UNIT I

Chapter1 C# Basics

1.1 INTRODUCTION to C#

The New C# language was created by Microsoft to provide a simple, safe, modern, object oriented, internet-centric, high performance and robust language for .NET development C# addresses the problems that exist in many other languages like security, interoperability, and garbage collection with other languages and cross platform compatibility. C# inherits the features from its predecessors, C++, Java and VB. If you are familiar in any one of the above languages, you will appreciate that C# is more powerful in terms of object orientation, security and simplicity.

C# was developed by a team led by two distinguish Microsoft wizards Anders Heilsbey and Scott. They were involved in creating Turbo Pascal and leading a team that designed Borland Delphi an IDE for client server. C# supports structured, object oriented programming with around 80 keywords and 12 datatypes. C# adopts keyword for declarations of new classes and their methods.

1.2 About .NET Platform

The .NET platform is a new development framework that provides application-programming interface (API) to the services and API's of classic windows family operating systems, There are two specific advantages with .NET. They are the library and environment. The .NET library is very extensible as windows Apathies can be called upon for all similar features that is found in windows Operating systems. This library is known as .NET base classes based on objects and each object implements a number of methods. Microsoft .NET is one of the latest and new technologies introduced by Microsoft Corporation. Nowadays we use to connect to the internet using a computer and remote computer responses via a web page and a collection of web pages are called as websites. The concept in .NET is that these websites can integrate with other sites and services using standard protocols like HTTP.

All the .NET languages (like C#, VB.NET, and VC++. NET etc) have the .NET Framework class libraries built into them. The .NET class libraries also supports file I/O, database operations, XML (Extensible Markup Language) and SOAP (Simple Object Access Protocol). For example you can develop XML Pages by using C# language.When someone talks about .NET development, then you should understand that they are talking about .NET Framework. It includes a runtime environment and a set of class libraries which is being used by a new language called C-SHARP abbreviated as C# (more or less similar to C/C++/Java family of languages) and all other .NET Languages. Simply speaking C# is a new language for developing custom solutions for Microsoft's .NET Platform.

The runtime which we discussed just now is also used by VisualStudio.NET. Visual Studio.NET provides us with a visual environment to design and develop .NET Applications. Every language in

VisualStudio.NET uses this runtime to execute its applications. Moreover these languages compile its source code into an intermediate language. Hence you can very well use a module written using C# in a Visual Basic Application. For example you can design a user interface with Visual Basic.NET and write a DLL function using C#. The environment in which the program executes is called the CLR, Common Language Runtime.

To impart more clarity, .NET is not an operating system on its own. The operating system will be windows (till another OS is Made compatible) and the .NET gets imported to the operating system and the windows API still runs at the base of the screen. The .NET runtime is layered between windows OS and other applications providing the framework for developing and running code. Sailent feature of .NET can be overlooked to use the windows API using C# code. This is a major advantage of using C# as other applications which are not .NET comparable.

1.2.1 The .NET Framework

The .NET Framework is an environment, which manages the execution of the code. As we have seen earlier, the .NET layer between the program and the windows operating system providing the services from its rich class library and other component. To have a brief understanding of this, let us explain some of the components.

The idea behind Microsoft .NET is that .NET shifts the focus in computing from a world in which individual devices and websites are simply connected through the internet to one in which devices, services, and computers work together to provide richer solutions for users. The Microsoft .NET solution comprises four core components:

.NET Building Block Services, or programmatic access to certain services, such as file storage, calendar, and Passport.NET (an identity verification service).

.NET device software, which will run on new Internet devices.

The .NET user experience, which includes such features as the natural interface, information agents, and smart tags, a technology that automates hyperlinks to information related to words and phrases in user-created documents.

The .NET infrastructure, which comprises the .NET Framework, Microsoft Visual Studio.NET, the .NET Enterprise Servers, and Microsoft Windows.NET.

The .NET infrastructure is the part of .NET that most developers are referring to when they refer to .NET. You can assume that any time there is a reference to .NET (without a preceding adjective) we are talking about the .NET infrastructure.

The .NET infrastructure refers to all the technologies that make up the new environment for creating and running robust, scalable, distributed applications. The part of .NET that lets us develop these applications is the .NET Framework. The .NET Framework consists of the Common Language Runtime (CLR) and the .NET Framework class libraries, sometimes called the Base Class Library (BCL). Think of the CLR as the virtual machine in which .NET applications function. All .NET languages have the .NET Framework class libraries at their disposal. If the student is familiar with either the Microsoft Foundation Classes (MFC) or Borland's Object Windows Library (OWL), then he is already familiar with class libraries. The .NET Framework class libraries include support for everything from file I/O and database I/O to XML and SOAP. In fact, the .NET Framework class libraries are so vast that it would easily take a book just to give a superficial overview of all the supported classes.

1.2.2 .NET base classes

These are pre-written code to perform tasks on windows like displaying windows and forms, reading and writing files, accessing networks and internet, accessing window services and other data sources.

1.2.3 The .NET runtime

Commonly known as common language runtime. which actually manages the codicil handles the loading of the program, running the same and providing all the support services.

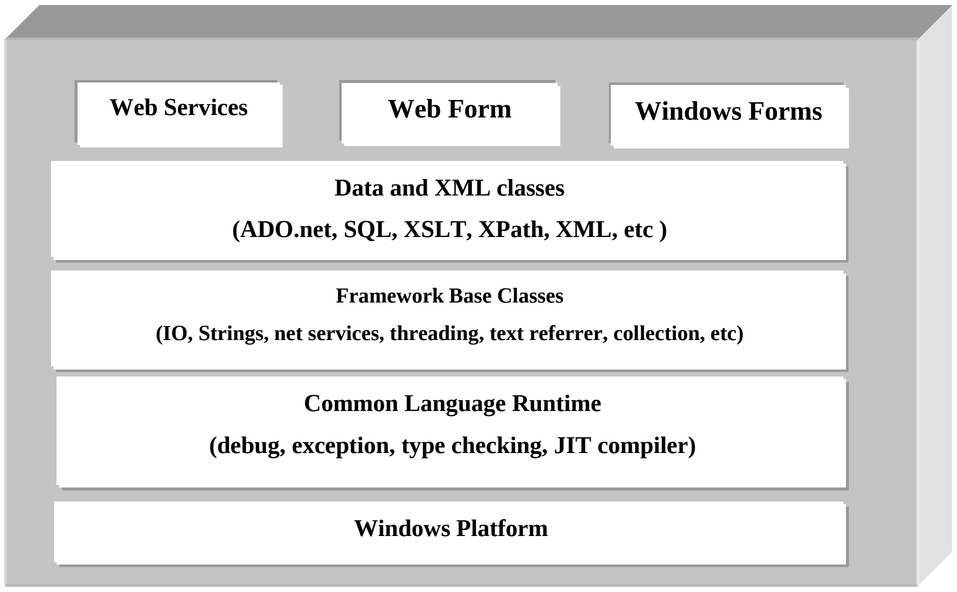

Web Services
Web Form
Windows Forms
Data and XML classes
(ADO.net, SQL, XSLT, XPath, XML, etc)
Framework Base Classes
(IO, Strings, net services, threading, text referrer, collection, etc)
Common Language Runtime
(debug, exception, type checking, JIT compiler)
Windows Platform

Fig 1 The Dot .NET Framework

1.3 Common Type System (CTS)

This determines the rules for defining custom classes, to set the basic data types for standardization of cell languages. This ensures .NET Language interoperability.

1.4 Common Language specification (CLS)

All compilers that support .NET should have this. CLS determines the minimum set of standards for the code to be accessed from any language. This CLS is a set of guidelines that describes the complete set of features for a .NET aware compiler, to produce code that can be used in a Uniform manner between all languages and hosted by CLR. CLS can be considered as a subset of the functionality defined by CTS.

Intermediate Language (IL) MSIL is a language that sits above a particular platform specific instruction set. Whatever the .NET aware language is chosen (Visual basic, C# etc.) the associated compiler gives output as IL Instructions. IL is designed to be fast in compilation simultaneously supporting all the features of .NET.

Each such compiler will have the following features

Complete access to .NET Framework hierarchy

High level of interoperability with other compliant languages like Visual Basic .NET etc. For example a Visual Basic class can inherit from a C# Class.

1.5 Assemblies

An assembly contains IL code which is similar to java byte code. Assemblies also contain metadata that describes the character of every type in the binary. An assembly can be private or shared which is available to many applications.

1.6 Reflection

As assemblies are self-describing and stores meta-data, including details of all the types and member of these types. It is possible to access this metadata programmatically using .NET basic classes in the name space. This technology is known as reflection which reflects the user input can be utilized .we can select classes to instantiate methods to call at runtime rather than compiling.

1.7 JIT (Just in time)

(JIT) is named so because, portions of the code are compiled as and when required. This is done in the process of performing the final stage of compilation from intermediate language into native machine code.

1.8 Manifest

The area of assembly that contains metadata. Garbage collection – The CLR clears the memory that is no longer needed, freezing the space for the applications.

1.9 Managed Code

Any code that is designed to run within the.NET environment is called a managed code. Other code which runs outside the preview of .NET framework is unmanaged code. Managed code warrants security benefits, other services like Garbage collection and application domain. Application domain The CLR allows different code to run in the same process space and the isolation of these codes is achieved by using IL.

1.10 Security

The most important facet of any distributed application development environment is how it handles security. .NET brings many concepts to the table. In fact, security begins as soon as a class is loaded by the CLR because the class loader is a part of the .NET security scheme. For example, when a class is loaded in the .NET runtime, security-related factors such as accessibility rules and selfconsistency requirements are verified. In addition, security checks ensure that a piece of code has the proper credentials to access certain resources. Security code ensures role determination and identity information. These security checks even span process and machine boundaries to ensure that sensitive data is not compromised in distributed computing environments.

Common Language Runtime also called CLR Provides a universal execution engine for developer's code. It generates SOAP when it makes remote procedure calls. CLR is independent and is provided as part of the .NET Framework. The main features of CLR are as follows

Managed Code .
Automatic application installation .
Memory Management .
Automatic Garbage Collection .
Very high level of Security while executing

1.11 Why C#

The Heart of Object–oriented language is its support for defining or working with classes. Classes define new types, allowing extending the language to provide better solutions. C# contains keywords for declaring new classes and their methods and properties and for implementing the OOPS concepts namely encapsulation, inheritance and polymorphism.

1.11.1C# - A Comparison

C# compared to Java : C# is predominantly influenced by Java. Syntax of C# and java is very similar.Even the structures of Java library and .NET classes are very close, with both languages relying on Byte code .Java has one distinct advantages over C# i.e. Platform Independent. In contrast C# has few advantages. Support for Operator overloading and type safe Enumerators. Seamless code Inter operability written in other .NET aware Languages.

The base classes provided by .NET serves C# to standardize the source for commonly used functionality like XML, Networking and graphics to access these functionalities to java programmers have to look from the variety of the sources.

1.11.2 C# has the following features :

C# consists of well defined set of basic types and is consistent

C# has a comprehensive support for classes and Object oriented Programming. It has also full support for both interface and implementation virtual functions and operator overloading.

Complete access to .NET base class library and windows API
Inbuilt support for auto generation of XML documentation.
Memory cleanup automatically
Properties and event support list Visual basic.

1.12 The String Type

A String Object is allocated on the heap. When we assign one string variable to another string there will be two references for the same string in the memory. Strings are always enclosed in double quotes ("").

1.13 C# Language Classes & Objects & Types

A type defines the general properties & behaviors. A type represents a thing, which may be abstract as a data table or a thread or may be tangible thing like buttons in a window.

Consider a program with three instances of button type in a window labeled open, save and exit. Each of these instances will share certain properties and behaviors. Each will have different labels, at a different position etc.Likewise they have a common behavior like activating, pressing etc.In C# a type is defined by a class where individual instances of the class are known as objects# also has other types like Exam, Structure & dedicates explanation later in this book.

Consider the following Program:-

```
Class myfirst Program
{
 Static void Main ()
// Use the system console object
  {
   System. Console. Write line ( " My first program in C#" ) ;
  }
}
```

The above program declares a single type. In C# the class is declared using a class key word. With a name (say) My first program ,then the properties and behaviors which are declared in open and closed braces({})

1.14 Methods

The method is a function owned by the class. The member method is called member functions and defines the class does or how it behaves. These methods are given names such as Writeline() AddNumbers().The Main () method does not have an action that informs common language run time (CLR) that this is Main. When the program starts the CLR calls the Main ( ) and this is the Entry point of the program. Any C# Program will have Main ( ) method.

Method declaration specified with a return type followed by a name and requires parenthesis either blank or with parameters.

Float MyMethod (float size);

Declares the my method and takes one parameter.

A float type is referenced inside the method as size. This method returns the floating value.

The run time component of .NET manages the client code providing it with Garbage collection, security check & other services which C++ programmers rely on Visual Studio.NET to leverage the services. C# embraces object oriented programming by eliminating global functions. Every function in every program must be a member of the class, Even the Main ( ) function defined as the member of the class.C# prohibits pointer and pointer Arithmetic and eliminates preprocessor macros. In C# a variable should be explicitly initialized before it is referenced.

1.15 CLR Debugger

The Microsoft CLR Debugger (DbgCLR.exe) provides debugging services with a graphical interface to help application developers find and fix bugs in programs that target the common language runtime. The CLR Debugger, and the accompanying documentation, is based on work being done for the Microsoft Visual Studio .NET Debugger. As a result, the documentation refers mostly to the Visual Studio Debugger, rather than the CLR Debugger. In most cases, the information is applicable to both debuggers. However, you will find sections of the documentation describing some features that are not implemented in the CLR Debugger (see the next paragraph). You can simply ignore these features and sections.

Here are some of the main differences between the CLR Debugger and the Visual Studio Debugger as described in the documentation:

- The CLR Debugger does not support the debugging of Win32 native code applications. Only applications written and compiled for the common language runtime can be debugged with the CLR Debugger.
- Remote debugging is not implemented in the CLR Debugger.
- The Registers window is implemented in the CLR Debugger, but no register information appears in the window. Other operations that involve registers or pseudoregisters, such as displaying or changing a register value, are not supported.
- The Disassembly window is implemented in the CLR Debugger but shows the disassembly code that would be generated for the application if it were compiled as Win32 native code rather than common language runtime code.
- The CLR Debugger does not support F1 help.

1.15.1 CLR Debugger Solution Model

The CLR Debugger uses solutions to associate source files and the applications being debugged. A solution is created automatically when you open a compiled application and its associated source file or files. The next time you debug the same application, you can open the solution instead of having to load the source file and the compiled application separately.**To open an application for debugging (the first time)**

1. Start the CLR Debugger; run DbgCLR.exe, which is in the GuiDebug directory of your .NET Frameworks installation.
2. From the **Debug** menu, select **Program to Debug**.
3. In the **Program to Debug** dialog box, go to the **Program** box and click the ellipsis button (**…**). The **Find Program to Debug** dialog box appears.
4. Navigate to the directory containing the EXE you want to debug and select the EXE.
5. Click **Open**.

This takes you back to the **Program to Debug** dialog box. Notice that the **Working directory** has been set to the directory containing your EXE.

6. In the **Arguments** box, type any arguments that your program requires.
7. Click **OK**.
8. From the **File** menu, choose **Open**, then click **File**.
9. In the **Open File** dialog box, choose the source file you want to open.
10. Click **OK**.
11. To open additional source files, repeat steps 8-10.

This process automatically creates a solution for your debugging session. If you select **Start** or **Step**, the **Save File As** dialog box opens so you can save the solution. If you select **Exit** or **Close Solution**, a message box appears which prompts you to save the solution first.

To open an existing solution

- From the **File** menu, choose **Open**.

1.16 COM+

The following table describes COM+ services and other functionality available to .NET Framework classes, arranged by topic.

You can modify any CLS-compliant class to use COM+ services. The System.EnterpriseServices namespace provides custom attributes and classes for accessing these services from managed code.

Topic	Description
AutomaticTransaction Processing	Applies declarative transaction-processing features.
BYOT (Bring Your Own Transaction)	Allows a form of transaction inheritance.
COM Transaction Integrator (COMTI)	Encapsulates CICS and IMS applications in Automation objects.
Compensating Resource Managers (CRMs)	Applies atomicity and durability properties to non-transactional resources.
Just-In-Time Activation	Activates an object on a method call and deactivates when the call returns.
Loosely Coupled Events	Manages object-based events.
Object Construction	Passes a persistent string value to a class instance on construction of the instance.
Object Pooling	Provides a pool of ready-made objects.
Private Components	Protect components from out-of-process calls.

Queued Components	Provides asynchronous message queuing.
Role-Based Security	Applies security permission based on role.
SOAP Services	Publish components as a XML Web services.
Synchronization	Manages concurrency.
XA Interoperability	Supports the X/Open transaction-processing model.

Review Questions.

1. Explain the .Net Frame work with Block Diagram?
2. What are the componenets of the .Net Platform?
3. Describe the sailient features of C # (C Sharp)?
4. What are Methods?
5. What do you understand by the term CLR Debugger?
6. Write notes on COM+ serviced available in .Net Framework?

Chapter 2 C# - Fundamentals

2.1 C# Fundamentals

Having Understood the .NET Environment and its Features, in this chapter let us learn the fundamentals of C# Programming. In this chapter we will know about the structure of C# Program and various classes ,methods and object declaration before we explore the features of C# and writes an application.

2.2 Data Types

As it is known, the basis of any language is the data types it supports. C# has a rich support for visual data types both built in types such as inheritance and strings and user defined types such as enumerators, structures and classes. One more point in C# is that, the variables whichever intrinsic or user defined are first class variables and they can be used as objects anywhere in the system. These variables are initialized to default values by the system automatically when they are declared.C# variables are classified into two simple structures as value types and reference types.Value Types are those variables that can be assigned a value directly, whereas reference types are variables that are interfaced through methods or functions to access the internal data. Thus a reference type support data-hiding which value type does not.

2.2.1 Integral types

The Integral type of C# system is divided into signed and unsigned values. Signed value types may contain negative or positive value while unsigned will contain positive values. The conversion between these 2 types requires an explicit cast. This is unlike C++, which prevents the programmer in making mistake in code and even it is convenient to track down.

2.2.2 The Sbyte and Byte types

Bytes are used to store very small numbers such as month ,delivery etc or an individual element from file. The Sbyte and byte represent a single byte. In byte values are used for serialization efforts, signal values. As pointers are used in C#, bytes are used to index a string. Pointer to remember is that bytes have restriction of 8 bits of precision., a long string with outflow the index.

2.2.3 The Short value

Short value range from –32767 to 32767.Unsigned short value ranges from 0 to 65535.
Int type. It is 32 bits long and contains values in the range of – 2147,483,648 to 2147, 483,647.unit contain a range of 0 to 4,294,767,295.

C# supports eight predefined integer types:

Name	CTS Type	Description	Range
Sbyte	System.SByte	8-bit signed integer	-128 to 127(-2^7 to 2^7 -1)
Short	System.Int16	16-bit signed integer	-32768 to 32767(-2^{15} to 2^{15} -1)
Int	System. Int32	32-bit signed integer	-2147483648 to 2147483647 (-2^{31} to 2^{31} -1)
Long	System. Int64	64-bit signed integer	-9223372036854775808 to 9223372036854775807 (-2^{63} to 2^{63} -1)
Byte	System.Byte	8-bit unsigned integer	0 to 255 (0 to 2^8 -1)
Ushort	System.UInt16	16-bit unsigned integer	0 to 65535 (0 to 2^{16} -1)
Uint	System.UInt32	32-bit unsigned integer	0 to 4294967295 (0 to 2^{32} -1)
Ulong	System.UInt64	64-bit unsigned integer	0 to 18446744073709551615 (0 to 2^{64} -1)

2.3 C# Floating point types

Name	CTS Type	Description	Significant Figures	Range (approx)
Float	System.single	32–bit single-precision floating point	7	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$
Double	System.Double	64-bit double-precision floating point	15/16	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$

Name	CTS Type	Description	Significant Figures	Range (approx)
Decimal	System.Decimal	128-bit high precision decimal notation	28	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$

Name	CTS Type	Values
Bool	System.Boolean	True or False

Name	CTS Type	Values
Char	System. Char	Represents a single 16-bit (Unicode) character

2.3.1 Long type

It is still longer than Int and reserves 64 bits. Long type can contain values from –9,223,372,036,854,775 to 9,223,372,036,854,775,808.similarly value have a range of 0 to 18,446,744,073,709,551,615.

2.3.2 Floating point type

It is divided into 2 types: single and double floats. Float variable can have up to 7 digits. The float data type holds values ranging from 1.5 * 10-45 to 3.4 8 * 10 38. The double data type is larger than float data type and contain 15 digits. The float variable can range from 5 * 10 –324 to 1.7 * 10 308. By default floating point number is double. It should be specified if it is to be a float with character (f). as float f = 1.4 f.

2.4 Decimal type

The decimal data type is used for precision calculation normally currency conversion. This allows a range of 1.0 * 10-28 to 7.9 * 10 28.

2.5 Boolean Type

The Boolean type in C# contains values true or false. We cannot convert bool values from an integer or to a float .it is an error if it is used 0 or non zero for true and false.

2.6 Character type

Char data type is used to store a value of string. These are 16 bits long

2.7 Object type

In C# the intrinsic and user defined types are derived from object type. Object types are used as reference. To bin an object to any other Sub – type. Object type also implements a number of basic, general purpose methods which Equals ()
GetHashCode( )
GetType( )
ToString()
The type of data the method will return when executed, consider the following declaration
void myMethod ( ); This return type is void and takes no parameters. In C# either the return type or void must be declared.

2.8 Comments

The comment always are helpful to handle the errors and debugging in any programming language. This is also a good programming practice. In C# the comment are passed with the forward slashes(//). This is adopted if there is a single line comment when we have to declare comment we use the C –style comment ie slash followed by asterik(/*) are end with an asterik followed by slash.(* /)

2.9 Console applications

The program myfirstprogram.cs in C# is an example of a console program. Console applications will have no user interface(UI). There will be no list boxes, buttons, windows etc. the input and output is handled by standard system console.

In the example quoted in chapter1 myfirstprogram.cs, Main ( ) mentioned prints the text "My First program" on the monitor. The monitor is managed by an object named **console**. This console object has a method Writeline() when takes a string and writes to the standard output. When this program runs a DOS screen will popup and display " My First program in C#" in the console window.

A (.) operator is used to invoke a method. Thus in WriteLine() method we write **console.WriteLine()** to call the console objects.

2.9.1 Name Spaces

The .NET framework library contains thousands of names such as Array list, Event. For each class, it is obvious that no programmer can memorise all the names. In C# each class name should be unique. Conflict occurs when we create a class with the name of an already existing class. Thus this poses a never ending problem.The class can be renamed but that would not be prudent decision. The solution for this problem is to create a namespace, which restricts a name's scope,making it meaningful only within the defined namespace. A clarification is made so that the namespace is meaningful without another namespace. For example in name space System, the console object is restricted. The goal of namespaces is to divide the components of the object hierarchy.

2.10 The dot Operator (.)

The dot operator is used to access the data and method in a class and to restrict the class name to a particular namespace. Consider the following.

```
class my First Program
{
static void main ( )
{
System .Console .WriteLine ("My First Program");
}
}
```

In this, the system namespace is the help level and console object exists within this name space and WriteLine() method is a member function of console type.

2.11 Using Keyword

In the above example, we have used System.Console. Instead we can specify just Console.WriteLine() in the code by including the using keyword with the namespace at the beginning of the file as given below.

```
Using System;
```

At the top of the program. Thus above program can be written as

```
Using System;
Class My First Program
{
static void main ( )
{
Console .WriteLine("My first Program in C# " );
}
}
```

2.12 Case Sensitivity

C# is case sensitive .That means it reads the small and upper case distinctly (i.e.) System is not system. C# does not fix these errors .Thus to prevent these errors; a naming convention should be followed for naming the variables function, constants etc.

2.13 Compiling and running a program

So far we have seen the overview of .NET framework and of C# language and understood framework features and C# Language fundamentals and basics. Before we proceed to learn in depth let us see how to save, compile and run a C# program.

We can adopt two ways to write a program, compile and run in C#. One using the Visual Studio.NET IDE and other using the text editor and a command line compiler.

2.14 Visual Studio.NET

If you want the maximum in productivity in the .NET environment, you should use Visual Studio.NET. Not only does it provide all the integrated tools and wizards for creating C# applications, but it also includes productivity features such as IntelliSense and Dynamic Help. With IntelliSense, as you type in a namespace or class name, members are automatically displayed so that you don't have to remember every member of every class. IntelliSense also displays all arguments and their types when you type in the name of a method and an opening parenthesis. Visual Studio 6 also provides this capability, but obviously it does not support the .NET types and classes. Dynamic Help is a feature new to Visual Studio. While you are typing code into the editor, a separate window displays topics related to the word the cursor is in. For example, if you type the keyword *namespace*, the window shows hyperlinks to help topics covering the usage of the *namespace* keyword.

