Q.2 a. What features makes .NET Platform suitable for developing applications which integrates programs in different languages? (6)

Answer:

The .NET Framework is a rather radical and brute-force approach to making our lives easier. The solution proposed by .NET is "Change everything". The .NET Framework is a completely new model for building systems on the Windows family of operating systems, as well as on numerous non-Microsoft operating systems such as Mac OS X and various Unix/Linux distributions. Some core features provided by .NET to support developing applications which integrates programs in different languages are:

• *Full interoperability with existing code*: Existing COM binaries can commingle (i.e., interop) with newer .NET binaries and vice versa. Also, Platform Invocation Services (PInvoke) allows you to call C-based libraries (including the underlying API
of the operating system) from .NET code.

• *Complete and total language integration*: Unlike COM, .NET supports cross-language inheritance, cross-language exception handling, and cross-language debugging.

• *A common runtime engine shared by all .NET-aware languages*: One aspect of this engine is a well-defined set of types that each .NET-aware language "understands."

• *A base class library*: This library provides shelter from the complexities of raw API calls and offers a consistent object model used by all .NET-aware languages.

• *No more COM plumbing*: IClassFactory, IUnknown, IDispatch, IDL code, and the evil VARIANTcompliant data types (BSTR, SAFEARRAY, and so forth) have no place in a native .NET binary.

• *A truly simplified deployment model*: Under .NET, there is no need to register a binary unit into the system registry. Furthermore, .NET allows multiple versions of the same *.dll to exist in harmony on a single machine. In fact, the only
way .NET and COM types can interact with each other is using the interoperability layer.

b. Give a brief description of Common Language Runtime (CLR). Also, explain the workflow between source code, .NET compiler and .NET execution engine diagrammatically. (10)

Answer:

Common language *runtime* can be understood as a collection of external services that are required to execute a given compiled unit of code. For example, when developers make use of the Microsoft Foundation Classes (MFC) to create a new application, they are aware that their program requires the MFC runtime library (i.e., mfc42.dll). Other popular languages also have a corresponding runtime. VB6 programmers are also tied to a runtime module or two (e.g., msvbvm60.dll). Java developers are tied to the Java Virtual Machine (JVM) and so forth. The .NET platform offers yet another runtime system. The key difference between the .NET runtime and the various other runtimes is the fact that the .NET runtime provides a single well-defined runtime layer that is shared by *all* languages and platforms that are .NET-aware. The crux of the CLR is physically represented by a library named mscoree.dll. When an assembly is referenced for use, mscoree.dll is loaded automatically, which in turn loads the required assembly into memory. The runtime engine is responsible for a number of tasks. First and foremost, it is the entity in charge of resolving the location of an assembly and finding the requested type within the binary by reading the contained metadata. The CLR then lays out the type in memory, compiles the associated CIL into platform-specific instructions, performs any necessary security checks, and then executes the code in question.

In addition to loading your custom assemblies and creating your custom types, the CLR will also interact with the types contained within the .NET base class libraries when required. Although the entire base class library has been broken into a number of discrete assemblies, the key assembly is mscorlib.dll. mscorlib.dll contains a large number of core types that encapsulate a wide variety of common programming tasks as well as the core data types used by all .NET languages.

When you build .NET solutions, you automatically have access to this particular assembly.

Figure 1-3 illustrates the workflow that takes place between your source code (which is making use of base class library types), a given .NET compiler, and the .NET execution engine.

