class, System.Delegate) provides the necessary infrastructure for the delegate to hold onto the list of methods to be invoked at a later time. For example, if you examine the BinaryOp delegate using ildasm.exe, you would find the items as shown in following figure.



The C# delegate keyword represents a sealed class deriving from System.MulticastDelegate

The generated BinaryOp class defines three public methods. Invoke() is perhaps the core method, as it is used to invoke each method maintained by the delegate type in a *synchronous* manner, meaning the caller must wait for the call to complete before continuing on its way.
Strangely enough, the synchronous Invoke() method is *not* directly callable from C#. Invoke() is called behind the scenes when you make use of the appropriate C# syntax. BeginInvoke() and EndInvoke() provide the ability to call the current method *asynchronously* on a second thread of execution. One of the most common reason developers create secondary threads of execution is to invoke methods that require time to complete. Although the .NET base class libraries provide an entire namespace devoted to multithreaded programming (System.Threading), delegates provide this functionality out of the box.
Now, how exactly does the compiler know how to define the Invoke(), BeginInvoke(), and EndInvoke() methods? To understand the process, here is the crux of the generated BinaryOp class type:

```
sealed class BinaryOp : System.MulticastDelegate {
      public BinaryOp(object target, uint functionAddress);
      public int Invoke(int x, int y);
      public IAsyncResult BeginInvoke(int x, int y,
      AsyncCallback cb, object state);
      public int EndInvoke(IAsyncResult result);
}
```

First, notice that the parameters and return value defined for the Invoke() method exactly match the definition of the BinaryOp delegate. The initial parameters to BeginInvoke() members (two integers in our case) are also based on the BinaryOp delegate; however, BeginInvoke() will always provide two final parameters (of type AsyncCallback and object) that are used to facilitate asynchronous method invocations. Finally, the return value of EndInvoke() is identical to the original delegate declaration and will always take as a sole parameter an object implementing the IAsyncResult interface.

b.Explain the meaning of following members of System. Multicast Delagate / System

Delegate:
(i) Target
(ii) Combine( )
(iii) GetInvocation List( )
(iv) Remove( )

Answer

i) Target: If the method to be called is defined at the object level (rather than a static method), Target returns an object that represents the method maintained by the delegate. If the value returned from Target equals null, the method to be called is a static member.
ii) Combine(): This static method adds a method to the list maintained by the delegate. In C#, you trigger this method using the overloaded += operator as a shorthand notation.
iii) GetInvocationList(): This method returns an array of System.Delegate types, each representing a particular method that may be invoked.
iv) Remove() & RemoveAll(): These static methods removes a method (or all methods) from the invocation list. In C#, the Remove() method can be called indirectly using the overloaded -= operator.

Q.9 a. What do you mean by shared assemblies? Describe the process of assigning a strong name to assembly before you can deploy an assembly into the Global Assembly Cache (GAC).

Answer

Like a private assembly, a shared assembly is a collection of types and (optional) resources. The most obvious difference between shared and private assemblies is the fact that a single copy of a shared assembly can be used by several applications on a single machine.

a shared assembly is not deployed within the same directory as the application making use of it. Rather, shared assemblies are installed into the Global Assembly Cache (GAC). The GAC is located under a subdirectory of your Windows directory named Assembly (e.g., C:\Windows\Assembly)

Before you can deploy an assembly to the GAC, you must assign it a *strong name*, which is used to uniquely identify the publisher of a given .NET binary. Understand that a "publisher" could be an individual programmer, a department within a given company, or an entire company at large.