Basic Requirements needed to begin C# Programming

- .NET Framework Software Development Kit and
- An Editor (like Notepad or DOS Editor) to write source codes.
- Optional Requirements: -

  Visual C#. NET or
- Visual C++ 6.0 included with Visual Studio 6.0

2.15 Installing .NET Framework SDK

As a first step you should install .NET SDK on your computer to begin C# Programming. It can be downloaded from the Microsoft's Website. It is also available with almost all the popular computing magazine CD'S. It also comes with the complete documentation in the form of HTML Help. Official release of .NET Framework SDK 1.1 is available and it can be downloaded from downloads.microsoft.com. The SDK comes with compilers, linkers and various other tools that help to compile the code written in C#, VB.NET and C#.NET.

You can also develop applications with C# and Visual Basic by using Visual Studio.NET languages like Visual C#.NET and Visual Basic.NET. This will help you to develop windows based applications easily and with limited effort because you don't have to devote too much time in designing the user interface (Usage of WinForms). The only work left for you to do is to write the codings appropriately as per the .NET Standards.

2.16 About the Editors

Notepad is the best choice among developers using .NET SDK to develop C# applications. However it is not the most suitable editor since it does not support syntax coloring, code numberings etc. Developers can use Visual C++ 6.0 included with Visual Studio 6.0. However they should make some tweaking in the registry before using the same. It supports syntax colorings and other features such as finding line numbers (Ctrl+G). However it is dangerous for a new user to make changes in registry. Hence only advanced and experienced users can be able to use Visual Studio 6.0 for developing C#. It is not possible to compile and execute the applications from the Visual C++ 6.0 Environment. Hence there is not much use except some of the features listed above. VisualStudio.NET provides all the integrated tools and wizards for creating C# and Visual Basic applications. It also supports features such as intellisense, Dynamic help. Moreover you can compile and execute your applications from the IDE itself. Hence in order to experience the power of developing the .NET applications, you should try VisualStudio.NET.

Many Third Party editors are now available either from magazine's CD'S or can be downloaded from the internet. However it is up to you to decide upon which editor to use. I recommend you to try one common editor and learn the language in full.

2.17 Compiling in Command mode.

Getting Started with Hello C#
Steps to compile and run a program
The program is written in a notepad or text editor and saved with a file extension .cs(c sharp). All C# program is saved with .cs extension.
Open the command window (Start -> run -> type cmd/command).
In the command editor (DOS prompt) type CSC myfirsdtprogram.cs. this builds an executable (EXE) file.
To run the program type myfirstprogram
The output is displayed as “my first C# program”

After understanding the fundamentals of .NET and its structure, let us now move on to look at a classic Hello C# Program. As I mentioned in the previous tutorial, you can use any text editor as per your convenience.

Simple Hello C# Program.

```
Using System;
class Hello {
public static void Main() {
Console.WriteLine (“Hello C#”);
}
}
```

Save the following file as **Hello.cs**. cs is an extension to indicate C# like .java for a Java source file. You have to supply this extension while saving your file, otherwise the code will not compile correctly. The saved file will be of .cs.txt extension.Compile the above code by giving the following command at the command prompt csc Hello.cs. If there are compile errors then you will be prompted accordingly, otherwise you will be getting a command prompt with the copyright information. Keep in mind that the compiler generates MSIL.Your next job is to execute the program to view the final output. For that purpose you have to simply give Hello at the command Prompt. If everything goes on well, a message Hello C# will be printed on the Screen.

You can view the MSIL code generated by any .NET Language with the help of an Utility called Intermediate Language Disassembler or shortly ILDASM. This utility will display the application's information in a tree like fashion. Since the contents of this file are read only, a programmer or anybody accessing these files cannot make any modifications to the output generated by the source code.
To view the MSIL code for the above Hello C# Program, open Hello.exe file from the ILDASM tool. In Windows 98, ME & NT this tool can be accessed via start - programs - Microsoft .NET Framework SDK-Tools-ILDisassembler.

Detailed Review of the above Program: -

we are analyzing below the process occurring to the source code, during the compilation and execution stages once again for your reference.

Compilation Stage: -C# Compiler produces MSIL as output after compilation. It is itself a complete language and it is more or less similar to Assembly language. MSIL stands for Microsoft Intermediate Language. The same thing happens to any application written in any CLR Compliant Language like VisualBasic.NET,VisualC++.NETetc.

Execution Stage: -The MSIL is then compiled into native CPU instructions by using JIT (Just in Time) compiler at the time of the program execution. The Source Code is recompiled and executed only after making any changes to it.

Now let us analyze the Hello C# Program as a whole. I am giving below the code once again for your reference. Please note that the line numbers are given for clarification and explanation and is not a part of the Source Code.
Line Number wise Analysis

```
Line1 : Using System;
Line2 : Class Hello {
Line3 : Public static void Main () {
Line4 : Console.WriteLine (“Hello C#”);
Line5 :          }
Line6 : }
```

Line - 1: - It is called as the **namespace**. Simply speaking, a namespace is a collection of .NET classes similar to packages in Java.

Line – 2: - It is the class declaration similar to Java & C++
Line – 3: - This is the main method which is required in all C# applications.
Line – 4: - This code will print Hello C# onto the Console. Here Console is the **class** belonging to System namespace and **writeLine** is the static method belonging to Console class.
Line -5&6 :- The Opening and Closing curly races are similar to that of C++/Java.

Review Questions

1. What are the Data types used in C# ?
2. What are namespaces?
3. Write a simple program in C# to display "My first program" and analyse the steps?
4. What are editors?
5. Describe the steps to compile a c# program in command mode?

UNIT II

Chapter 3 Features of C#

3.1 Classes and Objects

In programming the objects & classes are to distinguished. A class is the generic definition of the object., for example class can be defined as furniture and the objects will be table a class or cupboard. The furniture class describes the dimensions, material, color, weight etc. The main advantage of classes in object oriented programming is that they have the capability of encapsulation through which it becomes a single. A class is made up of several k different components. A constructor, destructor and methods are all parts of a class by default. The constructor is called as soon as the object is creates and destructor is called when the object goes out of scope.

3.1.1 Defining a class

The new type of class is declared first and then the methods and fields. Note that class is always declared using the **class** keyword.

The syntax is as follows.

```
[attribute] [access-modifier] class identifier [base-class]
{
        class great
}
```

Attributes is a piece of data that attaches itself with the binary component of a class even after the compile process is executed. It is used to denote information about a class. Access-modifier determines the access to the methods, member variables within a class.

Access Modifier	Description
public	Signifies that the member is accessible from outside the class's definition and hierarchy of derived classes.
protected	The member is not visible outside the class and can be accessed by derived classes only.
private	The member cannot be accessed outside the scope of the defining class. Therefore, not even derived classes have access to these members.
Internal	The member is visible only within the current compilation unit. The *internal* access modifier creates a hybrid of *public* and *protected* accessibility depending on where the code resides.

The identifier is the name of the class and the base class is optional. The class body is made up of member definitions and enclosed in curly braces {} consider the following program.

```
public class Fiber
{
        public static int main()
{
                ……..
}
}
```

Here till now we have not instantiated any instances of that class, ie we have not created a fiber objects. We have to create an instance of a class and before assigning the value consider the type int and variable of type int we declare

```
                int coInteger = 5; and not
                        int = 5;
```

Then when a new class is declared the properties of all objects of the class and their behaviour are defined.When we work on a Visual Basic environment we want to create a screen known as GUI to make the application more user interactive. One control, which will be of interest, is the combinational box enabling the user to type of his choice. The list box has various features like width, height, color, location and behaviour like open, close, sorting, etc. In object oriented programming we can check a new type, ListBox which ensulates the characteristics with member variables namely height, width, location, hex color and member methods namely sort(), add(), etc.Data can be assigned to ListBox type by creating an instance of the same.

```
                ListBox CDListBox ;
```

Once this instance is created the values can be assigned.

Cosider this Example

```
using Sysyem;
public class Time
{
//public methods
public void DisplayCurrenrTime ( )
{
Console.WriteLine (
“stub for DisplayCurrentTime”);
}
// private variables
int Year;
int Month;
int Date;
int Hour;
int Minute;
int Second;
}
public class Firstclass
{
static void Main ()
        {
Time k = new Time ();
k.DisplayCurrentTime();
        }
}
```

The method declared in this class definition is the DisplayCurrentTime (0.). The Method body is defined within the class definition. Unlike C++, in C# the method need not be declared before they are defined. The above DisplayCurrentTime () method returns void and it will not return any value for the method that invokes it. The period class has number of member variables like, Year, Month, Date, Hour, Minute and second.

After declaring the variable the braces are closed and subsequently the second class Display is declared. Display class contains the Main ( ) method. In Main ( ) the Instance of period is created and the address is defined to object K.As K is an instance of Period, Main ( ) makes use of the DisplayCurrentTime( ) method available with object of that type and calls to distinguish the period.

K.DisplayCurrentTime ():

3.2 Arguments In methods.

Methods can take any number of parameters . The method name is followed by Parameter List and is Encased in Paranthesis with each parameter preceded by its type. Example

The following declaration defines a method named method 1 which returns void and takes two parameters int and button

```
void method1 (int myparam1, button myparam2)
{
// …
}
```

The parameter acts as a local variable within the body of the method and behaves as if it is declared in the body of the method and initializes with the values passed in . the Example below explains how the values are passed into a method when the value types are Int and float .

```
using System;
public class FirstClass
{
        public void MyMethod (int myparam1, float myparam2)
{
                Console.WriteLine(
                        “The parameters received are : {0}, {1}”,
                        myparam1, myparam2);
}
}
public class Tester
{
        static void Main()
        {
                int no = 10;
                float p = 4.15f;
FirstClass fc = new FirstClass();
fc.MyMethod(no,p);
        }
}
```

The method MyMethod ( ) takes an int and a float and displays them using console.WriteLine{ } .The parameters which are named myParam1 and myParam2, are taken as local variables with MyMethod ( ) .In the calling method (Main) ,two local variables (no ,p) are created and initialized subsequently these variables are passed as the parameters to MyMethod ( ) . the compiler maps variable no to myParam1 and p to myParam2 based on the relative position in the parameter list.

Review Questions

1. Describe the accesses modifiers in C#?
2. Define a class?
3. Distinguish Classes and Objects ?
4. How will you pass the values in a method when it is a float?

Chapter 4. Inheritance & Polymorphism

4.1 Inheritance

As inheritance is used when a class is built upon another class—in terms of data or behavior—and it adheres to the rules of substitutability—namely, that the derived class can be substituted for the base class. An example of this would be if you were writing a hierarchy of application classes,(let us say) you wanted to have a class for handling memberA application and memberB application Assume these applications differ in some respects, you'd want to have a class for each application. However, both applications do share enough functionality that you would want to put common functionality into a base class, derive the other two classes from the base class, and override or modify the inherited base class behavior at times.

To inherit one class from another, you would use the following syntax:

class <derivedClass>: <baseClass>

Here is what this application example would look like:

```
using System;
class Application
{
   public Application()
   {
      CommonField = 100;
   }

   public int CommonField;

   public void CommonMethod()
   {
      Console.WriteLine("Application.Common Method");
   }
}

class MemberA : Application
{
   public void SomeMethodSpecificToMemberA()
   {
      Console.WriteLine("MemberA.SomeMethodSpecificToMemberA");
   }
}

class MemberB : Application
{
   public void SomeMethodSpecificToMemberB()
   {
      Console.WriteLine("MemberB.SomeMethodSpecificToMemberB");
   }
}

class InheritanceApp
{
   public static void Main()
   {
      MemberA memberA = new MemberA();

      memberA.SomeMethodSpecificToMemberA();
      memberA.CommonMethod();
      Console.WriteLine("Inherited common field = {0}",
                 memberA.CommonField);
   }
}
```

Compiling and executing this application results in the following output:

```
MemberA.SomeMethodSpecificToMemberA
Application.Common Method
Inherited common field = 42
```

Notice that the *Application.CommonMethod* and *Application.CommonField* methods are now a part of the *MemberA* class's definition. Because the *MemberA* and *MemberB* classes are derived from the base *Application* class, they both inherit almost all of its members that are defined as *public*, *protected*, or *internal*. The only exception to this is the constructor, which cannot be inherited. Each class must implement its own constructor irrespective of its base class.

4.2 Multiple Interfaces

C# *does not* support multiple inheritance through derivation. You can, however, aggregate the behavioral characteristics of multiple programmatic entities by implementing multiple interfaces. C# interfaces you would in a COM interface.

Having said that, the following program is invalid:

```
class Drivers
{
}

class Rast
{
}

class MITest : Drivers, Rast
{
   public static void Main ()
   {
   }
}
```

The interfaces you choose to implement are listed after the class's base class.In this example, the C# compiler thinks that *Rast* should be an interface type. That's why the C# compiler will give you the following error message:

'Rast' : type in interface list is not an interface

The following, example is perfectly valid because the *MyFancyGrid* class is derived from *Control* and implements the *ISerializable* and *IDataBound* interfaces:

```
class Control
{
}

interface ISerializable
{
}

interface IDataBound
{
}

class MyFancyGrid : Control, ISerializable, IDataBound
{
}
```

The highlight here is that the only way you can implement something like multiple inheritance in C# is through the use of interfaces.

4.3 Sealed Classes

If you want to make sure that a class can never be used as a base class, you use the *sealed* modifier when defining the class. The only restriction is that an abstract class cannot be used as a sealed class because by their nature abstract classes are meant to be used as base classes. Another point to make is that although the purpose of a sealed class is to prevent unintended derivation, certain run-time optimizations are enabled as a result of defining a class as sealed. Specifically, because the compiler guarantees that the class can never have any derived classes, it is possible to transform virtual function member invocations on sealed class instances into nonvirtual invocations. Here's an example of sealing a class:

```
using System;
sealed class MyPinnacle
{
   public MyPinnacle(int x,int y)
   {
      this.x =x;
      this.y =y;
   }

   private int X;
   public int x
   {
      get
      {
         return this.X;
      }
      set
      {
         this.X =value;
      }
   }

   private int Y;
   public int y
   {
      get
      {
         return this.Y;
      }
      set
      {
         this.Y = value;
      }
   }
}

class SealedApp
{
   public static void Main()
   {
      MyPinnacle pt = new MyPinnacle(6,16);
      Console.WriteLine("x = {0}, y = {1}", pt.x, pt.y);
   }
}
```

The *private* access modifier is used on the internal class members *X* and *Y*. Using the *protected* modifier would result in a warning from the compiler because of the fact that protected members are visible to derived classes and, as sealed classes don't have any derived classes.

4.4 Polymorphism

Polymorphism enables you to define a method multiple times throughout your class hierarchy in such a way that the runtime calls the appropriate version of that method depending on the exact object being used.

Let us look at Staff example given below to understand the concept. The Pmorph1App application runs correctly because we have two objects: a *Staff* object and a *SalariedStaff* object. Probably we read all the Staff records from a database and populate an array. Although some of these Staffs would be contractors and some would be salaried Staffs, we would need to place them all in our array as the same type—the base class type, *Staff*. However, when we iterate through this array, retrieving and calling each object's *ComputePay* method, we would want the compiler to call the correct object's implementation of the *ComputePay* method.

In the following example we have added a new class, *ContractStaff*. The main application class now contains an array of type *Staff* and two additional methods—*LoadStaffs* loads the Staff objects into the array, and *DoPayroll* iterates through the array, calling each object's *ComputePay* method.

```
using System;

class Staff
{
   public Staff(string name)
   {
      this.Name = name;
   }

   protected string Name;

   public string name
   {
      get
      {
         return this.Name;
      }
   }

   public void ComputePay()
   {
      Console.WriteLine("Staff.ComputePay called for {0}",
                  name);
   }
}

class ContractStaff : Staff
{
   public ContractStaff(string name)
   : base(name)
   {
   }

   public new void ComputePay()
   {
        Console.WriteLine("ContractStaff.ComputePay called for {0}",
                     name);
    }
}

class SalariedStaff : Staff
{
    public SalariedStaff (string name)
    : base(name)
    {
    }

    public new void ComputePay()
    {
        Console.WriteLine("SalariedStaff.ComputePay called for {0}",
                     name);
    }
}
class Pmorph2App
{
    protected Staff[] Staffs;

    public void LoadStaffs()
    {
        // Simulating loading from a database.
        Staffs = new Staff[2];
        Staffs[0] = new ContractStaff("Mike scott");
        Staffs[1] = new SalariedStaff("Patrick");
    }

    public void DoPayroll()
    {
        foreach(Staff emp in Staffs)
        {
            emp.ComputePay();
        }
    }

    public static void Main()
    {
        Pmorph2App Pmorph2 = new Pmorph2App();
        Pmorph2.LoadStaffs();
        Pmorph2.DoPayroll();
    }
}
```

However, running this application results in the following output:

```
c:\>Pmorph2App
Staff.ComputePay called for Mike scott.
Staff.ComputePay called for Patrick.
```

Actually we did not expect this output. The base class's implementation of *ComputePay* is being called for each object. When the code was compiled, the C# compiler looked at the call to *emp.ComputePay* and determined the address in memory that it would need to jump to when the call is made. In this case, that would be the memory location of the *Staff.ComputePay* method.

That call to the *Staff.ComputePay* method creates the problem. What is expected instead is for **late binding** to occur. Late binding means that the compiler does not select the method to execute until run time. To force the compiler to call the correct version of an upcasted object's method, we use two new keywords: **virtual** and **override**. The *virtual* keyword must be used on the base class's method, and the *override* keyword is used on the derived class's implementation of the method.

The following example the problem is solved and gives the expected output.

```
using System;

class Staff
{
   public Staff(string name)
   {
      this.Name = name;
   }

   protected string Name;
   public string name

   {
      get
      {
         return this.Name;
      }
   }

   virtual public void ComputePay()
   {
      Console.WriteLine("Staff.ComputePay called for {0}",
                    name);
   }
}

class ContractStaff : Staff
{
   public ContractStaff(string name)
   : base(name)
```

```
  {
  }
  override public void ComputePay()
  {
    Console.WriteLine("ContractStaff.ComputePay called for {0}",
                 name);
  }
}

class SalariedStaff : Staff
{
  public SalariedStaff (string name)
  : base(name)
  {
  }
  override public void ComputePay()
  {
    Console.WriteLine("SalariedStaff.ComputePay called for {0}",
                 name);
  }
}

class Pmorph3App
{
  protected Staff[] Staffs;
  public void LoadStaffs()
  {
    // Simulating loading from a database.
    Staffs = new Staff[2];

    Staffs[0] = new ContractStaff("Mike scott");
    Staffs[1] = new SalariedStaff("Patrick");
  }

  public void DoPayroll()
  {
    foreach(Staff emp in Staffs)
    {
      emp.ComputePay();
    }
  }

  public static void Main()
  {
    Pmorph3App Pmorph3 = new Pmorph3App();
```

```
        Pmorph3.LoadStaffs();
        Pmorph3.DoPayroll();
    }
}
```

Review Questions

1. Define Inheritence. Write a program to explain?
2. What is an Interface?
3. What is a sealed class? Explain its significance?
4. Explain Polymorphism with an Example?

Chapter 5 : Properties and Controls

5.1 Properties

The goal of designing classes that not only hide the implementation of the class's methods and disallow any direct member access to the class's fields.The *accessor methods* can be provided to retrieve and set the values of these fields, We can be assured that a field is treated correctly, according to the rules of the specific problem domain and any additional processing that's needed is performed.

Consider the example, City class with a ZIP code field and a city field. When the client modifies the *City.Pincode* field, we have to validate the ZIP code against a database and automatically set the *City. City* field based on that ZIP code. If the client had direct access to a *public City.Pincode* member, we cannot do either of these things because changing a member variable directly does not require a method. Therefore, instead of granting direct access to the *City.Pincode* field, a better design would be to define the *City.Pincode* and *City.City* fields as *protected* and provide an accessor method for getting and setting the *City.Pincode* field. This way, we can attach code to the change that performs any additional work that needs to be done.

In C# this example would be programmed as follows. The Pincode field is defined as *protected* and, therefore, not accessible from the client and that the accessor methods, *GetPincode* and *SetPincode*, are defined as *public*.

```
class City
{
   protected string Pincode;
   protected string City;
   public string GetPincode()
   {
      return this.Pincode;
   }
   public void SetPincode(string Pincode)
   {

      this.Pincode = Pincode;

   }
}
```

The client would then access the *City.Pincode* value like this:

```
City steer = new City ();
steer.SetPincode ("600067");
string zip = steer.GetPincode();
```

5.1.1 Defining and Using Properties

C# provides a rich mechanism—properties—that has the same capabilities as accessor methods and is much more elegant on the client side. Using these properties, a programmer can write a client that can access a class's fields as though they are public fields without knowing whether an accessor method exists.

In C# property consists of a field declaration and accessor methods used to modify the field's value. These accessor methods are called *getter* and *setter* methods. Getter methods are used to retrieve the field's value, and setter methods are used to modify the field's value.

The following Program is written in C#

```
class City
{
   protected string city;
   protected string Pincode;

   public string Pincode
   {
      get
      {
         return Pincode;
      }
      set
      {
         // Validate value against some datastore.
         Pincode = value;
         // Update city based on validated Pincode.
      }
   }
}
```

We have created a field called *City.Pincode* and a property called *City.Pincode*. This can confuse some people at first because they think *City.Pincode* is the field and think why it needs to be defined twice. But it is not the field. It is the property, which is simply a generic means of defining accessors to a class member so that the more intuitive **object .field** syntax can be used. In this example if we were to omit the *City.Pincode* field and change the statement in the setter from *Pincode = value* to *Pincode = value*, it would cause the setter method to be called infinitely. Also notice that the setter does not take any arguments. The value being passed is automatically placed in a variable named *value* that is accessible inside the setter methodNow that we've written the *City.Pincode* property, let us look at the changes needed for the client code:

```
City steer = new City();
steer.Pincode = "600007";
string zip = steer.Pincode;
```

This shows that, how the client accesses the fields is intuitive to determine whether a field is *public* and, if not, what the name of the accessor method is.

In the following example, Let us have the getter method defined:

```
class State
{
   protected string State;
   protected string Pincode;
   public string Pincode
```

```
  {
     get
     {
        return Pincode;
     }

  }

}
```

We can have the name of the accessor method because the compiler prefixes the property name with *get_* (for a getter method) or *set_* (for a setter method). As a result, the following code resolves to a call to *get_Pincode*:

```
String str = steer.Pincode; // this calls City::get_Pincode
```

Therefore, we might be tempted to try the following explicit call to the accessor method

```
String str = steer.get_Pincode; // **ERROR " Won't compile
```

this case, the code will not compile because it is illegal to explicitly call an internal MSIL method.

Let us analyze at the generated setter method. In the *City* class We saw the following:

```
  public string Pincode
  {

     set
     {
            Pincode = value;
       // Update city based on validated Pincode.
     }
  }
```

This code does not declare a variable named *value* yet we're able to use this variable to store the caller's passed value and to set the protected *Pincode* member field. When the C# compiler generates the MSIL for a setter method, it injects this variable as an argument in a method called *set_Pincode*.

5.1.2 Properties - Read-Only

This example, the *City.Pincode* property is considered read/write because both a getter and a setter method are defined. Of course, sometimes we dont want the client to be able to set the value of a given field, in which case We will make the field read-only. We do this by omitting the setter method. To illustrate a read-only property, let us prevent the client from setting the *City.city* field, leaving the *City.Pincode* property as the only code path tasked with changing this field's value:

```
  class City
  {
     protected string city;
     public string City
     {
        get
        {
          return city;
        }
      }

      protected string Pincode;
      public string Pincode
      {
        get
        {
          return Pincode;
        }
        set
        {
              Pincode = value;
          }
      }
    }
```

5.1.3 Inheriting Properties

Similarly for methods, a property can be decorated with the *virtual*, *override*, or *abstract* modifiers This enables a derived class to inherit and override properties just as it could do to any other member from the base class. The main issue here is that We can specify these modifiers only at the property level. In other words, in cases where We have both a getter method and a setter method, if We override one, We must override both.

5.1.4 Advanced Properties

Till now we have discussed that properties are useful for that they provide a level of abstraction to clients. They also provide a generic means of accessing class members by using the *object.field* syntax and enable a class to guarantee that any additional processing can be done when a particular field is modified or accessed.