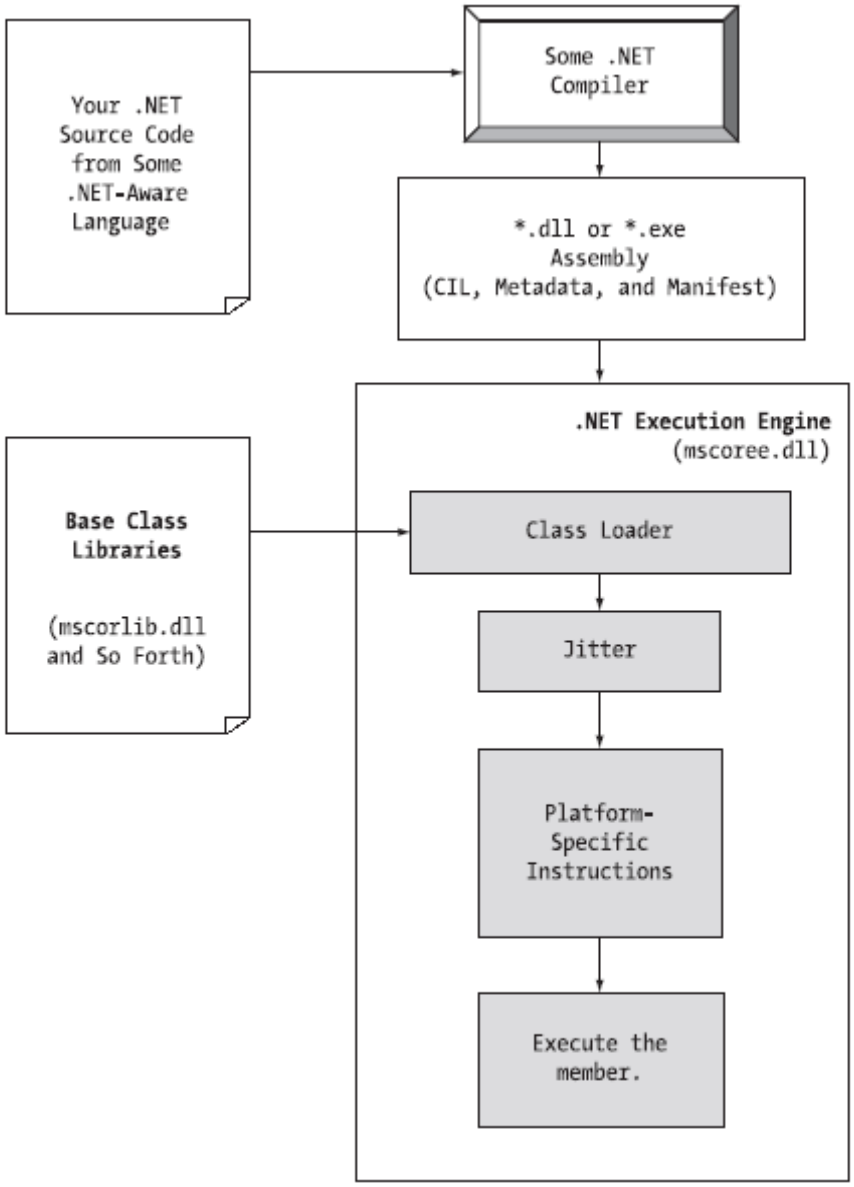

Figure 1-3. *mscoree.dll in action*

Q.3 a. Explain how C # application is build using CSC. exe compiler? Also, explain output centric options of C# compiler. (8)

Answer:

To build a simple single file assembly named TestApp.exe using the C# command-line compiler and Notepad. Open Notepad and enter the following:

```
// A simple C# application.
using System;
class TestApp
{
public static void Main()
{
```

```
Console.WriteLine("Testing! 1, 2, 3");
}
}
```

Then save the file in a convenient location (e.g., C:\CscExample) as TestApp.cs.

Output-centric Options of the C# Compiler

/out - This option is used to specify the name of the assembly to be created. By default, the assembly name is the same as the name of the initial input *.cs file (in the case of a *.dll) or the name of the type containing the program's Main() method (in the case of an *.exe).
/target:exe - This option builds an executable console application. This is the default file output type, and thus may be omitted when building this application type.
/target:library - This option builds a single-file *.dll assembly.
/target:module - This option builds a *module*.Modules are elements of multifile assemblies
/target:winexe - Although you are free to build Windows-based applications using the /target:exe flag, the /target:winexe flag prevents a console window from appearing in the background.

To compile TestApp.cs into a console application named TestApp.exe, change to the directory containing your source code file and enter the following command set (command-line flags must come before the name of the input files, not after):
csc /**target:exe** TestApp.cs or csc /**t:exe** TestApp.cs
Furthermore, given that the /t:exe flag is the default output used by the C# compiler, you could also compile TestApp.cs simply by typing
csc TestApp.cs

b. How can you debug a C# application using command line debugger (cordbg.exe)? Mention five command line flags also. (8)

Answer:

The .NET Framework does ship with a command-line debugger named cordbg.exe. This tool provides dozens of options that allow you to debug your assembly. You may view them by specifying the /? flag:
cordbg /?
Before you can debug your application using cordbg.exe, the first step is to generate debugging symbols for your current application by specifying the /debug flag of csc.exe. For example, to generate debugging data for TestApp.exe, enter the following command set:
csc @testapp.rsp **/debug**
This generates a new file named (in this case) testapp.pdb. Once you have generated a *.pdb file, open a session with cordbg.exe by specifying your .NET assembly as a command-line argument (the *.pdb file will be loaded automatically):
cordbg.exe testapp.exe
At this point, you are in debugging mode and may apply any number of cordbg.exe flags at the (cordbg) command prompt .
When you wish to quit debugging with cordbg.exe, simply type exit (or the shorthand ex).

cordbg.exe *Command-Line Flags*

b[reak] - Set or display current breakpoints.
del[ete] - Remove one or more breakpoints.
ex[it] - Exit the debugger.
g[o] - Continue debugging the current process until hitting next breakpoint.
o[ut] - Step out of the current function.
p[rint] - Print all loaded variables (local, arguments, etc.).
si - Step into the next line.
so - Step over the next line.