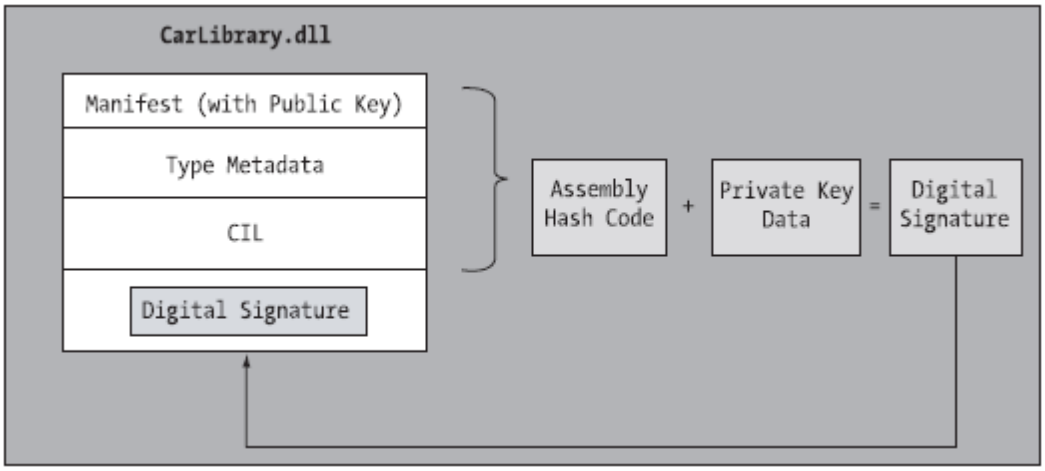
In some ways, a strong name is the modern day .NET equivalent of the COM globally unique identifier (GUID) identification scheme. If you have a COM background, you may recall that AppIDs are GUIDs that identify a particular COM application. Unlike COM GUID values (which are nothing more than 128-bit numbers), strong names are based on two cryptographically related keys (termed the *public key* and the *private key*), which are much more unique and resistant to tampering than a simple GUID.

Formally, a strong name is composed of a set of related data, much of which is specified using assembly-level attributes:

• The friendly name of the assembly (which you recall is the name of the assembly minus the file extension)
• The version number of the assembly (assigned using the [AssemblyVersion] attribute)
• The public key value (assigned using the [AssemblyKeyFile] attribute)
• An optional culture identity value for localization purposes (assigned using the [AssemblyCulture]

attribute)
• An embedded *digital signature* created using a hash of the assembly's contents and the private key value

To provide a strong name for an assembly, your first step is to generate public/private key data using the .NET Framework 2.0 SDK's sn.exe utility (which you'll do momentarily). The sn.exe utility responds by generating a file (typically ending with the *.snk [Strong Name Key] file extension) that contains data for two distinct but mathematically related keys, the "public" key and the "private" key.
Once the C# compiler is made aware of the location for your *.snk file, it will record the full public key value in the assembly manifest using the .publickey token at the time of compilation.
The C# compiler will also generate a hash code based on the contents of the entire assembly (CIL code, metadata, and so forth). A *hash code* is a numerical value that is unique for a fixed input. Thus, if you modify any aspect of a .NET assembly (even a single character in a string literal) the compiler yields a unique hash code. This hash code is combined with the private key data within the *.snk file to yield a digital signature embedded within the assembly's CLR header data. The process of strongly naming an assembly is illustrated in following figure.



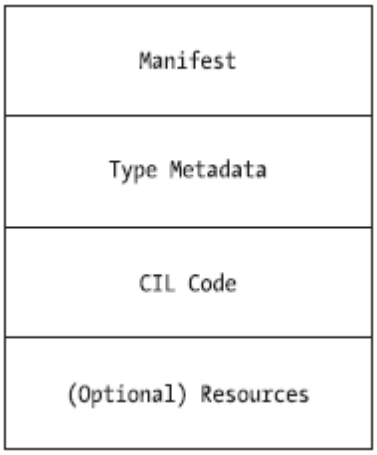
At compile time, a digital signature is generated and embedded into assembly based on public and private key data

The actual *private* key data is not listed anywhere within the manifest, but is used only to digitally sign the contents of the assembly (in conjunction with the generated hash code). Again, the whole idea of making use of public/private key cryptography is to ensure that no two companies, departments, or individuals have the same identity in the .NET universe. In any case, once the process of assigning a strong name is complete, the assembly may be installed into the GAC.

b. Compare and contrast Single-file and Multifile Assemblies with suitable figure