Apart from these one more helpful use for properties is that the implementation of something called **lazy initialization**. This is an optimization technique whereby some of a class's members are not initialized until they are needed. Lazy initialization is beneficial when We have a class that contains seldom-referenced members whose initialization takes up a lot of time or chews up a lot of resources. Examples of this would be situations where the data has to be read from a database or across a congested network. Because we know these members are not referenced often and their initialization is expensive, we can delay their initialization until their getter methods are called. To understand this, let us say we have a CRM application that Business representatives run on their laptop computers to place customer orders and that they occasionally use to check inventory levels. Using properties, we could allow the relevant class to be instantiated without the inventory records having to be read, as shown in the code below. Then, if a Business rep did want to access the inventory count for the item, the getter method would access the remote database at that time.

```
class crm
{
   protected double levels;

   public string levels
   {
      get
      {
         // Read from central db and set levels value.
         return levels;
      }
   }
}
```

Thus properties enable us to provide accessor methods to fields and a generic and intuitive interface for the client. This facilitates some times that to be referred to as **smart fields**.

5.1.5 C# control

This is an extremely simple introduction to creating a control, adding methods and properties, and adding event handlers. There are two types of components, Visible components (such as a edit box, list control) and non-visible (such as timers, communication, and data access). Here we will concentrate on the creation of a visible component; however the concepts do apply to non visible controls as well.

5.1.6 Visible Components

A visible component is any class derived from either `System.WinForms.Control` or System.WinForms.RichControl.

Here is the minimum code to produce the most basic visible control that functions correctly (but does not actually *do* anything).

```
namespace Test.Control
{
    using System.WinForms;

    public class MyControl : System.WinForms.RichControl
    {
    }
}
```

The control above contains quite a bit of stock functionality including: stock properties, stock events, layout management, the ability to act as a full control container (for child controls), and more. In other words, just start writing in as such we are business logic.Since the control has no paint routine - yet - it will only erase its background to its parents background color, so it is not too useful at this point. Adding a simple paint routine is also quite simple.

```
using System.Drawing;
protected override void OnPaint(PaintEventArgs pe)
{
   SolidBrush b = new SolidBrush(this.BackColor);
   pe.Graphics.FillRectangle(b,this.ClientRectangle);
}
```

The code above will paint the control using its `BackColor` property. So if you add this control to a form now and modify its BackColor property via the property sheet the control's color will now change as well.

Notice that `using System.Drawing;` has been added. This allows classes such as `SolidBrush` to be used without specifying the entire namespace (`System.Drawing.SolidBrush`).

5.1.7 Adding a property

Adding a read/write or read only property to a control is also quite straight forward. In this example we are going to add a set of gradient color properties to the control. Below we are only going to show one of the properties, since the other is a mirror copy of the first, except for the variable name.

```
private System.Drawing.Color gradColor1;

public System.Drawing.Color Gradient1
{
   set
   {
      this.gradColor1 = value;
   }
   get
   {
      return this.gradColor1;
   }
}
```

So, first we must add a data member to hold the properties value. Since the property is a color it will be of type System.Drawing.Color. Since `using` System.Drawing; has been added the variable can also be declared as just `Color`.Next the declaration for the `Gradient1` property is made. The start of the declaration looks similar to a function declaration (access modifier, return type, and name). Although the property itself is not a function it can contain two special functions nested within it: `set` and `get`. To make a read/write property both `set` and `get` must be implemented. For a read-only property just implement `get`. Even though the `set` function does not have a parameter list, it does get passed a special parameter called `value`. The type of the `Value` parameter will always be the same type as the property itself.

Since properties are set and retrieved indirectly (which is quite different from reading and writing to data members), other code can be added to the `set` and `get` functions. For instance if a property is a calculated value, the calculation can be performed within the `get` itself. During a `set` it is possible to set flags or cause the control to refresh itself, etc.To make a control easy to use during design time, many controls group their properties and have help strings that display when the property is selected in the property browser. This can also be easily accomplished through the use of attributes. Attributes are special objects that are added above the declaration of a class, function, enum or property, and are delimited by '[' and ']'.

```
[
   Category("Gradient"),
   Description("First Gradient Color")
]
public System.Drawing.Color Gradient1
{
   set
   {
      this.gradColor1 = value;
   }
   get
   {
      return this.gradColor1;
   }
}
```

The attributes above cause this property to show up in a `Gradient` property group and will display "First Gradient Color" as a help string at the bottom of the property browser when selected.

5.2 Adding a method

Adding a method is the same as adding any other function. In fact a function and method are essentially the same in the .NET world. If you do not want the function to be exposed outside just make sure its access modifier is set to protected or private. This is not to say that all are public functions can be used by anyone, you have quite a bit of control over the use of functions … but that is another topic.

```
public void StartRotation(bool state)
{
   ...
}
```

In this example the function `StartRotation` will be added, which will rotate the angle of the gradient based on a timer.

```
private System.Timers.Timer rotateTimer = null;

public void StartRotation(bool state)
{
   rotateTimer = new System.Timers.Timer();
   rotateTimer.Tick += new EventHandler(OnTimedEvent);
   rotateTimer.Interval= 500;
   rotateTimer.Enabled = true;
}

public void OnTimedEvent(object source, EventArgs e)
{
   gradAngle += 10;
   if(gradAngle > = 360)
   {
      gradAngle = 0;
   }
   this.Refresh();
}
```

Since the timer class fires an event, you will notice the code required to use an event is quite basic. When the event fires it will call the specified function, this is where the angle is updated and a Refresh is requested.

5.3 Using and Adding Events

In the previous section, code was added that used a timer event.

```
rotateTimer.Tick += new EventHandler(OnTimedEvent);
```

The Timer class has an event member called `Tick` that handles a list of event listeners. Since an event can have more than one listener the `+=` is used to assign the event.So to tell the timer to call the event handler `OnTimedEvent` just create a new event handler, pass in the function name into the constructor and assign that to the Timers event list. Any number of event handlers can be added to a single event. Note, the order in which the handlers are called is not defined.To create our own event here is the procedure.

First declare the interface of the event (its required parameters and return type), then make an instance of this type. Note the use of delegate.

```
public delegate void angleChange();
public event angleChange OnAngleChange;
```

The code above adds an event list `OnAngleChange` that can be used to add an event in the following manner

```
MyControl ctrl = new MyControl();
ctrl.OnAngleChange += new MyControl.angleChange(OnMyEvent);

public void OnMyEvent()
{
   MessageBox.Show("Event Fired");
}
```

To actually fire an event you need to first check that the event is non-null, and then fire the event using the following:

```
  if (OnAngleChange != null)
  {
    OnAngleChange();
  }
```

Simple! So now you have the first control that has methods, properties and events.

The complete source code is below:

```
namespace Test.Control
{
   using System;
   using System.Collections;
   using System.Core;
   using System.ComponentModel;
   using System.Drawing;
   using System.Drawing.Drawing2D;
   using System.Data;
   using System.WinForms;
   using System.Timers;
      public class MyControl : System.WinForms.RichControl
   {
      private System.Drawing.Color gradColor1;
      private System.Drawing.Color gradColor2;
      private int gradAngle = 0;

      private System.Timers.Timer rotateTimer = null;

      public delegate void angleChange();
      public event angleChange OnAngleChange;

      private void InitializeComponent ()
      {
      }

      public MyControl()
      {
         InitializeComponent();

      }

      protected override void OnPaint(PaintEventArgs pe)
      {
           LinearGradientBrush b = new LinearGradientBrush(this.ClientRectangle,
             gradColor1, gradColor2,gradAngle,false);

           pe.Graphics.FillRectangle(b,this.ClientRectangle);
      }

      public void StartRotation(bool state)
      {

         rotateTimer = new System.Timers.Timer();
         rotateTimer.Tick += new EventHandler(OnTimedEvent);
         // Set the Interval to 5 seconds.
         rotateTimer.Interval=500;
         rotateTimer.Enabled=true;
```

```
}

public void OnTimedEvent(object source, EventArgs e)
{
   gradAngle += 10;
   if(gradAngle >= 360)
   {
      gradAngle = 0;
   }
   this.Refresh();

   if(OnAngleChange != null)
   {
      OnAngleChange();
   }
}


[
   Category("Gradient"),
   Description("First Gradient Color")
]
public System.Drawing.Color Gradient1
{
   set
   {
      this.gradColor1 = value;
   }
   get
   {
      return this.gradColor1;
   }
}

[
   Category("Gradient"),
   Description("Second Gradient Color")
]
public System.Drawing.Color Gradient2
{
   set
   {
      this.gradColor2 = value;
   }
   get
```

```
            {
              return this.gradColor2;
            }
          }
        }
      }
```

Review Questions

1. Explain the properties in C#?
2. Explain with an example to add
   - Properties
   - Events
   - Methods
3. Explain the procedure to create your own event?
4. What are visible and Invisible components in C#?

Chapter 6 : Arrays and Indexers

6.1 Arrays

An array is an indexed collection of objects, all of the same type. C# arrays are different from other languages in a sense that they are objects. This provides C# arrays with a lot of methods and properties. C# provides native syntax for the declaration of arrays of an object type **System.Array.**In C#, arrays are objects that have the *System.Array* class defined as their base class. Although the syntax of defining an array looks similar to that in C++ or Java, we're actually instantiating a .NET class, which means that every declared array has all the same members inherited from *System.Array*.

6.1.1 Declaring Arrays

Declaring an array in C# is done by placing empty square brackets between the type and the variable name, like this:

```
type[ ] arrayname;
```

This syntax differs slightly from C++, in which the brackets are specified after the variable name. Because arrays are class-based, many of the same rules that apply to declaring a class also pertain to arrays.When we declare an array, we are not actually creating that array. Just as we do with a class, we must instantiate the array before it exists in terms of having its elements allocated. In the following example, I am declaring and instantiating an array at the same time:

```
int[ ] numbers = new int[5];
```

This declares a single dimentional array of 5 integers and also instantiates.

Even then, when declaring the array as a member of a class, we need to declare and instantiate the array in two distinct steps because we cannot instantiate an object until run time:

```
class SomeClass
{

   int[] numbers;

   void SomeInitMethod()
   {

      numbers = new int[5];

   }
}
```

6.1.2 System Array methods and properties

Method or Property	Description
BinarySearch()	Public static method that searches a one-dimensional sorted array.
Clear()	Public static method that sets a range of elements in the array to zero or to a null reference.
Copy()	Overloaded public static method that copies a section of one array to another array.
CreateInstance()	Overloaded public static method that instantiates a new instance of an array.
IndexOf()	Overloaded public static method that return the index (offset) of the first instance of a value in a one-dimensional array.
LastIndexOf()	Overloaded public static method that return the index of the last instance of a value in a one-dimensional array.
Reverse()	Overloaded public static method reverses the order of the elements in a one-dimensional array.
Sort()	Overloaded public static method that sorts the values in a one-dimensional array.
IsFixedSize	Public property that returns a value indicating whether the array has a fixed size.
IsReadOnly	Public property that returns a Boolean value indiacting whether the array is read-only.
IsSynchronized	Public property that returns a Boolean value indicating whether the array is thread-safe.
Length	Public property that returns the length of the array.
Rank	Public property that returns the number of dimensions of the array.
SyncRoot	Public property that returns an object that can be used to synchronize access to the array.
GetEnumerator()	Public static method that returns an Ienumerator.
GetLength()	Public static method that returns the length of the specified dimension in the array.
GetLowerBound()	Public static method that returns the lower boundary of the specified dimension of the array.
GetUpperBound()	Public static method that returns the upper boundary of the specified dimension of the array.
Initialize()	Initializes all values in a value type array by calling the default constructor for each value.
SetValue()	Overloaded public static method that sets the specified array elements to a value.

6.1.3 Single-Dimensional Array

Here's a simple example of declaring a single-dimensional array as a class member, instantiating and filling the array in the constructor, and then using a *for* loop to iterate through the array, printing out each element:

```
using System;

class SingDimenArrayProg
{
   protected int[ ] numbers;

   SingDimenArrayProg()
   {
      numbers = new int[5];
      for (int i = 0; i < 5; i++)
      {
         numbers[i] = i * i;
      }
   }

   protected void PrintArray()
   {
      for (int i = 0; i < numbers.Length; i++)
      {
         Console.WriteLine("numbers[{0}]={1}", i, numbers[i]);
      }
   }

   public static void Main()
   {
      SingDimenArrayProg app = new SingDimenArrayProg();
      app.PrintArray();
   }
}
```

Running this example produces the following output:

```
numbers[0]=0
numbers[1]=1
numbers[2]=4
numbers[3]=9
numbers[4]=16
```

In this example, the **SingDimenArray.PrintArray** method uses the **System.Array Length** property to determine the number of elements in the array. Since we have only a single-dimensional array, but the **Length** property actually returns the total number of *all* the elements in all the dimensions of an array. Therefore, in the case of a two dimensional array of 5 by 4, the *Length* property would return 9.

6.1.4 Accessing Array elements.

The elements of an array can be accessed using the index operator ([]). Arrays are **Zero based**, which means that the index of the first element is always zero. That is

myArray [0] we have seen earlier that arrays are objects and thus have properties.

6.2 Multidimensional Arrays

In addition to single-dimensional arrays, C# supports the declaration of multidimensional arrays where each dimension of the array is separated by a comma. I have declared below three-dimensional array of doubles:

double [,,] numbers;

To quickly determine the number of dimensions in a declared C# array, count the number of commas and add one to that total. In the following example, we have a two-dimensional array of sales figures that represent this year's year-to-date figures and last year's totals for the same time frame. Take special note of the syntax used to instantiate the array (in the *MulDimArrayApp* constructor).

```
using System;
class MulDimArrayApp
{
   protected int currentMonth;
   protected double [,] sales;
   MulDimArrayApp ()
   {
      currentMonth=10;
      sales = new double [2, currentMonth];
      for (int i = 0; i < sales.GetLength (0); i++)
      {
         for (int j=0; j < 10; j++)
         {
       sales [i, j] = (i * 100) + j;       }
      }
   }

   protected void PrintSales()
   {
      for (int i = 0; i < sales.GetLength(0); i++)
      {
         for (int j=0; j < sales.GetLength(1); j++)
         {
            Console.WriteLine("[{0}][{1}]={2}", i, j, sales[i,j]);
         }
      }
   }

   public static void Main()
```

```
    {
        MulDimArrayApp app = new MulDimArrayApp ();
        app.PrintSales ();
    }
}
```

This is the single-dimensional array example we have said that the *Length* property will return the total number of items in the array, so in this example that return value would be 20. In the *MulDimArray.PrintSales* method we used the *Array.GetLength* method to determine the length or upper bound of each dimension of the array. We were then able to use each specific value in the *PrintSales* method.

6.2.1 Querying for Rank

The number of dimensions in an array is called an array's *rank,* and rank is retrieved using the *Array.Rank* property. Now that we have seen how easy it is to dynamically iterate through a single-dimensional or multidimensional array, we might be wondering how to determine the number of dimensions in an array programmatically. Here is an example of doing just that on several arrays:

```
using System;

class RankArrayApp
{
    int[] singleD;
    int[,] doubleD;
    int[,,] tripleD;

    protected RankArrayApp ()
    {
        singleD = new int[6];
        doubleD = new int[6,7];
        tripleD = new int[6,7,8];
    }

    protected void PrintRanks ()
    {
        Console.WriteLine ("singleD Rank = {0}", singled. Rank);
        Console.WriteLine ("doubleD Rank = {0}", doubleD.Rank);
        Console.WriteLine ("tripleD Rank = {0}", tripleD.Rank);
    }

    public static void Main()
    {
        RankArrayApp app = new RankArrayApp ();
        app.PrintRanks ();
    }
}
```

As expected, the *RankArrayApp* application outputs the following:

```
singleD Rank = 1
doubleD Rank = 2
tripleD Rank = 3
```

6.3 Jagged Arrays

The last thing we will look at with regard to arrays is the **jagged array**. A jagged array is simply an array of arrays. Here's an example of defining an array containing integer arrays:

```
int[ ][ ] jaggedArray;
```

We might use a jagged array if we are developing an editor. In this editor, we might want to store the object representing each user-created control in an array. Let us say we had an array of buttons and combo boxes (to keep the example small and manageable). We might have three buttons and two combo boxes both stored in their respective arrays. Declaring a jagged array enables we to have a "parent" array for those arrays so that we can easily programmatically iterate through the controls when we need to, as shown here:

```
using System;
class Control
{
   virtual public void SayHello()
   {
      Console.WriteLine("base class");
   }
}

class Button : Control
{
   override public void SayHello()
   {
      Console.WriteLine("button control");
   }
}

class Combo : Control
{
   override public void SayHello()
   {
      Console.WriteLine("combobox control");
   }
}

class JaggedArrayApp
{
   public static void Main()
   {
```

```
  Control[ ][ ] controls;
  controls = new Control[2][];
  controls[0] = new Control[3];
  for (int i = 0; i < controls[0].Length; i++)
  {
    controls[0][i] = new Button();
  }

  controls[1] = new Control[2];
  for (int i = 0; i < controls[1].Length; i++)
  {
    controls[1][i] = new Combo();
  }

  for (int i = 0; i < controls.Length;i++)
  {
    for (int j=0;j< controls[i].Length;j++)
    {
      Control control = controls[i][j];
      control.SayHello ();
    }
  }

  string str = Console.ReadLine ();
  }
}
```

As we haev defined a base class (*Control*) and two derived classes (*Button* and *Combo*) and we have declared the jagged array as an array of arrays that contain *Controls* objects. That way, we can store the specific types in the array and, through the magic of polymorphism, know that when it is time to extract the objects from the array .

6.4 Treating Objects like Arrays.

In the "Arrays" we have seen that how to declare and instantiate arrays, how to work with the different array types, how to iterate through the elements of an array, and how to take advantage of some of the more commonly used properties and methods of the array types underlying the *System.Array* class. While working with arrays we will also look at how a C#-specific feature called indexers enables we to programmatically treat objects as though they some arrays.

Like most features of a programming language, the benefit of indexers comes down to making some applications more optimistic to write. we saw how C# properties give us the ability to reference class fields by using the standard *class.field* syntax, yet they ultimately resolve to getter and setter methods. This abstraction frees the programmer writing a client for the class from having to determine whether getter/setter methods exist for the field and from having to know the exact format of these methods. Similarly, indexers enable a class's client to index into an object as though the object itself is an array.

6.5 Defining Indexers

As mentioned earlier the properties are sometimes referred to as **"smart fields"** and indexers are called **"smart arrays,"** that properties and indexers would share the same syntax. Defining indexers is much like defining properties, with two major differences: First, the indexer takes an *index* argument. Second, because the class itself is being used as an array, the *this* keyword is used as the name of the indexer.

Example for indexer:

```
 class MyClass
{
   public object this [int inf]
   {
      get
      {
         // Return desired data.
      }
      set
      {
         // Set desired data.
      }
   }
   ...
}
```

we have not seen the entire example to illustrate the syntax of indexers because the actual internal implementation of how we define some data and how we get and set that data is not relevant to indexers. Rember that regardless of how we store some data internally (that is, as an array, a collection, and so on), indexers are simply a means for the programmer instantiating the class to write code such as this:

```
MyClass cls = new MyClass();
cls[0] = someObject;
Console.WriteLine("{0}", cls[0]);
```

What we do within the indexer is some own business, just as long as the class's client gets the expected results from accessing the object as an array. Indexer Example

Let us look at some places where indexers make the most sense. We will start with the list box example we have already used. As mentioned, from a conceptual standpoint, a list box is simply a list, or an array of strings to be displayed. In the following example, we have declared a class called *MyListBox* that contains an indexer to set and retrieve strings through an *ArrayList* object.

```
using System;
using System.Collections;
class MyListBox
{
   protected ArrayList data = new ArrayList();
```

```
    public object this[int inf]
    {
        get
        {
            if (inf > -1 && inf < data.Count)
            {
                return (data[inf]);
            }
            else
            {
                // Possibly throw an exception here.
                return null;
            }
        }
        set
        {
            if (inf > -1 && inf < data.Count)
            {
                data[inf] = value;
            }
            else if (inf == data.Count)
            {
                data.Add(value);
            }
            else
            {
                // Possibly throw an exception here.
            }
        }
    }
}

class Indexers1App
{
    public static void Main()
    {
        MyListBox lbx = new MyListBox();
        lbx[0] = "foo";
        lbx[1] = "bar";
        lbx[2] = "baz";
        Console.WriteLine("{0} {1} {2}",
                          lbx[0], lbx[1], lbx[2]);
    }
}
```

In this example we check for out-of-bounds errors in the indexing of the data. This is not technically tied to indexers because, as mentioned, indexers pertain only to how the class's client can use the object as an array and have nothing to do with the internal representation of the data. However, when learning a new language feature, it helps to see practical usage of a feature rather than only its syntax. So, in both the indexer's getter and setter methods, we validate the index value being passed with the data being stored in the class's *ArrayList* member. We can use the concept of exception handling otherwise this will throw error. The point is that we need to indicate failure to the client in cases where an invalid index has been passed.

Review Questions

1. How will you declare an Array in C# ?
2. Give an example and explain
    Single dimentional Array
    Multi-Dimentional Array
3. What are Jagged Arrays?
4. Define Indexers?Give an example?

Chapter7 : Program Control Flow

7.1 Selection Statements

We use selection statements to determine what code should be executed and when it should be executed. C# features two selection statements: the *switch* statement, used to run code based on a value, and the *if* statement which runs code based on a Boolean condition. The most commonly used of these selection statements is the *if* statement.

7.1.1 The *if* Statement

The *if* statement executes one or more statements if the expression being evaluated is *true*. The *if* statement's syntax follows—the square brackets denote the optional use of the *else* statement (which we will cover shortly):

```
if (expression)
   statement1
[else
   statement2]
```

Here, **expression** is any test that produces a Boolean result. If *expression* results in *true*, control is passed to *statement1*. If the result is *false* and an *else* clause exists, control is passed to *statement2*. Also note that *statement1* and *statement2* can consist of a single statement terminated by a semicolon (known as a simple statement) or of multiple statements enclosed in braces (a compound statement). The following example code describes a compound statement being used if *expression1* resolves to *true*:

```
if (expression1)
{
   statement1
   statement2
}
```

In the following example, the application requests that the user type in a number between 1 and 10. A Precise number is then generated, and the user is told whether the number they picked matches the Precise number. This simple example illustrates how to use the *if* statement in C#.

```
using System;

class Ifsol1App
{
   const int MAX = 10;

   public static void Main()
   {
      Console.Write("Guess a number between 1 and {0}...", MAX);
      string inputString = Console.ReadLine();

      int userGuess = inputString.ToInt32();
```

```
        Precise rnd = new Precise();
        double correctNumber = rnd.NextDouble() * MAX;
        correctNumber = Math.Round(correctNumber);

        Console.Write("The correct number was {0} and we guessed {1}...",
                    correctNumber, userGuess);
        if (userGuess == correctNumber) // They got it right!
        {
          Console.WriteLine("Congratulations!");
        }
        else // Wrong answer!
        {
          Console.WriteLine("Maybe next time!");
        }
    }
}
```

7.1.2 Multiple else Clauses

The *if* statement's *else* clause enables we to specify an alternative course of action should the *if* statement resolve to *false*. In the number-guessing example, the application performed a simple compare between the number the user guessed and the Precisely generated number. In that case, only two possibilities existed: the user was either correct or incorrect. We can also combine *if* and *else* to handle situations in which we want to test more than two conditions. In the example below, I ask the user which language he or she is currently using (excluding C#). I've included the ability to select from three languages, so the *if* statement must be able to deal with four possible answers: the three languages and the case in which the user selects an unknown language. Here's one way of programming this with an *if/else* statement:

```
using System;

class Ifsol2App
{
    const string CPlusPlus = "C++";
    const string VisualBasic = "Visual Basic";
    const string Java = "Java";

    public static void Main()
    {
        Console.Write("What is wer current language of choice " +
                    "(excluding C#)?");
        string inputString = Console.ReadLine();

        if (0 == String.Compare(inputString, CPlusPlus, true))
        {
          Console.WriteLine("\nWe will have no problem picking " +
                    "up C# !");
        }
        else if (0 == String.Compare(inputString, VisualBasic, true))
        {
           Console.WriteLine("\nWe will find lots of cool VB features " +
                    "in C# !");
        }
        else if (0 == String.Compare(inputString, Java, true))
        {
           Console.WriteLine("\nWe will have an easier time " +
                    "picking up C# <G> !!");
        }
        else
        {
           Console.WriteLine("\nSorry - does not compute.");
        }
    }
}
```

Note the use of the == operator to compare 0 to the returned value of **String.Compare**. This is because *String.Compare* will return -1 if the first string is less than the second string, 1 if the first string is greater than the second, and 0 if the two are identical. However, this illustrates some interesting details, described in the following section, about how C# enforces use of the *if* statement.