Q.4 a. What are various method parameter modifiers in C#? Explain each modifier with suitable function definition and call. (10)

Answer:

Methods (static and instance level) tend to take parameters passed in by the caller. However, unlike some programming languages, C# provides a set of parameter modifiers that control how arguments are sent into (and possibly returned from) a given method:

- (none)-If a parameter is not marked with a parameter modifier, it is assumed to be *passed by value,* meaning the called method receives a copy of the original data.
- out -Output parameters are assigned by the method being called (and therefore *passed by reference*). If the called method fails to assign output parameters, you are issued a compiler error.
- Params- This parameter modifier allows you to send in a variable number of identically typed arguments as a single logical parameter. A method can have only a single params modifier, and it must be the final parameter of the method.
- ref -The value is initially assigned by the caller, and may be *optionally* reassigned by the called method (as the data is also passed by reference). No compiler error is generated if the called method fails to assign a ref parameter.

The Default Parameter-Passing Behavior

The default manner in which a parameter is sent into a function is *by value*. Simply put, if you do not mark an argument with a parameter-centric modifier, a copy of the variable is passed into the function:

```
// Arguments are passed by value by default.
public static int Add(int x, int y)
{
int ans = x + y;
// Caller will not see these changes
// as you are modifying a copy of the
// original data.
x = 10000;
y = 88888;
return ans;
}
```

Here, the incoming integer parameters will be passed by value. Therefore, if you change the values of the parameters within the scope of the member, the caller is blissfully unaware, given that you are changing the values of copies of the caller's integer data types:

```
static void Main(string[] args)
{
int x = 9, y = 10;
Console.WriteLine("Before call: X: {0}, Y: {1}", x, y);
Console.WriteLine("Answer is: {0}", Add(x, y));
Console.WriteLine("After call: X: {0}, Y: {1}", x, y);
}
```

As you would hope, the values of x and y remain identical before and after the call to Add().

The out Modifier

For *output* parameters, methods that have been defined to take output parameters are under obligation to assign them to an appropriate value before exiting the method in question (if you fail to ensure this, you will receive compiler errors).

To illustrate, here is an alternative version of the Add() method that returns the sum of two integers using the C# out modifier

```
// Output parameters are allocated by the member.
public static void Add(int x, int y, out int ans)
{
ans = x + y;
```

```
}
```
Calling a method with output parameters also requires the use of the out modifier.
```
static void Main(string[] args)
{
// No need to assign local output variables.
int ans;
Add(90, 90, out ans);
Console.WriteLine("90 + 90 = {0} ", ans);
}
```
The previous example is intended to be illustrative in nature; you really have no reason to return the value of your summation using an output parameter. However, the C# out modifier does serve a very useful purpose: it allows the caller to obtain multiple return values from a single method invocation.

The ref Modifier

Reference parameters are necessary when you wish to allow a method to operate on (and usually change the values of) various data points declared in the caller's scope (such as a sorting or swapping routine). Note the distinction between
output and reference parameters:

• Output parameters do not need to be initialized before they passed to the method. The method must assign output parameters before exiting.

• Reference parameters *must* be initialized before they are passed to the method.

Let's check out the use of the ref keyword by way of a method that swaps two strings:
```
// Reference parameter.
public static void SwapStrings(ref string s1, ref string s2)
{
string tempStr = s1;
s1 = s2;
s2 = tempStr;
}
```
This method can be called as so:
```
static void Main(string[] args)
{
string s = "First string";
string s2 = "My other string";
Console.WriteLine("Before: {0}, {1} ", s, s2);
SwapStrings(ref s, ref s2);
Console.WriteLine("After: {0}, {1} ", s, s2);
}
```
Here, the caller has assigned an initial value to local string data (s and s2). Once the call to SwapStrings() returns, s now contains the value "My other string", while s2 reports the value "First string".