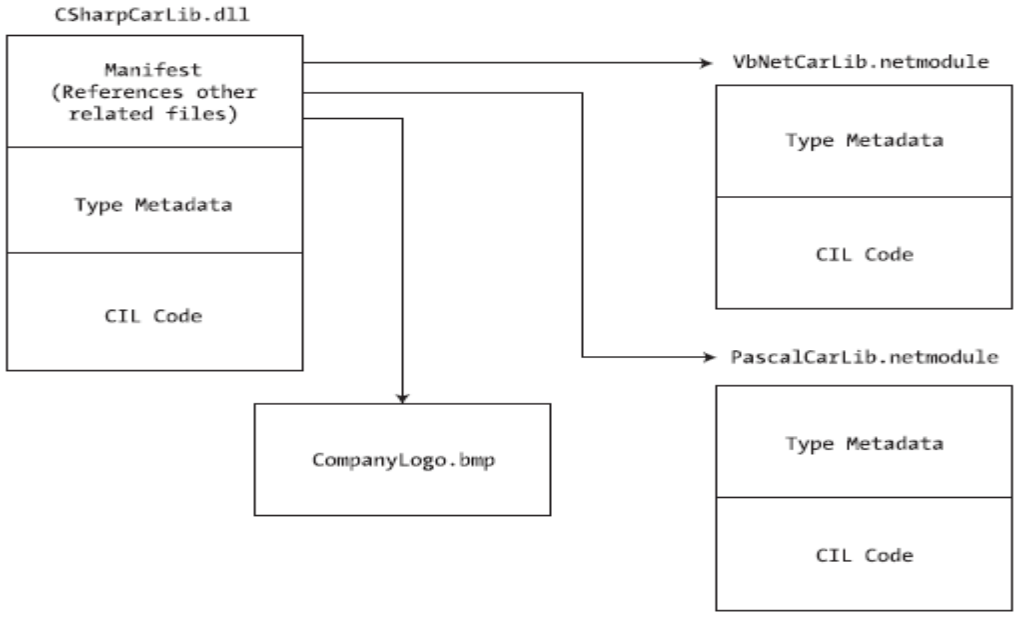
Answer
An assembly can be composed of multiple *modules*. A module is really nothing more than a generic term for a valid .NET binary file. In most situations, an assembly is in fact composed of a single module. In this case, there is a one-to-one correspondence between the (logical) assembly and the underlying (physical)

binary (hence the term *single-file assembly*).
Single-file assemblies contain all of the necessary elements (header information, CIL code, type metadata, manifest, and required resources) in a single *.exe or *.dll package. Following figure illustrates the composition of a single-file assembly.



A Single-file Assembly

A multifile assembly, on the other hand, is a set of .NET *.dlls that are deployed and versioned as a single logic unit. Formally speaking, one of these *.dlls is termed the *primary module* and contains the assembly-level manifest (as well as any necessary CIL code, metadata, header information, and optional resources). The manifest of the primary module records each of the related *.dll files it is dependent upon.
As a naming convention, the secondary modules in a multifile assembly take a *.netmodule file extension; however, this is not a requirement of the CLR. Secondary *.netmodules also contain CIL
code and type metadata, as well as a *module-level manifest*, which simply records the externally required assemblies of that specific module.
The major benefit of constructing multifile assemblies is that they provide a very efficient way to download content. For example, assume you have a machine that is referencing a remote multifile assembly composed of three modules, where the primary module is installed on the client. If the client requires a type within a secondary remote *.netmodule, the CLR will download the binary to the local machine on demand to a specific location termed the *download cache.*
Another benefit of multifile assemblies is that they enable modules to be authored using multiple .NET programming languages (which is very helpful in larger corporations, where individual departments tend to favor a specific .NET language). Once each of the individual modules has been compiled, the modules can be logically “connected” into a logical assembly using tools such as the assembly linker (al.exe).
In any case, do understand that the modules that compose a multifile assembly are *not* literally linked together into a single (larger) file. Rather, multifile assemblies are only logically related by information contained in the primary module’s manifest. Following figure illustrates a multifile assembly composed of three modules, each authored using a unique .NET programming language.



A Multifile Assembly: The primary module records secondary modules in the assembly manifest

Text Book

1. C# and the .NET Platform, Andrew Troelsen, Second Edition 2003, Dreamtech Press