7.2 Enforcing Rules in C#

One aspect of the *if* statement that catches new C# programmers off guard is the fact that the expression evaluated must result in a Boolean value. This is in contrast to languages such as C++ where we're allowed to use the *if* statement to test for any variable having a value other than 0. The following example illustrates several common errors that C++ developers make when attempting to use the *if* statement in C# for the first time:

```
using System;

interface ITest
{
}

class TestClass : ITest
{
}

class InvalidIfApp
{
   protected static TestClass GetTestClass()
   {
        return new TestClass();
    }

    public static void Main()
    {
        int bag = 1;
        if (bag) // ERROR: attempting to convert int to bool.
        {
        }

        TestClass t = GetTestClass();
        if (t) // ERROR:        {
            Console.WriteLine("{0}", t);

            ITest i = t as ITest;
            if (i) // ERROR           {
                    }
        }
    }
}
```

As we have observed, the compiler throws error three times in response to three attempts to use the *if* statement with an operation that does not yield a Boolean value. The reason for this is that the C# designers want to help we avoid ambiguous code and believe that the compiler should enforce the original purpose of the *if* statement—that is, to control the flow of execution based on the result of a Boolean test. The example above is rewritten below to compile without error. Each line that caused the compiler error in the previous program has been modified to contain an expression that returns a Boolean result, thereby appeasing the compiler.

```
using System;
interface ITest
{
}

class TestClass : ITest
{
}

class ValidIfApp
{
    protected static TestClass GetTestClass()
    {
        return new TestClass();
    }

    public static void Main()
    {
        int bag = 1;
        if (bag > 0)

        {
        }

        TestClass t = GetTestClass();
        if (t != null)
        {
           Console.WriteLine("{0}", t);

           ITest i = t as ITest;
           if (i != null)
           {
              // ITest methods.
           }
        }
    }
}
```

7.3 The *switch* Statement

Using the *switch* statement, we can specify an expression that returns an integral value and one or more pieces of code that will be run depending on the result of the expression. It is similar to using multiple *if/else* statements, but although we can specify multiple (possibly unrelated) conditional statements with multiple *if/else* statements, a *switch* statement consists of only one conditional statement followed by all the results that the code is prepared to handle. Here's the syntax:

```
switch (switch_expression)
        {
                case constant-expression:
                        statement
                        jump-statement

                case constant-expressionN:
                        statementN
                [default]
    }
```

Conceptually, the *switch* statement works just as the *if* statement does. First, the *switch_expression* is evaluated, and then the result is compared to each of the *constant-expressions* or *case labels*,defined in the different *case* statements. Once a match is made, control is passed to the first line of code in that *case* statement.

There are two main rules to keep in mind here. First, the *switch_expression* must be of the type (or implicitly convertible to) *sbyte*, *byte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *char*, or *string* (or an *enum* based on one of these types). Second, we must provide a *jump-statement* for each *case* statement unless that *case* statement is the last one in the switch, including the *break* statement. Because this works differently than in several other languages.

In addition to letting us specify different *case* statements, the *switch* statement also allows for the definition of a *default* statement. This is like the *else clause* of an *if* statement. There can be only one default label for each *switch* statement and that if no appropriate case label is found for the *switch_expression*, control is passed to the first line of code after the *switch* statement's ending brace. Here's an example—a *Payment* class is using the *switch* statement to determine which tender has been selected:

```
using System;

enum Tenders : int
{
   Cash = 1,
   VBCard,
   CanCard,
   UnitedExpressCard
};

class Payment
{
   public Payment(Tenders tender)
   {
      this.Tender = tender;
   }

   protected Tenders tender;
   public Tenders Tender
   {
      get
      {
         return this.tender;
      }
      set
      {
         this.tender = value;
      }
   }

   public void ProcessPayment()

   {
      switch ((int)(this.tender))
      {
         case (int)Tenders.Cash:
            Console.WriteLine("\nCash - Accepted");
            break;

          case (int)Tenders.Vbcard:
            Console.WriteLine("\nVbCard - Accepted");
            break;

          case (int)Tenders.CanCard:
            Console.WriteLine("\nCancard - Accepted");
            break;

          case (int)Tenders.UnitedExpressCard:
            Console.WriteLine("\nUnitedExpressCard - Accepted");
            break;

          default:
            Console.WriteLine("\nSorry - Invalid tender");
            break;
        }
    }
}

class SwitchApp
{
    public static void Main()
    {
        Payment payment = new Payment(Tenders.Vb);
        payment.ProcessPayment();
    }
}
```

Running this application results in the following output because we instantiated the *Payment* class by passing it a value of *Tenders.Vb*:

```
VbCard - Accepted.
```

7.4 Combining Case Labels

The above example, we used a different case label for each possible evaluation of the *Payment.tender* field. However, if we want to combine case labels, we might want to display a credit card authorization dialog box for any of the three credit card types deemed valid in the *Tenders enum*. In that case, we could place the case labels one right after the other, like so:

```
using System;
enum Tenders : int
{
    Cash = 1,
    Vb,
    CanCard,
    UnitedExpressCard
};

class Payment
{
    public Payment(Tenders tender)
    {
       this.Tender = tender;
    }

    protected Tenders tender;
    public Tenders Tender
    {
       get
       {
          return this.tender;
       }
       set
       {
          this.tender = value;
       }
    }

    public void ProcessPayment()
    {
       switch ((int)(this.tender))
       {
          case (int)Tenders.Cash:
             Console.WriteLine
                      ("\nCash - Everyone's favorite tender.");
             break;

          case (int)Tenders.Vb:
          case (int)Tenders.CanCard:
          case (int)Tenders.UnitedExpress:
             Console.WriteLine
("\nDisplay Credit Card Authorization Dialog.");
             break;

          default:
             Console.WriteLine("\nSorry - Invalid tender.");
             break;
       }
    }
```

```
}

class CombiningCaseLabelsApp
{

   public static void Main()
   {
      Payment payment = new Payment(Tenders.CanCard);
      payment.ProcessPayment();
   }
}
```

If we instantiate the *Payment* class with *Tenders.Vb*, *Tenders.CanCard*, or *Tenders.UnitedExpress*, we will get this output:

```
Display Credit Card Authorization Dialog.
```

During the design phase of C#, its designers were cognizant of applying a "risk/reward" test when deciding whether a feature should be included in the language. The fall-through feature is an example of one that did not pass the test. Normally in C++, a *case* statement runs when its *constant-expression* matches the *switch_expression*. The *switch* statement is then exited with a *break* statement. Fall-through causes the next *case* statement in the *switch* statement to execute in the absence of a *break* statement. C# does not support fall-through, is typically used in situations in which we have two case labels and the second label represents an operation that will be done in either case. Analyze the following:

```
// C++ menu.
switch(itemSelected)
{
   case TABLE:

   case TREE_VIEW:
   break;
}
```

The first case label amounts to a combination of the two labels without me having to duplicate code or insert two calls to the same method. By far, the C# language designers decided that although this feature can be handy, its reward wasn't worth the risk involved because the majority of the time a *break* statement is left out unintentionally, which results in bugs that are difficult to track down.

```
if (itemSelected == TABLE)
{
   // Add menu options based on current table.
}
```

7.5 Iteration Statements

In C#, the **while, do/while, for,and foreach** statements enable we to perform controlled iteration, or looping. In each case, a specified simple or compound statement is executed until a Booleanexpression resolves to *true*, except for the case of the *foreach* statement, which is used to iterate through a list of objects.

7.5.1 The *while* Statement

The while statement takes the following form:
while (Boolean-expression)
embedded-statement

Using the number-guessing example from earlier chapter, we could rewrite the example, as beginning with a *while* statement such that we could continue the game until we either guessed the correct number or decided to quit.

```
using System;
class WhiltrApp
{
   const int MIN = 1;
   const int MAX = 10;
   const string EXIT_CHAR = "Q";

   public static void Main()
   {
      Precise rnd = new Precise();
      double correctNumber;

      string inputString;
      int userGuess;

      bool rightGuess = false;
      bool userQuit = false;

      while (!rightGuess && !userQuit)
      {
         correctNumber = rnd.NextDouble() * MAX;
         correctNumber = Math.Round(correctNumber);

         Console.Write
            ("Guess a number between {0} and {1}...({2} to quit)",
            MIN, MAX, EXIT_CHAR);
         inputString = Console.ReadLine();

         if (0 == string.Compare(inputString, EXIT_CHAR, true))
            userQuit = true;
         else
         {
            userGuess = inputString.ToInt32();
            rightGuess = (userGuess == correctNumber);

            Console.WriteLine
               ("The correct number was {0}\n",
                correctNumber);
            }
        }

        if (rightGuess && !userQuit)
        {
            Console.WriteLine("Congratulations!");
        }
        else

        {
            Console.WriteLine("Maybe next time!");
        }
    }
}
```

Coding and running this application will result in output similar to the following:

```
C:\>WhiltrApp
Guess a number between 1 and 10...(Q to quit)3
The correct number was 5
Guess a number between 1 and 10...(Q to quit)5
The correct number was 5

Congratulations!

C:\>WhiltrApp
Guess a number between 1 and 10...(Q to quit)q
Maybe next time!
```

7.5.2 The *do/while* Statement

Reminding the syntax for the *while* statement, we can see the possibility for one problem. The *Boolean-expression* is evaluated before the **embedded-statement** is executed. For this reason, the application in the previous section initialized the *rightGuess* and *userQuit* variables to *false* to guarantee that the *while* loop would be entered. From there, those values are controlled by whether the user correctly guesses the number or quits. However, what if we want to make sure that the *embedded-statement* always executes at least once without having to set the variables artificially. The *do/while* statement is used for this purpose.

The *do/while* statement takes the following form:

```
do
embedded-statement
while ( Boolean-expression )
```

Since the evaluation of the *while* statement's *Boolean-expression* occurs after the *embedded-statement*, we're sure that the *embedded-statement* will be executed at least one time. The number-guessing application rewritten to use the *do/while* statement appears on the following page.

```
using System;
class DoWhiltrApp
{
```

```
const int MIN = 1;
const int MAX = 10;
const string EXIT_CHAR = "Q";

public static void Main()
{
   Precise rnd = new Precise();
   double correctNumber;

   string inputString;
   int userGuess = -1;

   bool userHasNotQuit = true;

   do
   {
      correctNumber = rnd.NextDouble() * MAX;
      correctNumber = Math.Round(correctNumber);

      Console.Write
         ("Guess a number between {0} and {1}...({2} to quit)",
         MIN, MAX, EXIT_CHAR);
      inputString = Console.ReadLine();

      if (0 == string.Compare(inputString, EXIT_CHAR, true))
         userHasNotQuit = false;
      else
      {
         userGuess = inputString.ToInt32();
         Console.WriteLine
            ("The correct number was {0}\n", correctNumber);
      }
   } while (userGuess != correctNumber
      && userHasNotQuit);

   if (userHasNotQuit
      && userGuess == correctNumber)
   {
      Console.WriteLine("Congratulations!");
   }
   else // wrong answer!
   {
      Console.WriteLine("Maybe next time!");
   }
}
```

}

The functionality of this application will be the same as the *while* sample. The only difference is how the loop is controlled. In practice, we will find that the *while* statement is used more often than the *do*/*while* statement. However, because we can easily control entry into the loop with the initialization of a Boolean variable, choosing one statement over the other is a choice of personal preference.

7.5.3 The *for* Statement

The most common iteration statement that is used is the *for* statement. The *for* statement is made up of three parts. One part is used to perform initialization at the beginning of the loop—this part is carried out only once. The second part is the conditional test that determines whether the loop is to be run again. And the last part, called **"stepping,"** is typically used to increment the counter that controls the loop's continuation—this counter is what is usually tested in the second part. The *for* statement takes the following form:

```
for (initialization; Boolean-expression; step)
    embedded-statement
```

The three parts (*initialization*, *Boolean-expression*, *step*) can be empty. When *Boolean-expression* evaluates to *false*, control is passed from the top of the loop to the next line following the *embedded-statement*. Therefore, the *for* statement works just like a *while* statement except that it gives the additional two parts (*initialization* and *step*). This is an example of a *for* statement that displays the printable ASCII characters:

```
using System;

class ForSolApp
{
   const int OpenChar = 23;
   const int CloseChar = 225;

   static public void Main()
   {
      for (int i = OpenChar; i <= CloseChar; i++)
      {
         Console.WriteLine("{0}={1}", i, (char)i);
      }
   }
}
```

The order of events for this *for* loop is as follows:

A value-typevariable (i) is allocated on the stack and initialized to 23. Note that this variable will be out of scope once the for loop concludes.

The embedded statement will execute while the variable i has a value less than 226. In this example, we have used a compound statement. However, because this for loop consists of a single line statement, we could also write this code without the braces and get the same results.

After each iteration through the loop, the variable i will be incremented by 1.

7.6 Nested Loops

In the *embedded-statement* of a *for* loop, we can also have other *for* loops, which are generally referred to as *nested loops*. Using the example from the previous section, I've added a nested loop that causes the application to print three characters per line, instead of one per line:

```
using System;

class NestedIsApp
{
   const int OpenChar = 23;
   const int CloseChar = 225;
   const int CharactersPerLine = 3;

   static public void Main()
   {
      for (int i = OpenChar; i <= CloseChar; i+=CharactersPerLine)
      {
         for (int j = 0; j < CharactersPerLine;  j++)
         {
            Console.Write("{0}={1} ", i+j, (char)(i+j));
         }
         Console.WriteLine("");
      }
   }

}
```

Note that the variable *i* that was defined in the outer loop is still in scope for the internal loop. However, the variable *j* is not available to the outer loop.

7.7 Using the Comma Operator

A comma can serve not only as a separator in method argument lists, but also as an operator in a *for* statement. In both the *initialization* and *step* portions of a *for* statement, the comma operator can be used to delimit multiple statements that are processed sequentially. In the following example the nested loop with a single for loop by using the comma operator is explained.

```
using System;
class CommaOpApp
{
   const int OpenChar = 33;
   const int CloseChar = 125;

   const int CharactersPerLine = 3;

   static public void Main()
   {
      for (int i = OpenChar, j = 1; i <= CloseChar; i++, j++)
      {
         Console.Write("{0}={1} ", i, (char)i);
         if (0 == (j % CharactersPerLine))
         {
            // New line if j is divisible by 3.
            Console.WriteLine("");
         }
      }
   }
}
```

Using the comma operator in the *for* statement can be powerful, but it can also lead to ugly and difficult-to-maintain code. Even with the addition of constants, the following code is an example of using the comma operator in a technically correct, but inappropriate, manner:

```
using System;
class CommaOp2App
{
   const int OpenChar = 23;
   const int CloseChar = 225;

   const int CharsPerLine = 3;
   const int NewLine = 13;
   const int Space = 32;

   static public void Main()
   {
      for (int i = OpenChar, extra = Space;
         i <= CloseChar;
         i++, extra = ((0 == (i - (OpenChar-1)) % CharsPerLine)
         ? NewLine : Space))
      {
         Console.Write("{0}={1} {2}", i, (char)i, (char)extra);
      }
   }
}
```

7.7.1 The *foreach* Statement

The foreach looping statements is new for programmers other than VB.The foreach statement allow us to iterate through all the items in an array or in other collections. C# also has such a construct, the *foreach* statement, which takes the following syntax.

```
foreach (type in expression)
      embedded-statement
```

Take a look at the following array class:

```
class ThisArray
```

```
{
   public ArrayList words;

   public ThisArray()
   {
      words = new ArrayList();
      words.Add("bag");
      words.Add("pouch");
      words.Add("case");
   }
}
```

From the different iteration statements we have already seen, we know that this array can be traversed using any of several different statements. However, to most Java and C++ programmers, the most logical way to write this application would be like this:

```
using System;
using System.Collections;

class ThisArray
{
   public ArrayList words;

   public ThisArray()
   {
      words = new ArrayList();
      words.Add("bag");
      words.Add("pouch");
      words.Add("case");
   }
}

class ForeachApp
{
   public static void Main()

   {
      ThisArray ThisArray = new ThisArray();

      for (int i = 0; i < ThisArray.words.Count; i++)
      {
         Console.WriteLine("{0}", ThisArray.words[i]);
      }
   }
}
```

But this approach is warrants with potential problems: If the *for* statement's initialization variable (*i*) isn't initialized properly, the entire list will not be be iterated and the *for* statement's

Boolean expression isn't correct, the entire list wil not be be iterated. Also the *for* statement's step is not correct, the entire list will not be iterated. Collections and arrays have different methods and properties for accessing their count. Collections and arrays have different semantics for extracting a specific element. The *for* loop's embedded statement needs to extract the element into a variable of the correct type.

With the *foreach* statement, the previous code would be rewritten as follows:

```
using System;
using System.Collections;

class ThisArray
{
   public ArrayList words;

   public ThisArray()
   {
      words = new ArrayList();
      words.Add("bag");
      words.Add("pouch");
      words.Add("case");
   }
}

class Foreach2App
{
   public static void Main()
   {
      ThisArray ThisArray = new ThisArray();

      foreach (string word in ThisArray.words)
      {
         Console.WriteLine("{0}", word);
      }
   }
}
```

Notice how much more intuitive using the *foreach* statement is. We are guaranteed to get every element because we don't have to manually set up the loop and request the count, and the statement automatically places the element in a variable that we name. We need only refer to the variable in the embedded statements.

7.7.2 Branching with Jump Statements

we can control the flow of execution of any of the iteration statements by embedding statements of covered in the previous sections, with one of several statements collectively known as jump statements: *break*, *continue*, *goto*, and *return*.

7.7.3 The *break* Statement

We use the *break* statement to terminate the current enclosing loop or conditional statement in which it appears. Control is then passed to the line of code following that loop's or conditional statement's embedded statement. Having the simplest syntax of any statement, the *break* statement has no parentheses or arguments and takes the following form at the point that we want to transfer out of a loop or conditional statement:

break

In the following example, the application will print each number from 1 to 100 that is equally divisible by 6. However, when the count reaches 66, the *break* statement will cause the *for* loop to discontinue.

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

   {
      for (int i = 1; i <= 100; i ++)
      {
         if (0 == i % 6)
         {
            Console.WriteLine(i);
         }

         if (i == 66)
         {
            break;
         }
      }
   }
}

}
```

7.7.4 Breaking Out of Infinite Loops

Another use for the *break* statement is to create an infinite loop in which control is transferred out of the loop only by a *break* statement being reached.

The following statement has changed the *while* statement to *while (true)* such that it will not end until a *break* statement is encountered.

```
using System;
class InfLoopApp
{
   const int MIN = 1;
   const int MAX = 10;
   const string EXIT_CHAR = "Q";
```

```
public static void Main()
{
   Precise rnd = new Precise();
   double correctNumber;

   string inputString;
   int userGuess;
   bool rightGuess = false;
   bool userQuit = false;

   while(true)
   {
      correctNumber = rnd.NextDouble() * MAX;
      correctNumber = Math.Round(correctNumber);

      Console.Write
         ("Guess a number between {0} and {1}...({2} to quit)",
         MIN, MAX, EXIT_CHAR);
      inputString = Console.ReadLine();

      if (0 == string.Compare(inputString, EXIT_CHAR, true))
      {
         userQuit = true;
         break;
      }
      else
      {
         userGuess = inputString.ToInt32();
         rightGuess = (userGuess == correctNumber);

         if ((rightGuess = (userGuess == correctNumber)))
         {
            break;
         }
         else
         {

            Console.WriteLine
               ("The correct number was {0}\n", correctNumber);
         }
      }
   }
   if (rightGuess && !userQuit)
   {
      Console.WriteLine("Congratulations!");
```

```
        }
        else
        {
            Console.WriteLine("Maybe next time!");
        }
    }
}
```

7.7.5 The *continue* Statement

Like the *break* statement, the *continue* statement enables us to alter the execution of a loop. However, instead of ending the current loop's embedded statement, the *continue* statement stops the current iteration and returns control back to the top of the loop for the next iteration. In the following example, the array of strings to be checked for duplicates. One way to accomplish that is to iterate through the array with a nested loop, comparing one element against the other. Therefore, if one index into the array (*i*) is the same as the other index into the array (*j*), it means that we have the same element and do not want to compare those two. In that case, we use the *continue* statement to discontinue the current iteration and pass control back to the top of the loop.

```
using System;
using System.Collections;

class ThisArray
{
    public ArrayList words;

    public ThisArray()
    {
        words = new ArrayList();
        words.Add("bag");
        words.Add("case");
        words.Add("pouch");
        words.Add("case");
        words.Add("case");
        words.Add("bag");
    }
}

class ContinueApp
{
    public static void Main()
    {
        ThisArray ThisArray = new ThisArray();
        ArrayList dupes = new ArrayList();

        Console.WriteLine("Processing array...");
        for (int i = 0; i < ThisArray.words.Count; i++)
        {
            for (int j = 0; j < ThisArray.words.Count; j++)

            {
                if (i == j) continue;

                if (ThisArray.words[i] == ThisArray.words[j]
                    && !dupes.Contains(j))
                {
                    dupes.Add(i);
                    Console.WriteLine("'{0}' appears on lines {1} and {2}",
                        ThisArray.words[i],
                        i + 1,
                        j + 1);
                }
            }
        }
        Console.WriteLine("There were {0} duplicates found",
            ((dupes.Count > 0) ? dupes.Count.ToString() : "no"));
    }
}
```

Notice that we could have used a *foreach* loop to iterate through the array. However, in this particular case, we wanted to keep track of which element we were on, so using a *for* loop was the best way.

7.7.6 goto Statement

Probably no other construct in programming history is as maligned as the *goto* statement. Therefore, before we get into the syntax for and some uses of the *goto* statement, let us look at why some people feel so strongly about not using this statement and the types of problems that can be solved by using it. The problem with using *goto* is not the keyword itself—rather, it is the use of *goto* in inappropriate places. The *goto* statement can be a useful tool in structuring program flow and can be used to write more expressive code than that resulting from other branching and iteration mechanisms. One such example is the "loop-and-a-half" problem, as coined by Dijkstra. Here is the traditional flow of the loop-and-a-half problem in pseudocode:

```
loop
    read in a value
    if value == sentinel then exit
    process the value
end loop
```

The exit from the loop is accomplished only by the execution of the *exit* statement in the middle of the loop. This *loop*/*exit*/*end loop* cycle, however, can be quite disturbing to some people who, acting on the notion that all uses of *goto* are evil, would write this code as follows:

```
read in a value
while value != sentinel
    process the value
    read in a value
end while
```

This second approach has two major drawbacks. First, it requires the duplication of the statement(s) required to read in a value. Any time we duplicate code, this results in an obvious maintenance problem in that any change to one statement must be made to the other. The second problem is more subtle and probably more damning. The key to writing solid code that's easy to understand, and therefore maintain, is to write code that reads in a natural manner. In any noncode description of what this code is attempting to do, one would describe the solution as follows: First, we need to read in a value. If that value is a sentinel, we stop. If not, we process that value and continue to the next value. Therefore, it is the code omitting the *exit* statement that's actually counterintuitive because that approach reverses the natural way of thinking about the problem. Now, let us look at some situations in which a *goto* statement can result in the best way to structure control flow.

7.7.7 Using the *goto* Statement

The *goto* statement can take any of the following forms:

```
goto identifier
goto case constant-expression
goto default
```

In the first usage of the *goto* statement here, the target of the *identifer* is a label statement. The label statement takes the form

```
identifer:
```

If that label does not exist in the current method, a compile-time error will result. Another important rule to remember is that the *goto* statement can be used to jump out of a nested loop. However, if the *goto* statement is not within the scope of the label, a compile-time error will result. Therefore, we cannot jump into a nested loop. In the following example, the application is iterating through a simple array, reading each value until it reaches a sentinel value whereupon it exits the loop. Notice that the *goto* statement really just acts like the *break* statement in that it causes the program flow to jump out of the *foreach* loop.