The params Modifier

The final parameter modifier is the params modifier, which allows you to create a method that may be sent to a set of identically typed arguments *as a single logical parameter*. Assume a method that returns the average of any number of doubles:
```
// Return average of 'some number' of doubles.
static double CalculateAverage(params double[] values)
{
double sum = 0;
for (int i = 0; i < values.Length; i++)
sum += values[i];
return (sum / values.Length);
}
```

This method has been defined to take a parameter array of doubles. What this method is in fact saying is, "Send me *any number of doubles* and I'll compute the average." Given this, you can call CalculateAverage() in any of the following ways (if you did not make use of the params modifier in the definition of CalculateAverage(), the first invocation of this method would result in a compiler error):

```
static void Main(string[] args)
{
// Pass in a comma-delimited list of doubles...
double average;
average = CalculateAverage(4.0, 3.2, 5.7);
Console.WriteLine("Average of 4.0, 3.2, 5.7 is: {0}",
average);
// ...or pass an array of doubles.
double[] data = { 4.0, 3.2, 5.7 };
average = CalculateAverage(data);
Console.WriteLine("Average of data is: {0}", average);
Console.ReadLine();
}
```

b. How System.Text.StringBuilder class differs from System.String class? Explain with suitable example. (6)

Answer:

The methods of System.String, *seem* to internally modify a string but in fact returns a modified *copy* of the original string. For example, when you call ToUpper() on a string object, you are not modifying the underlying buffer of an
existing string object, but receive a new string object in uppercase form:

```
static void Main(string[] args)
{
...
// Make changes to strFixed? Nope!
System.String strFixed = "This is how I began life";
Console.WriteLine(strFixed);
string upperVersion = strFixed.ToUpper();
Console.WriteLine(strFixed);
Console.WriteLine("{0}\n\n", upperVersion);
...
}
```

In a similar vein, when you assign an existing string object to a new value, you have actually allocated a *new* string in the process (the original string object will eventually be garbage collected).

A similar process occurs with string concatenation.

To help reduce the amount of string copying, the System.Text namespace defines a class named StringBuilder. Unlike
System.String, StringBuilder provides you direct access to the underlying buffer. Like System.String, StringBuilder provides numerous members that allow you to append, format, insert, and remove data from the object.

When you create a StringBuilder object, you may specify (via a constructor argument) the initial number of characters the object can contain. If you do not do so, the default capacity of a StringBuilder is 16. In either case, if you add more character data to a StringBuilder than it is able to hold, the buffer is resized on the fly.

Here is an example of working with this class type:

```
using System;
using System.Text; // StringBuilder lives here.
class StringApp
```

```
{
static void Main(string[] args)
{
StringBuilder myBuffer = new StringBuilder("My string data");
Console.WriteLine("Capacity of this StringBuilder: {0}",
myBuffer.Capacity);
myBuffer.Append(" contains some numerical data: ");
myBuffer.AppendFormat("{0}, {1}.", 44, 99);
Console.WriteLine("Capacity of this StringBuilder: {0}",
myBuffer.Capacity);
Console.WriteLine(myBuffer);
}
}
```

For most applications, the overhead associated with returning modified copies of character data will be negligible. However, if you are building a text-intensive application (such as a word processor program), you will most likely find that using System.Text.StringBuilder improves performance.

Q.5 a. Give the concept of abstraction in object oriented programming. Also, demonstrate a C# abstract class having properties and abstract methods.(10)

Answer:

Abstraction enables you to enforce a polymorphic trait on each descendent, leaving them to contend with the task of providing the details behind your abstract methods.

When a class has been defined as an abstract base class, it may define any number of *abstract members* (which is analogous to a C++ pure virtual function). Abstract methods can be used whenever you wish to define a method that *does not* supply a default implementation.

To understand the role of abstract methods and class: To prevent the direct creation of the Shape type, we can define it as an abstract class

```
namespace Shapes
{
public abstract class Shape
{
// Shapes can be assigned a friendly pet name.
protected string petName;
// Constructors.
public Shape(){ petName = "NoName"; }
public Shape(string s) { petName = s;}

// Draw() is virtual and may be overridden.
public abstract void Draw();

public string PetName
{
get { return petName;}
set { petName = value;}
}
}
```

To enforce that each child class redefines Draw() , you can simply establish Draw() as an abstract method of the Shape class, which by definition means you provide no default implementation whatsoever. Abstract methods can only be defined in abstract classes. If you attempt to do otherwise, you will be issued a compiler error:

You can now implement Draw() in your Circle class. If you do not, Circle is also assumed to be a noncreatable abstract type that must be adorned with the abstract keyword.

```
// If we did not implement the abstract Draw() method, Circle would also be considered abstract,
and could not be directly created!
public class Circle : Shape
{
public Circle(){ }
public Circle(string name): base(name) { }
// Now Circle must decide how to render itself.
public override void Draw()
{
Console.WriteLine("Drawing {0} the Circle", petName);
}
}
```

b. What are the advantages of using member hiding in inheritance? Illustrate with suitable example. (6)

Answer:

C# provides a facility that is the logical opposite of method overriding: member hiding. Formally speaking, if a derived class redeclares an identical member inherited from a base class, the derived class has hidden (or *shadowed*) the parent's member. In the real world, this possibility is the greatest when you are subclassing from a class you did not create yourselves For the sake of illustration, assume you receive a class named ThreeDCircle from a coworker (or classmate) that currently derives from System.Object:

```
public class ThreeDCircle
{
public void Draw()
{
Console.WriteLine("Drawing a 3D Circle");
}
}
```

The ThreeDCircle "is-a" Circle, so it can be derived from existing Circle type:

To address this issue, you can prefix the new keyword to the Draw() member of the ThreeDCircle type. Doing so explicitly states that the derived type's implemention is intentionally designed to hide the parent's version

```
// This class extends Circle and hides the inherited Draw() method.
public class ThreeDCircle : Circle
{
// Hide any Draw() implementation above me.
public new void Draw()
{
Console.WriteLine("Drawing a 3D Circle");
}
}
```

You can also apply the new keyword to any member type inherited from a base class (field, constant, static member, property, etc.).

Q.6 a. What is the purpose of garbage collector? Also, describe the role of finalize() method and the process of finalization. (8)

Answer:

When you are building your C# applications .NET memory management Allocates an object onto the managed heap using the new keyword and forget about it. Once "new-ed," the garbage collector will destroy the object when it is no longer needed. The garbage collector removes an object from the heap when it is *unreachable* by any part of your code base.

The supreme base class of .NET, System.Object, defines a virtual method named Finalize(). The default implementation of this method does nothing whatsoever:

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

When you override Finalize() for your custom classes, you establish a specific location to perform any necessary cleanup logic for your type. Given that this member is defined as protected, it is not possible to directly call an object's Finalize() method. Rather, the *garbage collector* will call an object's Finalize() method (if supported) before removing the object from memory.

Of course, a call to Finalize() will (eventually) occur during a "natural" garbage collection or when you programmatically force a collection via GC.Collect(). In addition, a type's finalizer method will automatically be called when the *application domain* hosting your application is unloaded from memory. When your AppDomain is unloaded from memory, the CLR automatically invokes finalizers for every finalizable object created during its lifetime.

Finalization Process:

The role of the Finalize() method is to ensure that a .NET object can clean up unmanaged resources when garbage collected. Thus, if you are building a type that does not make use of unmanaged entities, finalization is of little use for very simple reason that finalization takes time.

When you allocate an object onto the managed heap, the runtime automatically determines whether your object supports a custom Finalize() method. If so, the object is marked as *finalizable*, and a pointer to this object is stored on an internal queue named the *finalization queue*. The finalization queue is a table maintained by the garbage collector that points to each and every object that must be finalized before it is removed from the heap. When the garbage collector determines it is time to free an object from memory, it examines each entry on the finalization queue, and copies the object off the heap to yet another managed structure termed the *finalization reachable* table. At this point, a separate thread is spawned to invoke the Finalize() method for each object on the freachable table *at the next garbage collection.* Given this, it will take at very least *two* garbage collections to truly finalize an object. The bottom line is that while finalization of an object does ensure an object can clean up unmanaged resources, it is still nondeterministic in nature, and due to the extra behind-the-curtains processing, considerably slower.

b. What do you mean by an exception? What are different types to handle an exception in C#? Explain throwing an exception with the help of a custom exception. (8)

Answer:

Exceptions are typically regarded as runtime anomalies that are difficult, if not impossible, to account for while programming your application. Possible exceptions include attempting to connect to a database that no longer exists, opening a corrupted file, or contacting a machine that is currently offline. In each of these cases, the programmer (and end user) has little control over these "exceptional" circumstances.

Programming with structured exception handling involves the use of four interrelated entities:

- A class type that represents the details of the exception that occurred
- A member that *throws* an instance of the exception class to the caller
- A block of code on the caller's side that invokes the exception-prone member

• A block of code on the caller's side that will process (or *catch*) the exception should it occur
The C# programming language offers four keywords (try, catch, throw, and finally) that allow you to throw and handle exceptions. The type that represents the problem at hand is a class derived from System.Exception (or a descendent thereof).