```
using System;
using System.Collections;
class ThisArray
{
   public ArrayList words;
   public const string TerminatingWord = "stop";

   public ThisArray()
   {
      words = new ArrayList();

      for (int i = 1; i <= 5; i++) words.Add(i.ToString());
      words.Add(TerminatingWord);
      for (int i = 6; i <= 10; i++) words.Add(i.ToString());
   }
}
```

```
class Goto1App
{
   public static void Main()
   {
      ThisArray ThisArray = new ThisArray();

      Console.WriteLine("Processing array...");

      foreach (string word in ThisArray.words)
      {
         if (word == ThisArray.TerminatingWord) goto finished;
         Console.WriteLine(word);
      }

      finished:
         Console.WriteLine("Finished processing array");
   }
}
```

Regarding this use of the *goto* statement, one could argue that a *break* statement could have been used just as effectively and a label wouldn't have been necessary. We will look at the other forms of the *goto* statement, and we will see that the problems at hand can be resolved only with the *goto* statement.

In the section on the *switch* statement, We discussed the fact that fall-throughs are not supported in C#. Even if fall-throughs were supported, it would not solve the following problem. Let us say we have a *Payment* class (reused from earlier in the chapter) that accepted several forms of payment, or tenders: Vbcard, United Express, CanCard, cash, and charge off (basically a credit). Because Vb, United Express, and CanCard are all credit cards, we could combine them into a single case label and process them all in the same manner. In the case of the charge off, we would need to call methods specific to it, and in the case of the cash purchase, we would only want to print a receipt. Also, we would want to print a receipt in all cases. How could we have three distinct case labels but have the first two cases—the credit card and the charge back cases—both branch to the cash label when done? The following code, this problem is a good example of when to use the *goto* statement:

```
using System;
enum Tenders : int
{
   ChargeOff,
   Cash,
   Vbcard,
   CanCard,
   UnitedExpress
};

class Payment
{
```

```
public Payment(Tenders tender)
{
   this.Tender = tender;
}

protected Tenders tender;
public Tenders Tender
{
   get
   {
      return this.tender;
   }

   set
   {
      this.tender = value;
   }
}

protected void ChargeOff()
{
   Console.WriteLine("Charge off.");
}

protected bool ValidateCreditCard()
{
   Console.WriteLine("Card approved.");
   return true;
}

protected void ChargeCreditCard()
{
   Console.WriteLine("Credit Card charged");
}

protected void PrintReceipt()
{
   Console.WriteLine("Thank we and come again.");
}

public void ProcessPayment()
{
   switch ((int)(this.tender))
   {
      case (int)Tenders.ChargeOff:
```

```
                ChargeOff();
                goto case Tenders.Cash;

            case (int)Tenders.Vbcard:
            case (int)Tenders.CanCard:
            case (int)Tenders.UnitedExpress:
                if (ValidateCreditCard())
                    ChargeCreditCard();
                goto case Tenders.Cash;

            case (int)Tenders.Cash:
                PrintReceipt();
                break;

            default:
                Console.WriteLine("\nSorry - Invalid tender.");
                break;
        }
    }
}

class GotoCaseApp
{
    public static void Main()
    {
        Payment payment = new Payment(Tenders.Vbcard);
        payment.ProcessPayment();
    }
}
```

Instead of having to solve the problem counterintuitively, we simply tell the compiler that when a credit card or charge off case is finished, we want to branch to the cash label. One last thing to note here is that if we branch out of a case label in C#, we should not use a *break* statement, which would result in a compiler error for "unreachable code." The last form of the *goto* statement enables us to branch to the *default* label in a *switch* statement, thereby giving we another means of writing a single code block that can be executed as a result of multiple evaluations of the *switch* statement.

7.7.8 The *return* Statement

The *return* statement has two functions. It specifies a value to be returned to the caller of the currently executed code (when the current code is not defined as returning *void*), and it causes an immediate return to the caller. The *return* statement is defined with the following syntax:

```
return [ return-expression ]
```

When the compiler encounters in a method, a *return* statement that specifies a *return-expression*, it evaluates whether the *return-expression* can be implicitly converted into a form compatible with the current method's defined return value. It is the result of that conversion that is passed back to the caller. When using the *return* statement with exception handling, we have to be sure that we understand some rules. If the *return* statement is a *try* block that contains an associated *finally* block, control is actually passed to the first line of the *finally* block, and when that block of code finishes, control is passed back to the caller. If the *try* block is nested in another *try* block, control will continue back up the chain in this fashion until the final *finally* block has executed.

Review Questions

1. What are the selection statements in C # ?
2. Give an example for switch ststement and explain?
3. What are the Iteration Statements? Explain with appropriate examples?
4. Explain the use of the break Statements?

Chapter 8. Operator overloading

C# like many other programming languages has canned set of tokens that are used to perform basic operations on intrinsic types. We have seen ,how to use the [ ] operator with a class to programmatically index an object as if it were an array. Now let us deals with closely related features of C# that provide the ability to create structure and class interfaces that are easier and to use: operator overloading and user-defined conversions. We will start with a general overview of operator overloading in terms of the benefits it provides we and then look at the actual syntax for redefining the default behaviors of operators as well as a realistic example application in which I overload the + operator to aggregate multiple *Account* objects. Then we have a listing of which binary and unary operators are overloadable as well as some of the restrictions involved. The operator overloading will end with some design guidelines to take into account when deciding whether to overload operators in the classes.

8.1 Operator Overloading

Operator overloading allows existing C# operators to be redefined for use with user-defined types. The overloading operator is simply another means of calling a method. Also the feature does not fundamentally add anything to the language. Although this is technically true, operator overloading does aid in one of the most important aspects of object-oriented programming: abstraction.

Suppose that we want to aggregate a collection of Accounts for a particular customer. Using operator overloading, we can write code similar to the following in which the += operator is overloaded:

```
Account summaryAccount = new Account ();
foreach (Account Account in customer.GetAccounts ())
{
   summaryAccount += Account;
}
```

These codes have distinct benefits and include very natural syntax and the fact that the client is abstracted from having to understand the implementation details of how the Accounts are being aggregated. Thus, operator overloading aids in creating software that is less expensive to write and maintain.

Operator overloading is a means of calling a method. To redefine an operator for a class, we need only use the following pattern, where *op* is the operator we're overloading:

public static *retval* operator *op* (*object1* [, *object2*])

Keep in mind the following facts when using operator overloading:

All overloaded operator methods must be defined as public and static.

Technically, retval (the return value) can be any type. But common practice to return the type for which the method is being defined with the exception of the true and false operators, which should always return a Boolean value.

The number of arguments passed (object1, object2) depends on the type of operator being overloaded. If a unary operator (an operator having a single operand) is being overloaded, there will be one argument. If a binary operator (an operator taking two operands) is being overloaded, two arguments are passed.

In the case of a unary operator, the argument to the method must be the same type as that of the enclosing class or struct. In other words, if we redefine the! Unary operator for a class called Pooh that method must take as its only argument a variable of type Pooh.

If the operator being overloaded is a binary operator, the first argument must be the same type as that of the enclosing class and the second argument can be any type.

We used the += operator with an *Account* class. For reasons we will soon understand, we can't actually overload these compound operators. We can overload only the "base" operator—in this case, the +. Here's the syntax used to define the *Account* class's *operator+* method:

```
public static Account operator+ (Account Account1, Account Account2)
{
   // Create a new Account object.
   // Add the desired contents from
   // Account1 to the new Account object.
   // Add the desired contents from
   // Account2 to the new Account object.
   // Return the new Account object.
 }
```

Let us look at a more substantial example now, one with two main classes: *Account* and *AccountDetailLine*. The *Account* class has a member variable of type *ArrayList* that represents a collection of all Account detail lines. To allow for the aggregation of detail lines for multiple Accounts, we have to overloaded the + operator. The *Account. operator+* method creates a new *Account* object and iterates through both Account objects' arrays of Account detail lines, adding each detail line to the new *Account* object. This *Account* object is then returned to the caller. Obviously, in a real-world invoicing module, this would be much more complex, but the point here is to show somewhat realistically how operator overloading could be used.

```
using System;
using System.Collections;

class AccountDetailLine
{
   double lineTotal;
   public double LineTotal
   {
      get
      {
         return this.lineTotal;
      }
   }

   public AccountDetailLine(double LineTotal)
```

```
    {
        this.lineTotal = LineTotal;
    }
}

class Account
{
    public ArrayList DetailLines;

    public Account()
    {
        DetailLines = new ArrayList();
    }

    public void PrintAccount()
    {
        Console.WriteLine("\nLine Nbr\tTotal");

        int i = 1;
        double total = 0;
        foreach(AccountDetailLine detailLine in DetailLines)
        {
            Console.WriteLine("{0}\t\t{1}", i++, detailLine.LineTotal);
            total += detailLine.LineTotal;
        }

        Console.WriteLine("=====\t\t===");
        Console.WriteLine("Total\t\t{1}", i++, total);
    }

    public static Account operator+ (Account Account1, Account Account2)
    {
        Account returnAccount = new Account();

        foreach (AccountDetailLine detailLine in Account1.DetailLines)
        {
            returnAccount.DetailLines.Add(detailLine);
        }

        foreach (AccountDetailLine detailLine in Account2.DetailLines)
        {
            returnAccount.DetailLines.Add(detailLine);
        }
        return returnAccount;
    }
```

```
}

class AccountAddApp
{
   public static void Main()
   {
      Account i1 = new Account();
      for (int i = 0; i < 2; i++)
      {
         i1.DetailLines.Add(new AccountDetailLine(i + 1));
      }

      Account i2 = new Account();
      for (int i = 0; i < 2; i++)
      {
         i2.DetailLines.Add(new AccountDetailLine(i + 1));
      }

      Account summaryAccount = i1 + i2;

      summaryAccount.PrintAccount();
   }
}
```

8.1.1 Overloadable Operators

Only the following unary and binary operators can be overloaded.
Unary operands: +, -,!, ~, ++, --, *true*, *false*
Binary operands: +, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, <=

8.1.2 Restrictions (on Operator Overloading)

It is not possible to overload the = assignment operator. However, when we overload a binary operator, its compound assignment equivalent is implicitly overloaded. For example, if we overload the + operator, the += operator is implicitly overloaded in that the user-defined *operator+* method will be called.

The [ ] operators can't be overloaded. However, user-defined object indexing is supported through indexers. The parentheses used to perform a cast are also not overloadable. Instead, we should use conversion operators, which are also referred to as user-defined conversions. Operators that are currently not defined in the C# language cannot be overloaded. It is not possible for example, define ** as a means of defining exponentiation because C# does not define a ** operator. Also, an operator's syntax can't be modified. We can't change the binary * operator to take three arguments when, by definition, its syntax calls for two operands. Finally, an operator's precedence can't be altered

8.1.3 Design Guidelines

We've seen what operator overloading is and how to use it in C#, so let us examine an often-ignored aspect of this useful feature: design guidelines. What we want to stay away from is the natural tendency to want to use a new feature for the sake of using it. This phenomenon is sometimes referred to as "a solution in search of a problem." But it is always a good design approach to remember the adage, "Code is read more than it is written." Keep the class's client in mind when determining if and when to overload an operator or set of operators. we should overload an operator only if it makes the class's interface more intuitive to use. For example, it makes perfect sense to be able to add Accounts together. In addition, we have to think as a client of the class would. Let us say we're writing an *Account* class and we want the client to be able to discount the Account. We and I might know that a credit line item will be added to the Account, but the point of encapsulation is that the clients don't have to know the exact implementation details of the class. Therefore, overloading the * operator—as shown here—might be a good idea because it would serve to make the *Account* class's interface natural and intuitive:

```
Account *= .85; // 15% discount.
```

8.1.4 Conversions – User Defined

The parentheses used for casting cannot be overloaded and that user-defined conversions must be used instead. In short, user-defined conversions enable us to declare conversions on structures or classes such that the *struct* or *class* can be converted to other structures, classes, or basic C# types. Why and when would we want to do that? Let us consider say we needed to use the standard Pound and Dollar scales in the application such that we could easily convert between the two. By creating user-defined conversions, we could use the following syntax:

```
Pound f = 1.72F;
Dollar c = (Dollar) f; // User-defined conversion.
```

This does not provide any functional benefits over the following syntax, but it is more intuitive to write and easier to read.

```
pound f = 98.6P;
Dollar D= f.ConvertToDollar ();
```

Syntax

The syntax of the user-defined conversion uses the *operator* keyword to declare user-defined conversions:

public static implicit operator *conv-type-out* (*conv-type-in operand*)

public static explicit operator *conv-type-out* (*conv-type-in operand*)

There are only a couple of rules regarding the syntax of defining conversions:

Any conversion method for a struct or class—we can define as many as we need-must be static. Conversions must be defined as either implicit or explicit. The implicit keyword means that the cast is not required by the client and will occur automatically. Conversely, using the explicit keyword signifies that the client must explicitly cast the value to the desired type.

All conversions either must take (as an argument) the type that the conversion is being defined on or must return that type. As with operator overloading, the operator keyword is used in the conversion method signature but without any appended operator.

In this example, we have two structures (*Pound* and *Dollar*) that enable the client to convert a value of type *float* to either currency scale. I'll first present the *Pound* structure and make some points about it, and then we will see the complete working application.

```
struct Pound
{
   public Pound(float curr)
   {
      this.curr = curr;
   }

   public static implicit operator Pound(float curr)
   {
      Pound c;
      c = new Pound(curr);
      return(c);
   }

   public static implicit operator float(Pound c)
   {
      return((((c.curr /1.72 ));
   }

   public float curr;
}
```

The first decision that we see was the one to use a structure instead of a class. There is no real reason for doing that other than the fact that using classes is more expensive than using structures—in terms of how the classes are allocated—and a class is not really necessary here because the *Pound* structure does not need any C# class-specific features, such as inheritance.

Next notice that a constructor have been declared, Which that takes a float as its only argument. This value is stored in a member variable named *curr*. Now look at the conversion operator defined immediately after the structure's constructor. This is the method that will be called when the client attempts to cast a float to *Pound* or use a float in a place, such as with a method, where a *Pound* structure is expected. This method does not have to do much, and in fact this is fairly formulaic code that can be used in most basic conversions. Here, simply instantiate a *Pound* structure and then return that structure. That return call is what will cause the last method defined in the structure to be called. As we can see, the method simply provides the mathematical formula for converting from a Dollar value to a Pound value.

Here is the entire application, including a *Dollar* structure:

```
using System;

struct Pound
{
   public Pound(float curr)

   {
       this.curr = curr;
   }

   public static implicit operator Pound(float curr)
   {
       Pound c;
       c = new Pound(curr);
       return(c);
   }

   public static implicit operator float(Pound c)
   {
       return((((c.curr  /1.72));
   }

   public float curr;
}

struct Dollar
{
   public Dollar(float curr)
   {
       this.curr = curr;
   }

   public static implicit operator Dollar(float curr)
   {
       Dollar f;
       f = new Dollar(curr);
       return(f);
   }

   public static implicit operator float(Dollar f)
   {
       return((((f.curr * 1.72));
   }

   public float curr;
}

class Curr1App
{
   public static void Main()

   {
       float t;

       t=98.6F;
       Console.Write("Conversion of {0} to Pound = ", t);
       Console.WriteLine((Pound)t);

       t=0F;
       Console.Write("Conversion of {0} to Dollar = ", t);
       Console.WriteLine((Dollar)t);
   }
}
```

If the above code is compiled and executed, it produces the following output:

```
Conversion of 1.72 to Pound = 1
Conversion of 0 to Dollar = 1.72
```

This works pretty well, and being able to write *(Pound)1.72F* is certainly more apt than calling some static class method. But note that we can pass only values of type *float* to these conversion methods. For the application above, the following will not compile:

```
Pound c = new Pound(55);
Console.WriteLine((Dollar)c);
```

Also, because there is no Pound conversion method that takes a *Dollar* structure (or vice versa), the code has to assume that the value being passed in is a value that needs converting. However, if that value is then passed back to the conversion method again, the conversion method has no way of knowing that the value has already been converted and logically already represents a valid Pound currency—to the conversion method, it is just a float. As a result, the value gets converted again. Therefore, we need to modify the application so that each structure can take as a valid argument the other structure.

The revised code with ensuing comments:

```
using System;

class Currency
{
    public Currency(float Curr)
    {
        this.curr = Curr;
    }

    protected float curr;
    public float Curr
    {
        get
        {
            return this.curr;
        }
    }
}
```

```
class Pound : Curreratue
{
   public Pound(float Curr)
      : base(Curr) {}

   public static implicit operator Pound(float Curr)
   {
      return new Pound(Curr);
   }

   public static implicit operator Pound(Dollar F)
   {
      return new Pound(F.Curr);
   }

   public static implicit operator float(Pound C)
   {
      return((((C.curr/  1.72));
   }
}

class Dollar : Currency
{
   public Dollar(float Curr)
      : base(Curr) {}

   public static implicit operator Dollar(float Curr)
   {
      return new Dollar(Curr);
   }

   public static implicit operator Dollar(Pound C)
   {
      return new Dollar(C.Curr);
   }

   public static implicit operator float(Dollar F)

   {
      return((((F.curr * 1.72));
   }
}

class Curr2App
```

```
{
   public static void DisplayCurr(Pound Curr)
   {
      Console.Write("Conversion of {0} {1} to Dollar = ",
         Curr.ToString(), Curr.Curr);
      Console.WriteLine((Dollar)Curr);
   }

   public static void DisplayCurr(Dollar Curr)
   {
      Console.Write("Conversion of {0} {1} to Pound = ",
         Curr.ToString(), Curr.Curr);
      Console.WriteLine((Pound)Curr);
   }

   public static void Main()
   {
      Dollar f = new Dollar(1.72F);
      DisplayCurr(f);

      Pound c = new Pound(0F);
      DisplayCurr(c);
   }
}
```

We changed the *Pound* and *Dollar* types from *struct* to *class*. A more practical reason for doing so is to share the *curr* member variable by having the *Pound* and *Dollar* classes derive from the same *Currency* base class. I can also now use the inherited (from *System.Object*) *ToString* method in the application's output.

The only other difference of note is the addition of a conversion for each currency scale that takes as an argument a value of the other currency scale. Notice how similar the code is between the two Pound conversion methods:

```
public static implicit operator Pound(float curr)
{
   Pound c;
   c = new Pound(curr);
   return(c);
}

public static implicit operator Pound(Dollar f)
{
   Pound c;
   c = new Pound(f.curr);
   return(c);
}
```

The only tasks that should be done differently were to change the argument being passed and retrieve the currency from the passed object instead of a hard-coded value of type *float*. That is why we noted earlier how easy and formulaic conversion methods are once we know the basics

Review Questions

1.What do you mean by operater overloading? Illustrate with an example ?
2.What are overloadable operaters in C# ?
3.What is the restriction on operater overloading?
4.Explain the use of user defined conversion?

Chapter 9 Interface

9.1 Interfaces

Interfaces are contracts between two disparate pieces of code. That is, once an interface is defined and a class is defined as implementing that interface, clients of the class are guaranteed that the class has implemented all methods defined in the interface. The key to understanding interfaces might be to compare them with classes. Classes are objects that have properties and methods that act on those properties. While classes do exhibit some behavioral characteristics (methods), classes are **things** as opposed to **behaviors**, and that's how interfaces come in. Interfaces enable We to define behavioral characteristics, or abilities, and apply those behaviors to classes irrespective of the class hierarchy. For example, say that We have a distribution application in which some of the entities can be serialized. These might include the *Customer*, *Supplier* and *Invoice* classes. Some other classes, such as *MaintenanceView* and *Document*, might not be defined as serializable. The only way would be to create a base class called something like **Serializable**. However, that approach has a major drawback. A single inheritance path will not work because we do not want all the behaviors of the class to be shared. C# does not support multiple inheritance, so there's no way to have a given class selectively derive from multiple classes. The answer is interfaces. Interfaces give We the ability to define a set of semantically related methods and properties that selected classes can implement regardless of the class hierarchy.

9.1.1 Use of Interface

To understand where interfaces are useful, let us first look at a traditional programming problem in Microsoft Windows development that does not use an interface but in which two pieces of disparate code need to communicate in a generic fashion.

As we know that there is a standard programming model in windows for handling situations in which we have a piece of code that we want future unknown code to communicate with in a generic manner. The biggest disadvantage to this technique is that it forces the client the Control Panel code, in this case to include a lot of validation code. For example, Control Panel couldn't just assume that any .cpl file in the folder is a Windows DLL. Also, Control Panel needs to verify that the correction functions are in that DLL and that those functions do what the documentation specifies. This is where interfaces come in. Interfaces let us create the same contractual arrangement between disparate pieces of code but in a more object-oriented and flexible manner. In addition, because interfaces are a part of the C# language, the compiler ensures that when a class is defined as implementing a given interface, the class does what it says it does.

In C#, an interface is a first-class concept that declares a reference type that includes method declarations only. The term means that the feature in question is a built-in, integral part of the language. In other words, it is not something that was bolted on after the language was designed.

9.1.2 Declaring Interfaces

Interfaces can contain methods, properties, indexers, and events—none of which are implemented in the interface itself. Let us look at an example to see how to use this feature. Suppose

We are designing an editor for a company that hosts different windows controls. We are writing the editor and the routines that validate the controls users place on the editor's form. The rest of the team is writing the controls that the forms will host. We almost certainly need to provide some sort of form-level validation. At appropriate times—such as when the user explicitly tells the form to validate all controls or during form processing—the form could then iterate through all its attached controls and validate each control, or more appropriately, tell the control to validate itself.

We can provide this control validation capability. This is where interfaces excel. Here's a simple interface example containing a single method named **Validate:**

```
interface IOblivion
{
   bool Validate();
}
```

Now We can document the fact that if the control implements the *IOblivion* interface, the control can be validated.

First, We do not need to specify an access modifier such as *public* on an interface method. In fact, prepending the method declaration with an access modifier results in a compile-time error. This is because all interface methods are public by definition. In addition to methods, interfaces can define properties, indexers, and events, as shown here:

```
interface IExampleInterface
{
   // Example property declaration.
   int testProperty { get; }

   // Example event declaration.
   event testEvent Changed;

   // Example indexer declaration.
   string this[int index] { get; set; }
}
```

9.1.3 Implementing Interfaces

Because an interface defines a contract, any class that implements an interface **must define each and every item in that interface** or the code will not compile. Using the *IOblivion* example, a client class would need to implement only the interface's methods. In the following example, the base class called *ImpInter* and an interface called *IOblivion*. We then have a class, *MyControl*, that derives from *ImpInter* and implements *IOblivion*. Note the syntax and how the *MyControl* object can be cast to the *IOblivion* interface to reference its members.

```
using System;
public class ImpInter
{
   protected string Data;
      public string data
      {
         get
```

```
        {
           return this.Data;
        }
        set
        {
           this.Data = value;
        }
     }
}

interface IOblivion
{
   bool Validate();
}

class MyControl : ImpInter, IOblivion
{
   public MyControl()
   {
      data = "my grid data";
   }

   public bool Validate()
   {
      Console.WriteLine("Validating...{0}", data);
            return true;
   }
}

class InterfaceApp
{
   public static void Main()
   {
      MyControl myControl = new MyControl();

      IOblivion val = (IOblivion)myControl;
      bool success = val.Validate();
   Console.WriteLine("The validation of '{0}' was {1}successful",
                  myControl.data,
                  (true == success ? "" : "not "));
   }
}
```

Using the preceding class and interface definition, the editor can query the control as to whether the control implements the *IOblivion* interface. If it does, the editor can validate and then call the implemented interface methods. We will be pushed to think that, why we just define a base class to use with this editor that has a pure virtual function called *Validate*? The editor would then accept only controls that are derived from this base class, right?That would be a workable solution, but it would be severely limiting. Let us say We create our own controls and they all derive from this hypothetical base class. Consequently, they all implement this *Validate* virtual method. This works until the day we find a really cool control that we want to use with the editor. Let us say we find a grid that was written by someone else. As such, it won't be derived from the editor's mandatory control base class. In C++, the answer is to use multiple inheritance and derive the grid from both the third-party grid and the editor's base class. However, C# does not support multiple inheritance.

Using interfaces, we can implement multiple behavioral characteristics in a single class. In C# We can derive from a single class and, in addition to that inherited functionality, implement as many interfaces as the class needs. For example, if we wanted the editor application to validate the contents of the control, bind the control to a database, and serialize the contents of the control to disk, we would declare our class as follows:

```
public class MyGrid : ThirdPartyGrid, IOblivion,
                ISerializable, IDataBound
{
}
```

Implementation (Using *is*)

In the *InterfaceApp* example, We saw the following code, which was used to cast an object (*MyControl*) to one of its implemented interfaces (*IOblivion*) and then call one of those interface members (*Validate*):

```
MyControl myControl = new MyControl();
IOblivion val = (IOblivion)myControl;
bool success = val.Validate();
```

The following example will compile because *ISerializable* is a valid interface. Nonetheless, at run time a *System. InvalidCastException* will be thrown because *MyGrid* does not implement the *ISerializable* interface. The application would then abort unless this exception was being explicitly caught.