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, we 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 .

```
// 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.
}
...
}
```

Q.7 a. Demonstrate resolving name clashes by using an explicit interface implementation. (8)

Answer:

Explicit interface implementation can also be very helpful whenever you are implementing a number of interfaces that happen to contain identical members. For example, assume you wish to create a class that implements all the following new interface types:

```
// Three interfaces each define identically named methods.
public interface IDraw
{
void Draw();
}
public interface IDrawToPrinter
{
void Draw();
}
```

If you wish to build a class named SuperImage that supports basic rendering (IDraw), 3D rendering (IDraw3D), as well as printing services (IDrawToPrinter), the only way to provide unique implementations for each method is to use explicit interface implementation:

```
// Not deriving from Shape, but still injecting a name clash.
public class SuperImage : IDraw, IDrawToPrinter, IDraw3D
{
void IDraw.Draw()
{ /* Basic drawing logic. */ }
void IDrawToPrinter.Draw()
{ /* Printer logic. */ }
void IDraw3D.Draw()
{ /* 3D rendering logic. */ }
}
```

b. Explain IComparable interface with the help of an example. (8)

Answer:

The System.IComparable interface specifies a behavior that allows an object to be sorted based on some specified key. Here is the formal definition:

```
// This interface allows an object to specify its relationship between other like objects.
public interface IComparable
{
int CompareTo(object o);
}
```

Let's assume Car class to maintain an internal ID number (represented by a simple integer named carID) that can be set via a constructor parameter and manipulated using a property named ID.

```
public class Car
{
...
private int carID;
public int ID
{
get { return carID; }
set { carID = value; }
}
public Car(string name, int currSp, int id)
{
currSpeed = currSp;
petName = name;
carID = id;
}
...
}
```

Object users might create an array of Car types as follows:

```
static void Main(string[] args)
{
// Make an array of Car types.
Car[] myAutos = new Car[5];
myAutos[0] = new Car("Rusty", 80, 1);
myAutos[1] = new Car("Mary", 40, 234);
myAutos[2] = new Car("Viper", 40, 34);
myAutos[3] = new Car("Mel", 40, 4);
myAutos[4] = new Car("Chucky", 40, 5);
}
```

The System.Array class defines a static method named Sort(). When you invoke this method on an array of intrinsic types (int, short, string, etc.), you are able to sort the items in the array in numerical/alphabetic order as these intrinsic data types implement IComparable. However, to sort an array of Car types atleast one object must implement IComparable." When you build custom types, you can implement IComparable to allow arrays of your types to be sorted. When you flesh out the details of CompareTo(), it will be up to you to decide what the baseline of the ordering operation will be. For the Car type, the internal carID seems to be the most logical candidate:

```
// The iteration of the Car can be ordered
// based on the CarID.
public class Car : IComparable
{
...
// IComparable implementation.
int IComparable.CompareTo(object obj)
{
Car temp = (Car)obj;
if(this.carID > temp.carID)
return 1;
if(this.carID < temp.carID)
return -1;
else
return 0;
```

```
}
}
```

As you can see, the logic behind CompareTo() is to test the incoming type against the current instance based on a specific point of data. The return value of CompareTo() is used to discover if this type is less than, greater than, or equal to the object it is being compared with.
The corresponding user code is as follows:

```
// Exercise the IComparable interface.
static void Main(string[] args)
{
// Make an array of Car types.
...
// Dump current array.
Console.WriteLine("Here is the unordered set of cars:");
foreach(Car c in myAutos)
Console.WriteLine("{0} {1}", c.ID, c.PetName);
// Now, sort them using IComparable!
Array.Sort(myAutos);
// Dump sorted array.
Console.WriteLine("Here is the ordered set of cars:");
foreach(Car c in myAutos)
Console.WriteLine("{0} {1}", c.ID, c.PetName);
Console.ReadLine();
}
```

Q.8 a. What are the advantages of implementing callbacks using delegates in .NET Framework? Write a delegate in C# to add two integers. (8)

Answer:

The Windows API makes frequent use of C-style function pointers to create entities termed *callback functions* or simply *callbacks*. Using callbacks, programmers were able to configure one function to report back to (call back) another function in the application. The problem with standard C-style callback functions is that they represent little more than a raw
address in memory. Ideally, callbacks could be configured to include additional type-safe information such as the number of (and types of) parameters and the return value (if any) of the method pointed to. Sadly, this is not the case in traditional callback functions, and, as you may suspect, can therefore be a frequent source of bugs, hard crashes, and other runtime disasters.
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. Every delegate in the .NET Framework is automatically endowed with the ability to call their methods
synchronously or *asynchronously*.

```
namespace Mydelegate
{
  public delegate int operation(int x, int y);
```

```
class Program
{
  // method passed as argument has similar signature as delegate
static int Add(int a, int b)
{
        Return a+b;
}
public static void Main ()
{
        // delegate instantiation
      operation obj=new operation(Program.Add);
  Console.WriteLine("result of adding  two numbers is {0}", obj(20,30));
}
}
}
```

b. Demonstrate how events are handled in C# with the help of a simple example. (8)

Answer:

Demonstrate how events are handled in C# with the help of a simple example. (8)

Ans: Defining an event is a two-step process. First, you need to define a delegate that contains the methods to be called when the event is fired. Next, you declare the events (using the C# event keyword) in terms of the related delegate. In a nutshell, defining a type that can send events entails the following pattern:

```
public class SenderOfEvents
{
public delegate retval AssociatedDelegate(args);
public event AssociatedDelegate NameOfEvent;
...
}
```

The events of the Car type will take the same name as the previous delegates (AboutToBlow and Exploded). The new delegate to which the events are associated will be called CarEventHandler. Here are the initial updates to the Car type:

```
public class Car
{
// This delegate works in conjunction with the
// Car's events.
public delegate void CarEventHandler(string msg);
// This car can send these events.
public event CarEventHandler Exploded;
public event CarEventHandler AboutToBlow;
...
}
```

Sending an event to the caller is as simple as specifying the event by name as well as any required parameters as defined by the associated delegate. To ensure that the caller has indeed registered with event, you will want to check the event against a null value before invoking the delegate's method set. These things being said, here is the new iteration of the Car's Accelerate() method:

```
public void Accelerate(int delta)
{
// If the car is dead, fire Exploded event.
if (carIsDead)
```

```
{
if (Exploded != null)
Exploded("Sorry, this car is dead...");
}
else
{
currSpeed += delta;
// Almost dead?
if (10 == maxSpeed - currSpeed
&& AboutToBlow != null)
{
AboutToBlow("Careful buddy! Gonna blow!");
}
// Still OK!
if (currSpeed >= maxSpeed)
carIsDead = true;
else
Console.WriteLine("->CurrSpeed = {0}", currSpeed);
}
}
```

With this, you have configured the car to send two custom events without the need to define custom registration functions.

Listening to Incoming Events

C# events also simplify the act of registering the caller-side event handlers. Rather than having to specify custom helper methods, the caller simply makes use of the += and -= operators directly.

Here is the refactored Main() method, using the C# event registration syntax:

```
class Program
{
static void Main(string[] args)
{
Console.WriteLine("***** Events *****");
Car c1 = new Car("SlugBug", 100, 10);
// Register event handlers.
c1.AboutToBlow += new Car.CarEventHandler(CarIsAlmostDoomed);
c1.AboutToBlow += new Car.CarEventHandler(CarAboutToBlow);
Car.CarEventHandler d = new Car.CarEventHandler(CarExploded);
c1.Exploded += d;
Console.WriteLine("\n***** Speeding up *****");
for (int i = 0; i < 6; i++)
c1.Accelerate(20);
// Remove CarExploded method
// from invocation list.
c1.Exploded -= d;
Console.WriteLine("\n***** Speeding up *****");
for (int i = 0; i < 6; i++)
c1.Accelerate(20);
Console.ReadLine();
}
public static void CarAboutToBlow(string msg)
{ Console.WriteLine(msg); }
public static void CarIsAlmostDoomed(string msg)
{ Console.WriteLine("Critical Message from Car: {0}", msg); }
```

```
public static void CarExploded(string msg)
{ Console.WriteLine(msg); }
}
```

Q.9 a. Describe single-file and multi-file assemblies. What are the benefits of constructing multi-file assemblies? (10)

Answer:

Technically, an assembly can be composed of multiple *modules(* 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.

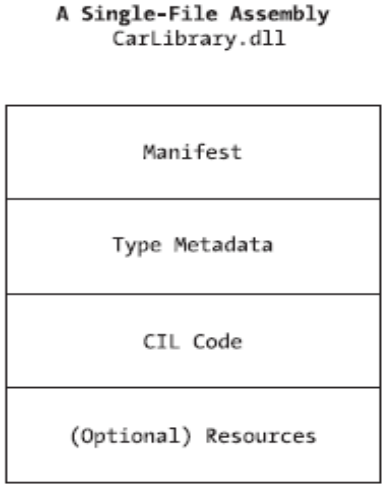

Figure 11-3. *A single-file assembly*

A multifile assembly, is a set of .NET *.dlls that are deployed and versioned as a single logic unit. 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. In any case, 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.