```
using System;

public class ImpInter
{
   protected string Data;
   public string data
   {
      get
      {
         return this.Data;
      }
      set
      {
         this.Data = value;
      }
```

```
    }
}

interface ISerializable
{
    bool Save();
}

interface IOblivion
{
    bool Validate();
}

class MyControl : ImpInter, IOblivion
{
    public MyControl()
    {
        data = "my grid data";
    }

    public bool Validate()
    {
        Console.WriteLine("Validating...{0}", data);
        return true;
    }
}

class IsOperator1App
{
    public static void Main()
    {
        MyControl myControl = new MyControl();

        ISerializable ser = (ISerializable)myControl;

        // NOTE: This will throw a System.InvalidateCastException
        // because the class does not implement the ISerializable
        // interface.
        bool success = ser.Save();

        Console.WriteLine("The saving of '{0}' was {1}successful",
 myControl.data,
 (true == success ? "" : "not "));
    }
}
```

Catching the exception does not change the fact that the code we want to execute will not execute in this case. What we need is a way to query the object *before* attempting to cast it. One way of doing this is by using the “**is**” operator. The “**is**” operator enables us to check at run time whether one type is compatible with another type. It takes the following form, where *expression* is a reference type:

expression is type

The **is** operator results in a Boolean value and can, therefore, be used in conditional statements. In the following example, has beeen modified the code to test for compatibility between the *MyControl* class and the *ISerializable* interface before attempting to use an *ISerializable* method:

```
using System;
public class ImpInter
{
   protected string Data;
   public string data
   {
      get
      {
         return this.Data;
      }
      set
      {
         this.Data = value;
      }
   }
}
interface ISerializable
{
   bool Save();
}
interface IOblivion
{
   bool Validate();
}
class MyControl : ImpInter, IOblivion
{
   public MyControl()
   {
      data = "my grid data";
   }

   public bool Validate()
   {
      Console.WriteLine("Validating...{0}", data);
      return true;
   }
}

class IsOperator2App
{
   public static void Main()
   {
      MyControl myControl = new MyControl();

      if (myControl is ISerializable)
      {
         ISerializable ser = (ISerializable)myControl;
         bool success = ser.Save();

         Console.WriteLine("The saving of '{0}' was {1}successful",
myControl.data,
(true == success ? "" : "not "));
      }
      else
      {
       Console.WriteLine("The ISerializable interface is not implemented.");
      }
   }
}
```

Now that We have seen how the *is* operator enables us to verify the compatibility of two types to ensure proper usage, let us look at one of its close relatives—the *as* operator—and compare the two.

9.1.4 Querying for Implementation (Using *as*)

If we look closely at the MSIL code generated from the preceding *IsOperator2App* example—that MSIL code - We will notice one problem with the *is* operator. We can see that *isinst* is called right after the object is allocated and the stack is set up. The *isinst* opcode is generated by the compiler for the C# *is* operator. This opcode tests to see if the object is an instance of a class or interface. Notice that just a few lines later—assuming the conditional test passes—the compiler has generated the *castclass* IL opcode. The *castclass* opcode does its own verification and, while this opcode works slightly differently than *isinst*, the result is that the generated IL is doing inefficient work by checking the validity of this cast twice.

We can think of the **as** operator as a combination of the *is* operator and, if the two types in question are compatible, a cast. An important difference between the *as* operator and the *is* operator is that the *as* operator sets the object equal to *null* instead of returning a Boolean value if *expression* and *type* are incompatible. Our example can now be rewritten in the following more efficient manner:

```
using System;
public class ImpInter
{
   protected string Data;
   public string data
```

```
    {
        get
        {
            return this.Data;
        }
        set
        {
            this.Data = value;
        }
    }
}

interface ISerializable
{
    bool Save();
}

interface IOblivion
{
    bool Validate();
}


class MyControl : ImpInter, IOblivion
{
    public MyControl()
    {
        data = "my grid data";
    }

    public bool Validate()
    {
        Console.WriteLine("Validating...{0}", data);
        return true;
    }
}

class AsOperatorApp
{
    public static void Main()
    {
        MyControl myControl = new MyControl();

        ISerializable ser = myControl as ISerializable;
        if (null != ser)
        {
           bool success = ser.Save();

           Console.WriteLine("The saving of '{0}' was {1}successful",
myControl.data,
(true == success ? "" : "not "));
        }
        else
        {
          Console.WriteLine("The ISerializable interface is not implemented.");
        }
    }
}
```

9.1.5 Explicit Interface

Untill now we have seen classes implement interfaces by specifying the access modifier *public* followed by the interface method's signature. However, sometimes We will want (or even need) to explicitly qualify the member name with the name of the interface. In this section, we will examine two common reasons for doing this.

9.1.6 Name Hiding with Interfaces

The most common way to call a method implemented from an interface is to cast an instance of that class to the interface type and then call the desired method. While this is valid and many people (including myself) use this technique, technically We don't have to cast the object to its implemented interface to call that interface's methods. This is because when a class implements method of an interface, those methods are also public methods of the class. Take a look at this C# code and especially the *Main* method to see it means:

```
using System;
public interface IDataBound
{
    void Bind();
}

public class EditBox : IDataBound
{
    // IDataBound implementation.
    public void Bind()
    {
        Console.WriteLine("Binding to data store...");
    }
}

class NameHiding1App
{
    // Main entry point.
```

```
   public static void Main()
   {
      Console.WriteLine();

      EditBox edit = new EditBox();
      Console.WriteLine("Calling EditBox.Bind()...");
      edit.Bind();

      Console.WriteLine();

      IDataBound bound = (IDataBound)edit;
      Console.WriteLine("Calling (IDataBound)EditBox.Bind()...");
      bound.Bind();
   }
}
```

This example will now output the following:

```
Calling EditBox.Bind()...
Binding to data store...

Calling (IDataBound))EditBox.Bind()...
Binding to data store...
```

Notice that although this application calls the implemented *Bind* method in two different ways—one with a cast and one without—both calls function correctly in that the *Bind* method is processed. Although at first blush the ability to directly call the implemented method without casting the object to an interface might seem like a good thing, at times this is less than desirable. The most obvious reason is that the implementation of several interfaces—each of which might contain numerous members—could quickly pollute The public namespace of the class with members that have no meaning outside the scope of the implementing class. We can prevent the implemented members of interfaces from becoming public members of the class by using a technique called *name hiding*.

Name hiding at its simplest is the ability to hide an inherited member name from any code outside the derived or implementing class (commonly referred to as the *outside world*). Let us say that we have the same example as we used earlier where an *EditBox* class needs to implement the *IDataBound* interface—however, this time the *EditBox* class does not want to expose the *IDataBound* methods to the outside world. Rather, it needs this interface for its own purposes, or perhaps the programmer simply does not want to clutter the class's namespace with a large number of methods that a typical client will not use. To hide an implemented interface member, We need only remove the member's *public* access modifier and qualify the member name with the interface name, as shown here:

```
using System;
public interface IDataBound
{
   void Bind();
}

public class EditBox : IDataBound
```

```
{
   // IDataBound implementation.
   void IDataBound.Bind()
   {
      Console.WriteLine("Binding to data store...");
   }
}

class NameHiding2App
{
   public static void Main()
   {
      Console.WriteLine();

      EditBox edit = new EditBox();
      Console.WriteLine("Calling EditBox.Bind()...");

      // ERROR: This line won't compile because
      // the Bind method no longer exists in the
      // EditBox class's namespace.
      edit.Bind();

      Console.WriteLine();

      IDataBound bound = (IDataBound)edit;
      Console.WriteLine("Calling (IDataBound)EditBox.Bind()...");

      // This is OK because the object was cast to
      // IDataBound first.
      bound.Bind();
}
}
```

The preceding code will not compile because the member name *Bind* is no longer a part of the *EditBox* class. Therefore, this technique enables us to remove the member from the class's namespace while still allowing explicit access by using a cast operation.

One point we have to reiterate is that when we are hiding a member, we cannot use an access modifier. We will receive a compile-time error if we try to use an access modifier on an implemented interface member. We might find this odd, but consider that the entire reason for hiding something is to prevent it from being visible outside the current class. Since access modifiers exist only to define the level of visibility outside the base class, we can see that they don't make sense when we use name hiding.

9.1.7 Avoiding Name Ambiguity

One of the main reasons that C# does not support multiple inheritance is the problem of name collision, which results from name ambiguity. Although C# does not support multiple inheritance at the object level (derivation from a class), it does support inheritance from one class and the additional implementation of multiple interfaces. However, with this power comes a price: name collision.

In the following example, we have two interfaces, *ISerializable* and *IDataStore*, which support the reading and storing of data in two different formats—one as an object to disk in binary form and the other to a database. The problem is that they both contain methods named *SaveData*:

```
using System;

interface ISerializable
{
   void SaveData();
}

interface IDataStore
{
   void SaveData();
}

class Test : ISerializable, IDataStore
{
   public void SaveData()
   {
      Console.WriteLine("Test.SaveData called");
   }
}

class NameCollisions1App
{
   public static void Main()
   {
      Test test = new Test();

      Console.WriteLine("Calling Test.SaveData()");
      test.SaveData();
   }
}
```

At the time of this writing, this code does compile. However, it was told that in future building of the C# compiler, the code will result in a compile-time error because of the ambiguity of the implemented *SaveData* method. Regardless of whether this code compiles, we would have a problem at run time because the resulting behavior of calling the *SaveData* method would not be clear to the programmer attempting to use the class. We get the *SaveData* that serializes the object to disk, or We get the *SaveData* that saves to a database?

In addition, take a look at the following code:

```
using System;
interface ISerializable
{
   void SaveData();
}


interface IDataStore
{
   void SaveData();
}

class Test : ISerializable, IDataStore
{
   public void SaveData()
   {
      Console.WriteLine("Test.SaveData called");
   }
}

class NameCollisions2App
{
   public static void Main()
   {
      Test test = new Test();

      if (test is ISerializable)
      {
         Console.WriteLine("ISerializable is implemented");
      }

      if (test is IDataStore)
      {
         Console.WriteLine("IDataStore is implemented");
      }
   }
}
```

Here, the *is* operator succeeds for both interfaces, which indicates that both interfaces are implemented although we know that not to be the case!

The problem is that the class has implemented either a serialized version or a database version of the *Bind* method (not both). However, if the client checks for the implementation of one of the interfaces—both will succeed—and happens to try to use the one that was not truly implemented, unexpected results will occur.

We can turn to explicit member name qualification to get around this problem: remove the access modifier, and prepend the member name—*SaveData*, in this case—with the interface name:

```
using System;
interface ISerializable
{
    void SaveData();
}

interface IDataStore
{
    void SaveData();
}

class Test : ISerializable, IDataStore
{
    void ISerializable.SaveData()
    {
        Console.WriteLine("Test.ISerializable.SaveData called");
    }
    void IDataStore.SaveData()
    {
        Console.WriteLine("Test.IDataStore.SaveData called");
    }
}

class NameCollisions3App
{
    public static void Main()
    {
        Test test = new Test();

        if (test is ISerializable)
        {
            Console.WriteLine("ISerializable is implemented");
            ((ISerializable)test).SaveData();
        }

        Console.WriteLine();

        if (test is IDataStore)
        {
            Console.WriteLine("IDataStore is implemented");
            ((IDataStore)test).SaveData();
        }
    }
}
```

Now there is no ambiguity as to which method will be called. Both methods are implemented with their fully qualified names, and the resulting output from this application is what We would expect:

```
ISerializable is implemented
```

```
Test.ISerializable.SaveData called

IDataStore is implemented
Test.IDataStore.SaveData called
```

9.2 Interfaces and Inheritance

Two common problems are associated with interfaces and inheritance. The first problem, illustrated here with a code example, deals with the issue of deriving from a base class that contains a method name identical to the name of an interface method that the class needs to implement.

```
using System;
public class Control
{
   public void Serialize()
   {
      Console.WriteLine("Control.Serialize called");
   }
}

public interface IDataBound
{
   void Serialize();
}

public class EditBox : Control, IDataBound
{
}

class InterfaceInh1App
{
   public static void Main()
   {
      EditBox edit = new EditBox();
      edit.Serialize();
   }
}
```

As we know, to implement an interface, We must provide a definition for every member in that interface declaration. However, in the preceding example we don't do that, and the code still compiles! The reason it compiles is that the C# compiler looks for an implemented **Serialize** method in the *EditBox* class and finds one. However, the compiler is incorrect in determining that this is the implemented method. The *Serialize* method found by the compiler is the *Serialize* method inherited from the *Control* class and not an actual implementation of the *IDataBound.Serialize* method. Therefore, although the code compiles, it will not function as expected, as we will see next.

Now let us make things a little more interesting. Notice that the following code first checks—via the *as* operator—that the interface is implemented and then attempts to call an implemented *Serialize* method. The code compiles and works. However, as we know, the *EditBox* class does not really implement a *Serialize* method as a result of the *IDataBound* inheritance. The *EditBox* already had a *Serialize* method (inherited) from the *Control* class. This means that in all likelihood the client is not going to get the expected results.

```
using System;
public class Control
{
   public void Serialize()
   {
      Console.WriteLine("Control.Serialize called");
   }
}

public interface IDataBound
{
   void Serialize();
}

public class EditBox : Control, IDataBound
{
}

class InterfaceInh2App
{
   public static void Main()
   {
      EditBox edit = new EditBox();

   IDataBound bound = edit as IDataBound;
   if (bound != null)
   {
      Console.WriteLine("IDataBound is supported...");
      bound.Serialize();
   }

   else
   {
      Console.WriteLine("IDataBound is NOT supported...");
   }
   }
}
```

Another potential problem to watch for occurs when a derived class has a method with the same name as the base class implementation of an interface method. Let us look at that in code as well:

```
using System;
interface ITest
{
   void Pooh();
}

// Base implements ITest.
class Base : ITest
{
   public void Pooh()
   {
      Console.WriteLine("Base.Pooh (ITest implementation)");
   }
}

class MyDerived : Base
{
   public new void Pooh()
   {
      Console.WriteLine("MyDerived.Pooh");
   }
}

public class InterfaceInh3App
{
   public static void Main()
   {
      MyDerived myDerived = new MyDerived();
      myDerived.Pooh();

      ITest test = (ITest)myDerived;
      test.Pooh();
   }
}
```

This code results in the following screen output:

```
MyDerived.Pooh
Base.Pooh (ITest implementation)
```

In this situation, the *Base* class implements the *ITest* interface and its *Pooh* method. However, the *MyDerived* class derives from *Base* with a new class and implements a new *Pooh* method for that class. Which *Pooh* gets called? It depends on what reference We have. If We have a reference to the *MyDerived* object, its *Pooh* method will be called. This is because even though the *myDerived* object has an inherited implementation of *ITest.Pooh*, the run time will execute the *MyDerived.Pooh* because the *new* keyword specifies an override of the inherited method.

However, when We explicitly cast the *myDerived* object to the *ITest* interface, the compiler resolves to the interface implementation. The *MyDerived* class has a method of the same name, but that's not what the compiler is looking for. When We cast an object to an interface, the compiler traverses the inheritance tree until a class is found that contains the interface in its base list. This is why the last two lines of code in the *Main* method result in the *ITest* implemented *Pooh* being called.

Hopefully, some of these potential pitfalls involving name collisions and interface inheritance have supported my strong recommendation: always cast the object to the interface whose member we are attempting to use.

Combining Interfaces

Another powerful feature of C# is the ability to combine two or more interfaces together such that a class need only implement the combined result. For example, let us say We want to create a new *TreeView* class that implements both the *IDragDrop* and *ISortable* interfaces. Since it is reasonable to assume that other controls, such as a *ListView* and *ListBox,* would also want to combine these features, We might want to combine the *IDragDrop* and *ISortable* interfaces into a single interface:

```
using System;
public class Control
{
}

public interface IDragDrop
{
   void Drag();
   void Drop();
}


public interface ISerializable
{
   void Serialize();
}

public interface ICombo : IDragDrop, ISerializable
{
   // This interface does not add anything new in
   // terms of behavior as its only purpose is
   // to combine the IDragDrop and ISerializable
   // interfaces into one interface.
}

public class MyTreeView : Control, ICombo
{
   public void Drag()
```

```
    {
        Console.WriteLine("MyTreeView.Drag called");
    }

    public void Drop()
    {
        Console.WriteLine("MyTreeView.Drop called");
    }

    public void Serialize()
    {
        Console.WriteLine ("MyTreeView.Serialize called");
    }
}

class CombiningApp
{
    public static void Main()
    {
        MyTreeView tree = new MyTreeView();
        tree.Drag();
        tree.Drop();
        tree.Serialize();
    }
```

With the ability to combine interfaces, We can, not only simplify the ability to aggregate semantically related interfaces into a single interface, but also add additional methods to the new "composite" interface, if needed.

Review Questions

1. What is an interface?
2. Explain the use if Interface with an example?
3. How will you implement an Interface?
4. Describe Explicit interface?
5. Explain Interface Inheirtence in C# ?
6. Explain with an example how to combine two interfaces?

Chapter 10. Exception Handling

10.1 Error Handling with Exceptions

This chapter deals about general mechanics and basic syntax of exception handling. Once we have this baseline knowledge, we will see how exception handling compares with the more prevalent methods of error handling today and we will discover the advantages that exception handling has over these other techniques. We will then dive into some of the more specific .NET exception-handling issues, such as using the *Exception* class and deriving our own exception classes. Finally, the last part of this chapter will deal with the issue of properly designing the system to use exception handling.

One of the main goals of the .NET Common Language Runtime (CLR) is for run-time errors to either be avoided (through features such as automatic memory and resource management when using managed code) or at least caught at compile time (by a strongly typed system). However, certain errors can be caught only at run time, and therefore a consistent means of dealing with errors must be used across all the languages that comply with the Common Language Specification (CLS). To that extent, this chapter focuses on the error-handling system implemented by the CLR—exception handling.

10.2 Overview of Exception Handling

Exceptions are error conditions that arise when the normal flow of a code path—that is, a series of method calls on the call stack—is impractical or imprudent. It is imperative to understand the difference here between an exception and an expected event (such as reaching the end of a file). If we have a method that is sequentially reading through a file, we know that at some point it is going to reach the end of the file. Therefore, this event is hardly *exceptional* in nature and certainly does not prevent the application from continuing. However, if we're attempting to read through a file and the operating system alerts we to the fact that a disk error has been detected, this is certainly an exceptional situation and definitely one that will affect the normal flow of wer method's attempt to continue reading the file.

Most exceptions also involve another problem: *context*. Let us look at an example. Assuming that we're writing tightly cohesive code—code in which one method is responsible for one action—the (pseudo)code might look like this:

```
public void Foo()
{
   File file = OpenFile(String fileName);
   while (!file.IsEOF())
   {
      String record = file.ReadRecord();
   }
CloseFile();
}
public void OpenFile(String fileName)
{   // Attempt to lock and open file.
}
```

Note that if the *OpenFile* method fails, it cannot handle the error. This is because its only responsibility is to open files. It can't determine whether the inability to open the specified file constitutes a catastrophic error or a minor annoyance. Therefore, *OpenFile* cannot handle the error condition because the method is said to not be in the correct context.

This is the entire reason for the existence of exception handling: one method determines that an exception condition has been reached and happens not to be in the correct context to deal with the error. It signals to the runtime that an error has occurred. The runtime traverses back up the call stack until it finds a method that can properly deal with the error condition. Obviously, this becomes even more important if the code path is five methods deep with the fifth method reaching an error condition and the first method being the only method that can deal with the error correctly. Now let us look at the syntax used in exception handling.

10.2.1 Basic Exception-Handling Syntax

Exception handling consists of only four keywords: **try, catch, throw**, and **finally**. The way the keywords work is simple and straightforward. When a method fails its objective and cannot continue—that is, when it detects an exceptional situation—it throws an exception to the calling method by using the *throw* keyword. The calling method, assuming it has enough context to deal with the exception, then receives this exception via the *catch* keyword and decides what course of action to take. In the following sections, we will look at the language semantics governing how to throw and catch exceptions as well as some code snippets that will illustrate how this works.

10.2.2 Throwing an Exception

When a method needs to notify the calling method that an error has occurred, it uses the *throw* keyword in the following manner:

throw statement:

throw *expression*$_{\text{opt}}$;

We will get into the different ways in which we can throw exceptions a bit later. For now, it is enough to realize that when we throw an exception, we're required to throw an object of type *System.Exception* (or a derived class). Following is an example of a method that has determined that an unrecoverable error has occurred and that it needs to throw an exception to the calling method. Notice how it instantiates a new *System.Exception* object and then throws that newly created object back up the call stack.

```
public void SomeMethod()
{
// Some error determined.
throw new Exception();
}
```

10.2.3 Catching an Exception

Obviously, because a method can throw an exception, there must be a reciprocal part of this equation where something has to catch the thrown exception. The *catch* keyword defines a block of code that is to be processed when an exception of a given type is caught. The code in this block is referred to as an *exception handler*.

One thing to keep in mind is that not every method needs to deal with every possible thrown exception, especially because a method might not have the context required to do anything with the error information. After all, the point of exception handling is that errors are to be handled by code that has enough context to correctly do so. We use two keywords to catch an exception: *try* and *catch*.

To catch an exception, we need to bracket the code we're attempting to execute in a *try* block and then specify which types of exceptions the code within that *try* block is capable of handling in a *catch* block. All the statements in the *try* block will be processed in order as usual unless an exception is thrown by one of the methods being called. If that happens, control is passed to the first line of the appropriate *catch* block. By "appropriate catch block," we mean the block that's defined as catching the type of exception that is thrown. Here's an example of a method (*Foo*) calling and catching an exception being thrown by another method (*Bar*):

```
public void Foo()
{
   try
   {
      Bar();
   }
   catch(System.Exception e)
   {
      // Do something with error condition.
   }
}
```

If *Bar* throws an exception and *Foo* does not catch it then the result depends on the design of the application. When an exception is thrown, control is passed back up the call stack until a method is found that has a *catch* block for the type of exception being thrown. If a method with an appropriate *catch* block is not located, the application aborts. Therefore, if a method calls another method that does throw an exception, the design of the system must be such that some method in the call stack must handle the exception.

10.2.4 Rethrowing an Exception

There will be times after a method has caught an exception and done everything it can in its context that it then *rethrows* the exception back up the call stack. This is done very simply by using the *throw* keyword. Here's an example of how that looks:

```
using System;
class RethrowApp
{
   static public void Main()
   {
      Rethrow rethrow = new Rethrow();

      try
      {
         rethrow.Foo();
      }
      catch(Exception e)
```

```
        {
            Console.WriteLine(e.Message);
        }
    }

    public void Foo()
    {
        try
        {
            Bar();
        }
        catch(Exception)
        {
            // Handle error.
            throw;
        }
    }

    public void Bar()
    {
        throw new Exception("thrown by Rethrow.Bar");
    }
}
```

In this example, *Main* calls *Foo*, which calls *Bar*. *Bar* throws an exception that *Foo* catches. *Foo* at this point does some preprocessing and then rethrows the exception back up the call stack to *Main* by using the *throw* keyword.

10.2.5 Cleaning Up with *finally*

One sticky issue with exception handling is ensuring that a piece of code is always run regardless of whether an exception is caught. For example, suppose we allocate a resource such as a physical device or a data file. Then suppose that we open this resource and call a method that throws an exception. Regardless of whether the method can continue working with the resource, we still need to deallocate or close that resource. This is when the *finally* keyword is used, like so:

```
using System;
public class ThrowException1App
{
    public static void ThrowException()
    {
        throw new Exception();
    }

    public static void Main()
    {
        try
        {
            Console.WriteLine("try...");
        }
        catch(Exception e)
        {
            Console.WriteLine("catch...");
        }
        finally
        {
            Console.WriteLine("finally");
        }
    }
}
```

As we can see, the existence of the *finally* keyword prevents we from having to put this cleanup code both in the *catch* block and after the *try*/*catch* blocks. Now, despite whether an exception is thrown, the code in the *finally* block will be executed.

10.3 Comparing Error-Handling Techniques

Now that we've seen the basics of throwing and catching exceptions, let us take a few minutes to compare the different approaches to error handling taken in programming languages.

The standard approach to error handling has typically been to return an error code to the calling method. The calling method is then left with the responsibility of deciphering the returned value and acting accordingly. The return value can be as simple as a basic C or C++ type or it can be a pointer to a more robust object containing all the information necessary to fully appreciate and understand the error. More elaborately designed error-handling techniques involve an entire error subsystem in which the called method indicates the error condition to the subsystem and then returns an error code to the caller. The caller then calls a global function exported from the error subsystem in order to determine the cause of the last error registered with the subsystem. We can find an example of this approach in the Microsoft **Open Database Connectivity** (ODBC) SDK. However, regardless of the exact semantics, the basic concept remains the same: the calling method in some way calls a method and inspects the returned value to verify the relative success or failure of the called method. This approach, although being the standard for many years, is severely flawed in a number of important ways. The following sections describe a few of the ways exception handling provides tremendous benefits over using return codes.

10.3.1 The Benefits of Exception Handling Over Return Codes

When using return codes, the called method returns an error code and the error condition is handled by the calling method. Because the error handling occurs outside the scope of the called method, there is no guarantee that the caller will check the returned error code. As an example, let us say that we write a class called *CommaDelimitedFile* that wraps the functionality of reading and writing standard comma-delimited files. Part of what wer class would have to expose includes methods to open and read data from the file. Using the older return code method of reporting errors, these methods would return some variable type that would have to be checked by the caller to verify the success of the method call. If the user of wer class called the *CommaDelimitedFile.Open* method and then attempted to call the *CommaDelimitedFile.Read* method without checking whether the *Open* call succeeded, this could—and probably would in the case of a demo in front of the most important client—cause less than desirable results. However, if the class's *Open* method throws an exception, the caller would be forced to deal with the fact that the *Open* method failed. This is because each time a method throws an exception, control is passed back up the call stack until it is caught. Here's an example of what that code might look like:

```
using System;
class ThrowException2App
{
   class CommaDelimitedFile
   {
      protected string fileName;

      public void Open(string fileName)
      {
         this.fileName = fileName;

         // Attempt to open file
         // and throw exception upon error condition.
         throw new Exception("open failed");
      }

      public bool Read(string record)
      {
         // Code to read file.
         return false; // EOF
      }
   }

   public static void Main()
   {
      try
      {
         Console.WriteLine("attempting to open file");

         CommaDelimitedFile file = new CommaDelimitedFile();
         file.Open("c:\\test.csv");

         string record = "";

         Console.WriteLine("reading from file");

         while (file.Read(record) == true)
         {
            Console.WriteLine(record);
```

```
        }

        Console.WriteLine("finished reading file");
      }
      catch (Exception e)
      {
        Console.WriteLine(e.Message);
      }
    }
}
```

In this example, if either the *CommaDelimitedFile.Open* method or the *CommaDelimitedFile.Read* method throws an exception, the calling method is forced to deal with it. If the calling method does not catch the exception and no other method in the current code path attempts to catch an exception of this type, the application will abort. Pay particular attention to the fact that because the *Open* method call is placed in a block, an invalid read (using our example in which the *Open* method has thrown an exception) would not be attempted. This is because programmatic control would be passed from the *Open* call in the *try* block to the first line of the *catch* block. Therefore, one of the biggest benefits of exception handling over return codes is that exceptions are programmatically more difficult to ignore.

10.4 Handling Errors

One general principle of good programming is the practice of **tight cohesion**, which refers to the objective or purpose of a given method. Methods that demonstrate tight cohesion are those that perform a single task. The main benefit of using tight cohesion in wer programming is that a method is more likely to be portable and used in different scenarios when it performs only a single action. A method that performs a single action is also certainly easier to debug and maintain. However, tightly cohesive code does cause one major problem with regards to error handling. Let us look at an example to illustrate the problem and how exception handling solves it.

In this example, a class (*AccessDatabase*) is used to generate and manipulate Microsoft Access databases. Let us say this class has a static method called *GenerateDatabase*. Because the *GenerateDatabase* method would be used to create new Access databases, it would have to perform several tasks to create the database. For example, it would have to create the physical database file, create the specified tables (including any rows and columns wer application needs), and define any necessary indexes and relations. The *GenerateDatabase* method might even have to create some default users and permissions.

The programmatic design problem is as follows: if an error were to occur in the *CreateIndexes* method, which method would handle it and how? Obviously, at some point, the method that originally called the *GenerateDatabase* method would have to handle the error, but how could it? It would have no idea how to handle an error that occurred several method calls deep in the code path. As we've seen, the calling method is said to not be in the correct context to handle the error. In other words, the only method that could logically create any meaningful error information about the error is the method that failed. Having said that, if return codes were used in our *AccessDatabase* class, each method in the code path would have to check for every single error code that every other method *might* return.

One obvious problem with this is that the calling method would potentially have to handle a ridiculously large number of error codes. In addition, maintenance would be difficult. Every time an error condition was added to any of the methods in the code path, every other instance in the application where a method calls that method would have to be updated to handle the new error code. Needless to say, this is not an inexpensive proposition in terms of software total cost of ownership (TCO).

Exception handling resolves all of these issues by enabling the calling method to trap for a given type of exception. In the example we're using, if a class called *AccessDatabaseException* were derived from *Exception*, it could be used for any types of errors that occur within any of the *AccessDatabase* methods. Then, if the *CreateIndexes* method failed, it would construct and throw an exception of type *AccessDatabseException*. The calling method would catch that exception and be able to inspect the *Exception* object to decipher what exactly went wrong. Therefore, instead of handling every possible type of return code that *GenerateDatbase* and any of its called methods could return, the calling method would be assured that if *any* of the methods in that code path failed, the proper error information would be returned. Exception handling provides an additional bonus: because the error information is contained within a class, new error conditions can be added and the calling method will remain unchanged. And was not extensibility—being able to build something and then add to it without changing or breaking existing code—one of the original promises of object-oriented programming to begin with? For these reasons, the concept of catching and dealing with errors in the correct context is the most significant advantage to using exception handling.

10.4.1 Improving Code Readability

When using exception-handling code, code readability improves enormously. This directly relates to reduced costs in terms of code maintenance. The way in which return codes are handled vs. the exception-handling syntax is the source of this improvement. If we used return codes with the *AccessDatabase.GenerateDatbase* method mentioned previously, code similar to the following would be required to handle error conditions:

```
public bool GenerateDatabase()
{
  if (CreatePhysicalDatabase())
  {
    if (CreateTables())
    {
      if (CreateIndexes())
      {
        return true;
      }
      else
      {
        // Handle error.
        return false;
      }

    }
    else
```

```
        {
           // Handle error.
           return false;
        }
    }
    else
    {
        // Handle error.
        return false;
    }
}
```

Add a few other validations to the preceding code and we wind up with a tremendous amount of error validation code mixed in with wer business logic. If we indent wer code 4 spaces per block, the first character of a line of code might not appear until column 20 or greater. None of this is disastrous for the code itself, but it does make the code more difficult to read and maintain, and the bottom line is that code that is difficult to maintain is a breeding ground for bugs. Let us look at how this same example would look if exception handling were used:

```
        // Calling code.
try
{
    AccessDatabase accessDb = new AccessDatabase();
    accessDb.GenerateDatabase();
}
catch(Exception e)
{
    // Inspect caught exception.
}

// Definition of AccessDatabase.GenerateDatabase method.
public void GenerateDatabase()
{
    CreatePhysicalDatabase();
    CreateTables();
    CreateIndexes();
}
```

Notice how much cleaner and more elegant the second solution is. This is because error detection and recovery code are no longer mixed with the logic of the calling code itself. Because exception handling has made this code more straightforward, maintaining the code has been made much easier.

10.4.2 Throwing exceptions

Another major advantage that exceptions have over other error-handling techniques is in the area of object construction. Because a constructor cannot return values, there's simply no easy, intuitive means of signaling to the method calling the constructor method that an error occurred during object construction. Exceptions, however, can be used because the calling method need only wrap the construction of the object in a *try* block, as in the following code:

```
try
{
   // If the AccessDatabase object fails to construct
   // properly and throws an exception, it will now be caught.
   AccessDatabase accessDb = new AccessDatabase();
}
catch(Exception e)
{
   // Inspect the caught exception.
}
```

10.4.3 Using the *System.Exception* Class

All exceptions that are to be thrown must be of the type (or derived from) *System.Exception*. In fact, the *System.Exception* class is the base class of several exception classes that can be used in the C# code. Most of the classes that inherit from *System.Exception* do not add any functionality to the base class. So why does C# bother with derived classes if the derived classes aren't going to significantly differ from their base class? The reason is that a single *try* block can have multiple *catch* blocks with each *catch* block specifying a specific exception type. This enables the code to handle different exceptions in a manner applicable to a specific exception type.

Constructing an *Exception* Object

As of this writing, there are four different constructors for the *System.Exception* class:

```
public Exception ();
public Exception(String);
protected Exception(SerializationInfo, StreamingContext);
public Exception(String, Exception);
```

The first constructor listed above is the default constructor. It takes no arguments and simply defaults all member variables. This exception is typically thrown as follows:

```
// Error condition reached.
throw new Exception();
```

The second exception constructor takes as its only argument a *String* value that identifies an error message and is the form that we've seen in most of the examples in this chapter. This message is retrieved by the code catching the exception via the *System.Exception.Message* property. Here's a simple example of both sides of this exception propagation:

```
using System;
class ThrowException3
{
   class FileOps
   {
      public void FileOpen(String fileName)
      {
         // ...
         throw new Exception("Oh bother");
```

```
        }

        public void FileRead()
        {
        }
    }

    public static void Main()
    {
        // Code catching exception.
        try
        {
            FileOps fileOps = new FileOps();

            fileOps.FileOpen("c:\\test.txt");
            fileOps.FileRead();
        }
        catch(System.Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

The third constructor initializes an instance of the *Exception* class with serialized data.

Finally, the last constructor enables we to specify not only an error message but also what's known as an *inner exception*. The reason for this is that when handling exceptions, at times we will want to *massage* the exception a bit at one level of the call stack before passing it further up the call stack. Let us look at an example of how we might use this feature.

Let us say that in order to ease the burden on the class's client, we decide to throw only one type of exception. That way the client only has to catch one exception and reference the exception's *InnerException* property. This has the added benefit that if we decide to modify a given method to throw a new type of exception after the client code has been written, the client does not have to be updated unless it wants to do something specific with this exception.

In the following example, notice that the client code catches the top-level *System.Exception* and prints a message contained in that exception's inner exception object. If at later date the *DoWork* method throws other kinds of exceptions-—as long as it does so as inner exceptions of an *System.Exception* object—the client code will continue to work.

```
using System;
using System.Globalization;
class FooLib
{
    protected bool IsValidParam(string value)
    {
        bool success = false;
        if (value.Length == 3)
        {
            char c1 = value[0];
            if (Char.IsNumber(c1))
            {
                char c2 = value[2];
                if (Char.IsNumber(c2))
                {
                    if (value[1] == '.')
                        success = true;
                }
            }
        }

        return success;
    }

    public void DoWork(string value)
    {
        if (!IsValidParam(value))throw new Exception
                ("", new FormatException("Invalid parameter specified"));
        Console.WriteLine("Work done with '{0}'", value);
    }
}

class FooLibClientApp
{
    public static void Main(string[] args)
    {
        FooLib lib = new FooLib();
        try
        {
            lib.DoWork(args[0]);
        }
        catch(Exception e)
        {
            Exception inner = e.InnerException;
            Console.WriteLine(inner.Message);
        }
    }
}
```

10.4.4 Using the *StackTrace* Property

Another useful property of the *System.Exception* class is the *StackTrace* property. The *StackTrace* property enables we to determine—at any given point at which we have a valid *System.Exception* object—what the current call stack looks like. Take a look at the following code:

```
using System;
class StackTraceTestApp
{
   public void Open(String fileName)
   {
      Lock(fileName);
      // ...
   }
   public void Lock(String fileName)

   {
      // Error condition
      throw new Exception("failed to lock file");
   }
   public static void Main()
   {
      StackTraceTestApp test = new StackTraceTestApp();

      try
      {
         test.Open("c:\\test.txt");
         // Use file.
      }
      catch(Exception e)
      {
         Console.WriteLine(e.StackTrace);
      }
   }
}
```

In this example, what prints out is the following:

```
at StackTraceTest.Main()
```

Therefore, the *StackTrace* property returns the call stack at the point that the exception is caught, which can be useful for logging and debugging scenarios.

Catching Multiple Exception Types

In various situations, we might want a *try* block to catch different exception types. For example, a given method might be documented as throwing several different exception types, or we might have several method calls in a single *try* block where each method being called is documented as being able to throw a different exception type. This is handled by adding a *catch* block for each type of exception that wer code needs to handle:

```
try
{
   Foo(); // Can throw FooException.
   Bar(); // Can throw BarException.
}
catch(FooException e)
```

```
{
   // Handle the error.
}
catch(BarException e)
{
   // Handle the error.
}

catch(Exception e)
{
}
```

Each exception type can now be handled with its distinct *catch* block (and error-handling code). However, one extremely important detail here is the fact that the base class is handled last. Obviously, because all exceptions are derived from *System.Exception*, if we were to place that *catch* block first, the other *catch* blocks would be unreachable. To that extent, the following code would be rejected by the compiler:

```
try
{
   Foo(); // Can throw FooException.
   Bar(); // Can throw BarException.
}
catch(Exception e)
{
   // ***ERROR - THIS WON'T COMPILE
}
catch(FooException e)
{
   // Handle the error.
}
catch(BarException e)
{
   // Handle the error.
}
```

10.5 Deriving Own *Exception* Classes

When we want to provide extra information or formatting to an exception before we throw it to the client code. We might also want to provide a base class for that situation so that the class can publish the fact that it throws only one type of exception. That way, the client only need concern itself with catching this base class.

Yet another example of when we might want to derive our own *Exception* class is if we wanted to perform some action—such as logging the event or sending an email to someone at the help desk—every time an exception was thrown. In this case, we would derive our own *Exception* class and put the needed code in the class constructor, like this:

```
using System;
public class TestException : Exception

{
   // We would probably have extra methods and properties
   // here that augment the .NET Exception that we derive from.

   // Base Exception class constructors.
   public TestException()
      :base() {}
   public TestException(String message)
      :base(message) {}
   public TestException(String message, Exception innerException)
      :base(message, innerException) {}
}
public class DerivedExceptionTestApp
{
   public static void ThrowException()
   {
      throw new TestException("error condition");
   }
   public static void Main()
   {
      try
      {
         ThrowException();
      }
      catch(Exception e)
      {
         Console.WriteLine(e.ToString());
      }
   }
}
```

Although not a hard-and-fast rule, it is good programming practice—and consistent with most C# code we will see—to name wer exception classes such that the name ends with the word "Exception." For example, if we wanted to derive an *Exception* class for a class called *MyFancyGraphics*, we could name the class *MyFancyGraphicsException*.

Another rule of thumb when creating or deriving wer own exception classes is to implement all three *System.Exception* constructors. Once again, this isn't absolutely necessary, but it does improve consistency with other C# code that wer client will be using.

This code will generate the following output. Note that the *ToString* method results in a combination of properties being displayed: the textual representation of the exception class name, the message string passed to the exception's constructor, and the *StackTrace*.

```
TestException: error condition
  at DerivedExceptionTestApp.Main()
```

10.6 Designing Code with Exception Handling

So far, we've covered the basic concepts of using exeption handling and the semantics related to the throwing and catching of exceptions. Now let us look at an equally important facet of exception handling: understanding how to design wer system with exception handling in mind. Suppose we have three methods: *Foo*, *Bar*, and *Baz*. *Foo* calls *Bar*, which calls *Baz*. If *Baz* publishes the fact that it throws an exception, does *Bar* have to catch that exception even if it can't, or won't, do anything with it? How should code be split with regards to the *try* and *catch* blocks?

10.6.1 Issues with the *try* Block

We know how to catch an exception that a called method might throw, and we know that control passes up the call stack until an appropriate *catch* block is found. So the question is, should a *try* block catch every possible exception that a method within it can throw? The answer is no, to maintain the biggest benefits of using exception handling in wer applications: reduced coding and lower maintenance costs. The following example illustrates a program where a catch is rethrown, to be handled by another catch block.

```
using System;
class WhenNotToCatchApp
{
   public void Foo()
   {
      try
      {
         Bar();
      }
      catch(Exception e)
      {
         Console.WriteLine(e.Message);
      }
   }

   public void Bar()
   {
      try
      {

         Baz();
      }
      catch(Exception e)
      {
// Bar should catch this because it
// does not do anything but rethrow it.
throw;
      }
   }

   public void Baz()
   {
      throw new Exception("Exception originally thrown by Baz");
   }

   public static void Main()
   {
      WhenNotToCatchApp test = new WhenNotToCatchApp();
      test.Foo(); // This method will ultimately
              // print the error message.
   }
}
```

In this example, *Foo* catches the exception that *Baz* throws even though it then does nothing except rethrow the exception back up to *Bar*. *Bar* then catches the rethrown exception and does something with the information—in this case, displaying the *Exception* object's error message property. Following are a few reasons why methods that simply rethrow an exception should not catch that exception in the first place Because the method does not do anything with the exception, we would be left with a catch block in the method that is superfluous at best. Obviously, it is never a good idea for code to exist when it has absolutely no function.

If the exception type being thrown by Baz were to change, we would have to go back and change both the Bar catch block and the Foo catch block. Why put werself in a situation in which we would have to alter code that does not even do anything. Because the CLR automatically continues up the call stack until a method catches the exception, intermediate methods can ignore the exceptions that they cannot process. Just make sure in the design that *some* method catches the exception because, as mentioned earlier, if an exception is thrown and no *catch* block is found in the current call stack, the application will abort.

10.6.2 Design Issues with the *catch* Block

The only code that should appear in a *catch* block is code that will at least partially process the caught exception. For example, at times a method will catch an exception, do what it can to process that exception, and then rethrow that exception so that further error handling can occur. The following example illustrates this. *Foo* has called *Bar*, which will do some database work on *Foo* 's behalf. Because commitment control, or transactions, are being used, *Bar* needs to catch any error that occurs and rollback the uncommitted changes before rethrowing the exception back to *Foo*.

```
using System;
class WhenToCatchApp
{
   public void Foo()
   {
      try
      {
         Bar();
      }
      catch(Exception e)
      {
         Console.WriteLine(e.Message);
      }
   }

   public void Bar()
   {
      try
      {
         // Call method to set a "commitment boundary."
         Console.WriteLine("setting commitment boundary");

         // Call Baz to save data.
         Console.WriteLine("calling Baz to save data");
         Baz();

         Console.WriteLine("commiting saved data");
      }
      catch(Exception)
      {
            // In this case, Bar should catch the exception
            // because it is doing something significant
            //(rolling back uncommitted database changes).

            Console.WriteLine("rolling back uncommited changes " +

                  "and then rethrowing exception");

         throw;
      }
   }

   public void Baz()
   {
      throw new Exception("db failure in Baz");
   }

   public static void Main()
   {
      WhenToCatchApp test = new WhenToCatchApp();
      test.Foo(); // This method will ultimately print
```

```
            // the error msg.
    }
}
```

Bar needed to catch the exception to do its own error processing. When finished, it then rethrew the exception back up the call stack, where *Foo* caught it and was able to do its error processing. This ability of each level of the call stack to do just the amount of error processing that it is capable of, given its context, is one example of what makes exception handling so important to the design of the system.

Review Questions

1. What is exception handling? Explain
2. What the keywords for exception handling in C#?
3. What are user defined exception . Illustrate with an example?
4. Differentiate Catch block and finally block?
5. Describe briefly Handling errors?
6. Explain the sailient features of error handling in C#?
7. Error handling improves code readability. Substantiate with an example?

Chapter 11. Delegates and events

11.1 Delegates and Event Handlers

Another useful innovation of the C# language is something called delegates, which basically serve the same purpose as function pointers in C++. However, delegates are type-safe, secure managed objects. This means that the runtime guarantees that a delegate points to a valid method, which further means that we get all the benefits of function pointers without any of the associated dangers, such as an invalid address or a delegate corrupting the memory of other objects. In this chapter, we will be looking at delegates, how they compare to interfaces, the syntax used to define them, and the different problems that they were designed to address. We will also see several examples of using delegates with both callback methods and asynchronous event handling.

We know that how interfaces are defined and implemented in C#. As it is explained, from a conceptual standpoint interfaces are simply contracts between two disparate pieces of code. However, interfaces are much like classes in that they're defined at compile time and can include methods, properties, indexers, and events. Delegates, on the other hand, refer only to single methods and are defined at run time. Delegates have two main usages in C# programming: callbacks and event handling. Let us start off by talking about callback methods.

11.1.1 Using Delegates as Callback Methods

Used extensively in programming for Microsoft Windows, callback methods are used when we need to pass a function pointer to another function that will then call we back (via the passed pointer). An example would be the Win32 API *EnumWindows* function. This function enumerates all the top-level windows on the screen, calling the supplied function for each window. Callbacks serve many purposes, but the following are the most common:

Asynchronous processing Callback methods are used in asynchronous processing when the code being called will take a good deal of time to process the request. Typically, the scenario works like this: Client code makes a call to a method, passing to it the callback method. The method being called starts a thread and returns immediately. The thread then does the majority of the work, calling the callback function as needed. This has the obvious benefit of allowing the client to continue processing without being blocked on a potentially lengthy synchronous call.

Injecting custom code into a code of the class path Another common use of callback methods is when a class allows the client to specify a method that will be called to do custom processing. Let us look at an example in Windows to illustrate this. Using the Listbox class in Windows, we can specify that the items be sorted in ascending or descending order. Besides some other basic sort options, the Listbox class can't really give any latitude and remain a generic class. Therefore, the Listbox class also enables we to specify a callback function for sorting. That way, when Listbox sorts the items, it calls the callback function and wer code can then do the custom sorting we need.

Now let us look at an example of defining and using a delegate.In this example, we have a database manager class that keeps track of all active connections to the database and provides a method for enumerating those connections. Assuming that the database manager is on a remote server, it might be a good design decision to make the method asynchronous and allow the client to provide a callback method. Note that for a real-world application we would typically create this as a multithreaded application to make it truly asynchronous.

First, let us define two main classes: *DBManager* and *DBConnection*.

```
class DBConnection
{
}
class DBManager
  {
    static DBConnection[] activeConnections;
    public delegate void EnumConnectionsCallback(DBConnection connection);
      public static void EnumConnections(EnumConnectionsCallback callback)
      {
        foreach (DBConnection connection in activeConnections)
        {
          callback(connection);
        }
      }
  }
```

The *EnumConnectionsCallback* method is the delegate and is defined by placing the keyword *delegate* in front of the method signature. We can see that this delegate is defined as returning *void* and taking a single argument: a *DBConnection* object. The *EnumConnections* method is then defined as taking an *EnumConnectionsCallback* method as its only argument. To call the *DBManager*. *EnumConnections* method, we need only pass to it an instantiated *DBManager*. *EnumConnectionCallback* delegate.