Figure illustrates a multifile assembly composed of three modules, each authored using a unique .NET programming language.

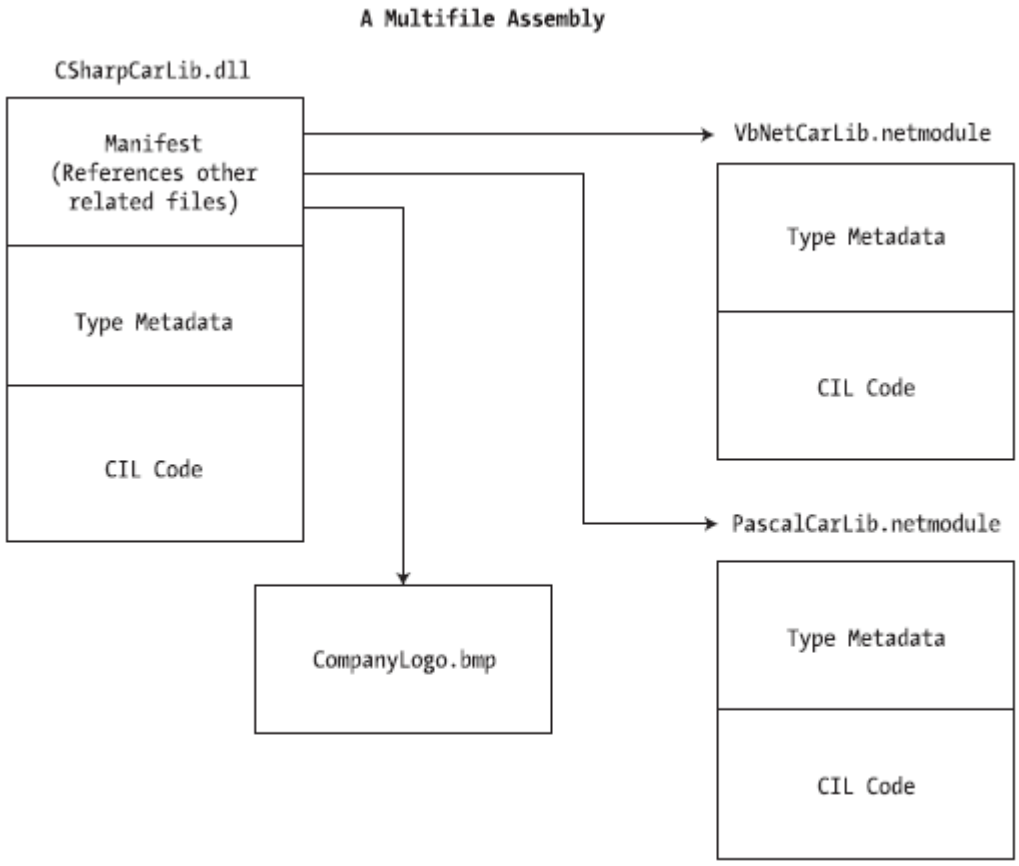

Figure 11-4. *The primary module records secondary modules in the assembly manifest*

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. 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).

b. Briefly explain the probing process of private assembly. (6)

Answer:

The .NET runtime resolves the location of a private assembly using a technique termed *probing*. Probing is the process of mapping an external assembly request to the location of the requested binary file. A request to load an assembly may be either *implicit* or *explicit*. An implicit load request occurs when the CLR consults the manifest in order to resolve the location of an assembly defined using the .assembly extern tokens:

// An implicit load request.
.assembly extern CarLibrary
{ ...}

An explicit load request occurs programmatically using the Load() or LoadFrom() method of the System.Reflection.Assembly class type, typically for the purposes of late binding and dynamic invocation of type members. **an example of an explicit load request** in the following code:

// An explicit load request.

Assembly asm = Assembly.Load("CarLibrary");
In either case, the CLR extracts the friendly name of the assembly and begins probing the client's application directory for a file named CarLibrary.dll. If this file cannot be located, an attempt is made to locate an executable assembly based on the same friendly name (CarLibrary.exe). If neither of these files can be located in the application directory, the runtime gives up and throws a FileNotFound exception at runtime.

TEXT BOOK

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