To do that, we *new* the delegate, passing to it the name of method that has the same signature as the delegate. Here's an example of that:

```
DBManager.EnumConnectionsCallback myCallback =
    new DBManager.EnumConnectionsCallback(ActiveConnectionsCallback);
DBManager.EnumConnections(myCallback);
```

Also note that we can combine this into a single call like so:

```
DBManager.EnumConnections(new
   DBManager.EnumConnectionsCallback(ActiveConnectionsCallback));
```

That's all there is to the basic syntax of delegates. Now let us look at the full example application:

```
using System;
class DBConnection
{
   public DBConnection(string name)
   {
      this.name = name;
   }

     protected string Name;
     public string name
     {
        get
        {
           return this.Name;
        }
        set
        {
           this.Name = value;
        }
     }
  }

  class DBManager
  {
     static DBConnection[] activeConnections;
     public void AddConnections()
     {
        activeConnections = new DBConnection[5];
        for (int i = 0; i < 5; i++)
        {
           activeConnections[i] =
new DBConnection("DBConnection " + (i + 1));
        }
     }

     public delegate void EnumConnectionsCallback(DBConnection connection);
     public static void EnumConnections(EnumConnectionsCallback callback)
     {
        foreach (DBConnection connection in activeConnections)
        {
           callback(connection);
        }
     }
  }

  class Delegate1App
  {
     public static void ActiveConnectionsCallback(DBConnection connection)
     {
        Console.WriteLine("Callback method called for "
                       + connection.name);
     }

     public static void Main()
     {
         DBManager dbMgr = new DBManager();
         dbMgr.AddConnections();
```

```
    DBManager.EnumConnectionsCallback myCallback =
    new DBManager.EnumConnectionsCallback(ActiveConnectionsCallback);

    DBManager.EnumConnections(myCallback);
    }
}
```

Compiling and executing this application results in the following output:

```
Callback method called for DBConnection 1
Callback method called for DBConnection 2
Callback method called for DBConnection 3
Callback method called for DBConnection 4
Callback method called for DBConnection 5
```

11.1.2 Delegates as Static Members

Because it is kind of clunky that the client has to instantiate the delegate each time the delegate is to be used, C# allows we to define as a static class member the method that will be used in the creation of the delegate. Following is the example from the previous section, changed to use that format. Note that the delegate is now defined as a static member of the class named *myCallback*, and also note that this member can be used in the *Main* method without the need for the client to instantiate the delegate.

```
using System;
class DBConnection
{
  public DBConnection(string name)
  {
    this.name = name;
  }

  protected string Name;
  public string name
  {
    get
    {
      return this.Name;
    }

    set
    {
      this.Name = value;
    }
  }
}

class DBManager
{
   static DBConnection[] activeConnections;
   public void AddConnections()
   {
      activeConnections = new DBConnection[5];
      for (int i = 0; i < 5; i++)
      {
         activeConnections[i] = new
DBConnection("DBConnection " + (i + 1));
      }
   }

   public delegate void EnumConnectionsCallback(DBConnection connection);
   public static void EnumConnections(EnumConnectionsCallback callback)
   {
      foreach (DBConnection connection in activeConnections)
      {
         callback(connection);
      }
   }
}

class Delegate2App
{
   public static DBManager.EnumConnectionsCallback myCallback =
      new DBManager.EnumConnectionsCallback(ActiveConnectionsCallback);
   public static void ActiveConnectionsCallback(DBConnection connection)
   {
      Console.WriteLine ("Callback method called for " +
                     connection.name);
   }

   public static void Main()
   {
      DBManager dbMgr = new DBManager();
      dbMgr.AddConnections();

      DBManager.EnumConnections(myCallback);
   }
}
```

11.1.3 Creating Delegates

In the two examples we've seen so far, the delegate is created whether or not it is ever used. That was fine in those examples because, we knew that it would always be called. However, when defining the delegates, it is important to consider when to create them. Let us say, for example, that the creation of a particular delegate is time-consuming and not something we want to do gratuitously. In these situations in which we know the client won't typically call a given callback method, we can put off the creation of the delegate until it is actually needed by wrapping its instantiation in a property. To illustrate how to do this, a modification of the *DBManager* class follows that uses a read-only property—because only a getter method is present—to instantiate the delegate. The delegate will not be created until this property is referenced.

```
using System;
class DBConnection
{
   public DBConnection(string name)
   {
      this.name = name;
   }

   protected string Name;
   public string name
   {
      get
      {
         return this.Name;
      }
      set

      {
         this.Name = value;
      }
   }
}

class DBManager
{
   static DBConnection[] activeConnections;
   public void AddConnections()
   {
      activeConnections = new DBConnection[5];
      for (int i = 0; i < 5; i++)
      {
         activeConnections[i] = new
            DBConnection("DBConnection " + (i + 1));
      }
   }

   public delegate void EnumConnectionsCallback(DBConnection connection);
   public static void EnumConnections(EnumConnectionsCallback callback)
   {
      foreach (DBConnection connection in activeConnections)
```

```
        {
           callback(connection);
        }
     }
}

class Delegate3App
{
   public DBManager.EnumConnectionsCallback myCallback
   {
      get
      {
         return new DBManager.EnumConnectionsCallback
                     (ActiveConnectionsCallback);
      }
   }

   public static void ActiveConnectionsCallback(DBConnection connection)
   {
      Console.WriteLine
         ("Callback method called for " + connection.name);
   }

   public static void Main()
   {
      Delegate3App app = new Delegate3App();

      DBManager dbMgr = new DBManager();
      dbMgr.AddConnections();

      DBManager.EnumConnections(app.myCallback);
   }
}
```

11.1.4 Delegate Composition

The ability to compose delegates by creating a single delegate out of multiple delegates—is one of those features that at first does not seem very handy, but if we ever need it we will be happy that the C# design team thought of it. Let us look at some examples in which delegate composition is useful. In the first example, we have a distribution system and a class that iterates through the parts for a given location, calling a callback method for each part that has an "on-hand" value of less than 50. In a more realistic distribution example, the formula would take into account not only "on-hand" but also "on order" and "in transit" in relation to the lead times, and it would subtract the safety stock level, and so on. But let us keep this simple: if a part's on-hand value is less than 50, an exception has occurred.

The twist is that we want two distinct methods to be called if a given part is below stock: we want to log the event, and then we want to email the purchasing manager. So, let us take a look at how we programmatically create a single composite delegate from multiple delegates:

```
using System;
using System.Threading;

class Part
{
   public Part(string sku)
   {
      this.Sku = sku;

      Random r = new Random(DateTime.Now.Millisecond);
      double d = r.NextDouble() * 100;

      this.OnHand = (int)d;
   }

   protected string Sku;
   public string sku
   {
      get
      {
         return this.Sku;
      }
      set
      {
         this.Sku = value;
      }
   }

   protected int OnHand;
   public int onhand
   {
      get
      {
         return this.OnHand;
      }
      set
      {
         this.OnHand = value;
      }
   }
}
```

```
class InventoryManager
{
   protected const int MIN_ONHAND = 50;

   public Part[] parts;
   public InventoryManager()
   {
      parts = new Part[5];
      for (int i = 0; i < 5; i++)
      {
         Part part = new Part("Part " + (i + 1));

         Thread.Sleep(10); // Randomizer is seeded by time.

         parts[i] = part;
         Console.WriteLine("Adding part '{0}' on-hand = {1}",
                       part.sku, part.onhand);
      }
   }

   public delegate void OutOfStockExceptionMethod(Part part);
   public void ProcessInventory(OutOfStockExceptionMethod exception)
   {
      Console.WriteLine("\nProcessing inventory...");
      foreach (Part part in parts)
      {
         if (part.onhand < MIN_ONHAND)
         {
            Console.WriteLine
               ("{0} ({1}) is below minimum on-hand {2}",
               part.sku, part.onhand, MIN_ONHAND);

            exception(part);
         }
      }
   }
}

class CompositeDelegate1App
{
   public static void LogEvent(Part part)
   {
      Console.WriteLine("\tlogging event...");
   }
   public static void EmailPurchasingMgr(Part part)
   {
      Console.WriteLine("\temailing Purchasing manager...");
   }

   public static void Main()
   {
      InventoryManager mgr = new InventoryManager();

      InventoryManager.OutOfStockExceptionMethod LogEventCallback =
         new InventoryManager.OutOfStockExceptionMethod(LogEvent);

      InventoryManager.OutOfStockExceptionMethod
         EmailPurchasingMgrCallback = new
         InventoryManager.OutOfStockExceptionMethod(EmailPurchasingMgr);

      InventoryManager.OutOfStockExceptionMethod
         OnHandExceptionEventsCallback =
         EmailPurchasingMgrCallback + LogEventCallback;

      mgr.ProcessInventory(OnHandExceptionEventsCallback);
   }
}
```

Running this application produces results like the following:

```
Adding part 'Part 1' on-hand = 16
Adding part 'Part 2' on-hand = 98
Adding part 'Part 3' on-hand = 65
Adding part 'Part 4' on-hand = 22
Adding part 'Part 5' on-hand = 70

Processing inventory...
Part 1 (16) is below minimum on-hand 50
   logging event...
   emailing Purchasing manager...
Part 4 (22) is below minimum on-hand 50
   logging event...
   emailing Purchasing manager...
```

Therefore, using this feature of the language, we can dynamically discern which methods comprise a callback method, aggregate those methods into a single delegate, and pass the composite delegate as though it were a single delegate. The runtime will automatically see to it that all of the methods are called in sequence. In addition, we can also remove desired delegates from the composite by using the minus operator.

However, the fact that these methods get called in sequential order does beg one important question: why can't I simply chain the methods together by having each method successively call the next method? In this section's example, where we have only two methods and both are always called as a pair, we could do that. But let us make the example more complicated. Let us say we have several store locations with each location dictating which methods are called. For example, Location1 might be the warehouse, so we would want to log the event and email the purchasing manager, whereas a part that's below the minimum on-hand quantity for all other locations would result in the event being logged and the manager of that store being emailed.

We can easily address these requirements by dynamically creating a composite delegate based on the location being processed. Without delegates, we would have to write a method that not only would have to determine which methods to call, but also would have to keep track of which methods had already been called and which ones were yet to be called during the call sequence. As we can see in the following code, delegates make this potentially complex operation very simple.

```
using System;
class Part
{
  public Part(string sku)

  {
    this.Sku = sku;

    Random r = new Random(DateTime.Now.Millisecond);
    double d = r.NextDouble() * 100;

    this.OnHand = (int)d;
  }

  protected string Sku;
  public string sku
  {
    get
    {
      return this.Sku;
    }
    set
    {
      this.Sku = value;
    }
  }

  protected int OnHand;
  public int onhand
  {
    get
    {
      return this.OnHand;
    }
    set
```

```
        {
            this.OnHand = value;
        }
    }
}

class InventoryManager
{
    protected const int MIN_ONHAND = 50;

    public Part[] parts;
    public InventoryManager()
    {
        parts = new Part[5];
        for (int i = 0; i < 5; i++)
        {
            Part part = new Part("Part " + (i + 1));
            parts[i] = part;
            Console.WriteLine
                ("Adding part '{0}' on-hand = {1}",
                 part.sku, part.onhand);
        }
    }

    public delegate void OutOfStockExceptionMethod(Part part);
    public void ProcessInventory(OutOfStockExceptionMethod exception)
    {
        Console.WriteLine("\nProcessing inventory...");
        foreach (Part part in parts)
        {
            if (part.onhand < MIN_ONHAND)
            {
                Console.WriteLine
                    ("{0} ({1}) is below minimum onhand {2}",
                    part.sku, part.onhand, MIN_ONHAND);

                exception(part);
            }
        }
    }
}

class CompositeDelegate2App
{
    public static void LogEvent(Part part)
```

```
{
   Console.WriteLine("\tlogging event...");
}

public static void EmailPurchasingMgr(Part part)
{
   Console.WriteLine("\temailing Purchasing manager...");
}

public static void EmailStoreMgr(Part part)
{
   Console.WriteLine("\temailing store manager...");
}

public static void Main()
{
   InventoryManager mgr = new InventoryManager();
   InventoryManager.OutOfStockExceptionMethod[] exceptionMethods
      = new InventoryManager.OutOfStockExceptionMethod[3];
   exceptionMethods[0] = new
      InventoryManager.OutOfStockExceptionMethod
         (LogEvent);
   exceptionMethods[1] = new
      InventoryManager.OutOfStockExceptionMethod
         (EmailPurchasingMgr);
   exceptionMethods[2] = new
      InventoryManager.OutOfStockExceptionMethod
         (EmailStoreMgr);

   int location = 1;

   InventoryManager.OutOfStockExceptionMethod compositeDelegate;

   if (location == 2)
   {
      compositeDelegate =
         exceptionMethods[0] + exceptionMethods[1];
   }
   else
   {
      compositeDelegate =
         exceptionMethods[0] + exceptionMethods[2];
   }

   mgr.ProcessInventory(compositeDelegate);
   }
}
```

Now the compilation and execution of this application will yield different results based on the value we assign the *location* variable.

11.2 Events with Delegates

Almost all applications for Windows have some sort of asynchronous event processing needs. Some of these events are generic, such as Windows sending messages to the application message queue when the user has interacted with the application in some fashion. Some are more problem domain"specific, such as printing the invoice for an order being updated.

Events in C# follow the publish-subscribe design pattern in which a class publishes an event that it can "raise" and any number of classes can then subscribe to that event. Once the event is raised, the runtime takes care of notifying each subscriber that the event has occurred. The method called as a result of an event being raised is defined by a delegate. However, keep in mind some strict rules concerning a delegate that's used in this fashion. First, the delegate must be defined as taking two arguments. Second, these arguments always represent two objects: the object that raised the event (the publisher) and an event information object. Additionally, this second object must be derived from the .NET Framework's **EventArgs** class.

Let us say we wanted to monitor changes to inventory levels. We could create a class called *InventoryManager* that would always be used to update inventory. This *InventoryManager* class would publish an event that would be raised any time inventory is changed via such actions as the receipt of inventory, sales, and physical inventory updates. Then, any class needing to be kept updated would subscribe to the event. Here is how this would be coded in C# by using delegates and events:

```
using System;
class InventoryChangeEventArgs : EventArgs
{
   public InventoryChangeEventArgs(string sku, int change)
   {
      this.sku = sku;
      this.change = change;
   }

   string sku;
   public string Sku
   {
      get
      {
         return sku;
      }
   }

   int change;
   public int Change
   {
      get
      {
         return change;
      }
   }
}

class InventoryManager // Publisher.
{
   public delegate void InventoryChangeEventHandler
      (object source, InventoryChangeEventArgs e);
   public event InventoryChangeEventHandler OnInventoryChangeHandler;



   public void UpdateInventory(string sku, int change)
   {
      if (0 == change)
         return; // No update on null change.

      // Code to update database would go here.

      InventoryChangeEventArgs e = new
         InventoryChangeEventArgs(sku, change);

      if (OnInventoryChangeHandler != null)
         OnInventoryChangeHandler(this, e);
   }
}

class InventoryWatcher // Subscriber.
{
   public InventoryWatcher(InventoryManager inventoryManager)
   {
      this.inventoryManager = inventoryManager;
      inventoryManager.OnInventoryChangeHandler += new
InventoryManager.InventoryChangeEventHandler(OnInventoryChange);
   }
   void OnInventoryChange(object source, InventoryChangeEventArgs e)
   {
      int change = e.Change;
      Console.WriteLine("Part '{0}' was {1} by {2} units",
         e.Sku,
         change > 0 ? "increased" : "decreased",
         Math.Abs(e.Change));
   }
   InventoryManager inventoryManager;
}

class Events1App
{
   public static void Main()
   {
      InventoryManager inventoryManager =
         new InventoryManager();

      InventoryWatcher inventoryWatch =
         new InventoryWatcher(inventoryManager);

      inventoryManager.UpdateInventory("111 006 116", -2);
      inventoryManager.UpdateInventory("111 005 383", 5);
   }
}
```

Let us look at the first two members of the *InventoryManager* class:

```
public delegate void InventoryChangeEventHandler
                    (object source, InventoryChangeEventArgs e);
public event InventoryChangeEventHandler OnInventoryChangeHandler;
```

The first line of code is a delegate, which by now we know is a definition for a method signature. As mentioned earlier, all delegates that are used in events must be defined as taking two arguments: a publisher object(in this case, *source*) and an event information object(an *EventArgs*-derived object). The second line uses the *event* keyword, a member type with which we specify the delegate and the method (or methods) that will be called when the event is raised.

The last method in the *InventoryManager* class is the *UpdateInventory* method, which is called anytime inventory is changed. As we can see, this method creates an object of type *InventoryChangeEventArgs*. This object is passed to all subscribers and is used to describe the event that took place.

Now look at the next two lines of code:

```
if (OnInventoryChangeHandler != null)
   OnInventoryChangeHandler(this, e);
```

The conditional *if* statement checks to see whether the event has any subscribers associated with the *OnInventoryChangeHandler* method. If it does—in other words, *OnInventoryChangeHandler* is not *null*—the event is raised. That's really all there is on the publisher side of things. Now let us look at the subscriber code.

The subscriber in this case is the class called *InventoryWatcher*. All it needs to do is perform two simple tasks. First, it adds itself as a subscriber by instantiating a new delegate of type *InventoryManager.InventoryChangeEventHandler* and adding that delegate to the *InventoryManager.OnInventoryChangeHandler* event. Pay special attention to the syntax used—it is using the += compound assignment operator to add itself to the list of subscribers so as not to erase any previous subscribers.

inventoryManager.OnInventoryChangeHandler
+= new InventoryManager.InventoryChangeEventHandler(OnInventoryChange);

The only argument that needs to be supplied here is the name of the method that will be called if and when the event is raised.

The only other task the subscriber needs to do is implement its event handler. In this case, the event handler is *InventoryWatcher.OnInventoryChange*, which prints a message stating the part number and the change in inventory.

Finally, the code that runs this application instantiates *InventoryManager* and *InventoryWatcher* classes and, every time the *InventoryManager.UpdateInventory* method is called, an event is automatically raised that causes the *InventoryWatcher.OnInventoryChanged* method to be called.

Review Questions

1. What are Delegates? Expalin
2. How will you use delegates as callback Method?

Chapter 12: Multithreading

12.1 Multithreading

A common question that comes to our mind when we are reminded of threads is that what does this thread exactly means. A Thread is single unit of program execution. Any other program at a lower level cannot interrupt a thread and it does not contain smaller units. The example of a single threaded application is our MS-DOS. If a number of different threads are doing different tasks in the application, then it is a multithreaded application. Threads are not specific to C#; the general subject of multithreading is one most programmers should be familiar with when learning this new language. We will discuss the basics and even some intermediate-to-advanced issues regarding the aborting, scheduling, and lifetime management of threads. We will also cover thread synchronization with the *System.Monitor* and *System.Mutex* classes and the C# *lock* statement.

12.1.1 Threading Basics

Having understood the meaning of multithreading we should know what it means in the actual programming. Multithreading is the most efficient way to do more things at a time in this modern computer era. Unless there is multiple processor there is remote chance that any machine doing two or more tasks at a time. Even with fast processors and time slice algorithms a machine appears to do numerous tasks. Multithreading allows an application to divide tasks such that they work independently of each other to make the most efficient use of the processor and the user's time. Threading is not the right choice for all applications and can sometimes even slow an application down. Hence it is necessary to learn the syntax of multithreading and also understand when to use it.

12.1.2 Threads and Multitasking

Very early versions of Microsoft Windows supported cooperative multitasking, which meant that each thread was responsible for relinquishing control to the processor so that it could process other threads. A thread is a unit of processing, and multitasking is the simultaneous execution of multiple threads. Multitasking comes in two flavors: cooperative and preemptive.

With preemptive multitasking, the processor is responsible for giving each thread a certain amount of time in which to execute—a *timeslice*. The processor then switches among the different threads, giving each its timeslice, and the programmer does not have to worry about how and when to relinquish control so that other threads can run. Because .NET will only work only on preemptive multitasking operating systems, this is what I'll be focusing on. Even with preemptive multitasking, if We're running on a single processor machine, We don't really have multiple threads executing at the same time. Because the processor is switching between processes at intervals that number in the milliseconds, it just "feels" that way. If We want to run true multiple threads concurrently, We will need to develop and run Wer code on a machine with multiple processors.

12.1.3 Context Switching

The processor uses a hardware timer to determine when a timeslice has ended for a given thread. When the hardware timer signals the interrupt, the processor saves all registers for the current thread onto the stack. Then the processor moves those same registers from the stack into a data structure called a *CONTEXT* structure. When the processor wants to switch back to a previously executing thread, it reverses this procedure and restores the registers from the *CONTEXT* structure associated with the thread. This entire procedure is called context switching.

12.2 Using Multithreading in C#

Consider compiling and executing this application below , We will see that the message from the *Main* method prints before the message from the Skilled thread, proving that the Skilled thread is indeed working asynchronously.

```
using System;
using System.Threading;

class SimpleThreadApp
{
   public static void SkilledThreadMethod ()
   {
      Console.WriteLine("Skilled thread started");
   }

   public static void Main()
   {ThreadStart Skilled = new ThreadStart(SkilledThreadMethod);

      Console.WriteLine("Main - Creating Skilled thread");

      Thread t = new Thread (Skilled);
      t. Start ();

   Console.WriteLine
      ("Main - Have requested the start of Skilled thread");
   }
}.
```

The using statement for the *System.Threading* namespace. We will get to know that namespace shortly. For now, it is enough to understand that this namespace contains the different classes necessary for threading in the .NET environment. Now take a look at the first line of the *Main* method:

```
ThreadStart SkilledThreadMethod = new ThreadStart (SkilledThreadMethod);
```

Any time we see the following form, We can be sure that *x* is a delegate or as We a definition of a method signature:

```
x varName = new x (methodName);
```

Hence we know that *ThreadStart* is a delegate. But it is not just any delegate. Specifically, it is the delegate that must be used when creating a new thread and it is used to specify the method that we want called as the thread method. From there, we instantiate a *Thread* object where the constructor takes as its only argument a *ThreadStart* delegate, like so

Thread t = new Thread (Skilled);

Subsequently we call the Thread object's *Start* method, which results in a call to the *SkilledThreadMethod*.

Three lines of code to set up and run the thread and the thread method itself, and we're off and running. Let us ponder into the *System.Threading* namespace and its classes that make all this happen.

Working with Threads

The management of threads is achieved through use of the *System.Threading.Thread* class.

12.2.1 AppDomain

The .NET, threads run in something called an *AppDomain*. A thread in one process cannot invoke a method in a thread that belongs to another process. In .NET, however, threads can cross *AppDomain* boundaries, and a method in one thread can call a method in another *AppDomain*. We will sometimes hear that an *AppDomain* is analogous to a Win32 process in that it offers many of the same benefits, including fault tolerance and the ability to be independently started and stopped. This is a good comparison, but the comparison breaks down in relation to threads.

12.2.2 The *Thread* Class

We can instantiate a *Thread* object in two ways. We've already seen one way: creating a new thread and, in that process, getting a *Thread* object with which to manipulate the new thread. The other way to obtain a *Thread* object is by calling the static *Thread.CurrentThread* method for the currently executing thread.

12.2.3 Managing Thread Lifetimes

There are many different tasks we might need to perform to manage the activity or life of a thread. We can manage all these tasks by using the different *Thread* methods. For example, it is quite common to need to pause a thread for a given period of time. To do this, we can call the *Thread.Sleep* method. This method takes a single argument that represents the amount of time, in milliseconds, that we want the thread to pause. Note that the *Thread.Sleep* method is a static method and cannot be called with an instance of a *Thread* object. There's a very good reason for this. We're not allowed to call *Thread.Sleep* on any other thread except the currently executing one. The static *Thread.Sleep* method calls the static *CurrentThread* method, which then pauses that thread for the specified amount of time. Here's an example:

```
using System;
using System.Threading;

class ThreadMulApp
{
   public static void SkilledThreadMethod ()
```

```
    {
        Console.WriteLine ("Skilled thread started");

        int sleepTime = 5000;

        Console.WriteLine("\tsleeping for {0} seconds", sleepTime / 1000);
        Thread.Sleep(sleepTime); // Sleep for five seconds.
        Console.WriteLine("\twaking up");
    }

    public static void Main ()
    {
        ThreadStart Skilled = new ThreadStart(SkilledThreadMethod);

        Console.WriteLine("Main - Creating Skilled thread");

        Thread t = new Thread (Skilled);
        t. Start ();

        Console.WriteLine
            ("Main - Have requested the start of Skilled thread");
    }
}
```

There are two more ways to call the *Thread.Sleep* method. First, by calling *Thread.Sleep* with the value *0*, we will cause the current thread to relinquish the unused balance of its timeslice. Passing a value of *Timeout.Infinite* results in the thread being paused indefinitely until the suspension is interrupted by another thread calling the suspended thread's *Thread.Interrupt* method.

The second way to suspend the execution of a thread is by using the *Thread.Suspend* method. There are some major differences between the two techniques. First, the *Thread.Suspend* method can be called on the currently executing thread or another thread. Second, once a thread is suspended in this fashion, only another thread can cause its resumption, with the *Thread.Resume* method. Note that once a thread suspends another thread, the first thread is not blocked. The call returns immediately. Also, regardless of how many times the *Thread.Suspend* method is called for a given thread, a single call to *Thread.Resume* will cause the thread to resume execution.

12.2.4 Destroying Threads

We can accomplish the need to destroy a thread, with a call to the *Thread.Abort* method. The runtime forces the abortion of a thread by throwing a *ThreadAbortException*. Even if the method attempts to catch the *ThreadAbortException*, the runtime won't let it. However, the runtime will execute the code in the aborted thread's *finally* block, if one's present. The *Main* method pauses for two seconds to make sure that the runtime has had time to start the Skilled thread. Upon being started, the Skilled thread starts counting to ten, pausing for a second in between each number. When the *Main* method resumes execution after its two-second pause, it aborts the Skilled thread. Notice that the *finally* block is executed after the abort.

```
using System;
using System.Threading;

class ThreadAbortApp
{
   public static void SkilledThreadMethod ()
   {
      try
      {
         Console.WriteLine("Skilled thread Moved");

         Console.WriteLine
            ("Skilled thread - counting slowly to 100");
         for (int i = 0; i < 100; i++)
         {
            Thread.Sleep (500);
            Console.Write("{0}...", i);
         }

         Console.WriteLine("Skilled thread reached");
      }
      catch (ThreadAbortException e)
      {
      }
      finally
      {
         Console.WriteLine
            ("Skilled thread -
             Can't catch the exception, but I can Move it");
      }
   }

   public static void Main ()
   {ThreadStart Skilled = new ThreadStart (SkilledThreadMethod);

      Console.WriteLine("Main - Started Skilled thread");

      Thread t = new Thread (Skilled);
      t. Start ();

      // Give the Skilled thread time to start.
      Console.WriteLine("Main - Sleeping for 34 seconds");
      Thread.Sleep (2000);

      Console.WriteLine("\nMain - Aborting Skilled thread");
      t. Abort ();
```

```
    }
}
```

When we compile and execute this application, the following output results:

```
Main – Started Skilled thread
Main - Sleeping for 34 seconds
Skilled thread started
Skilled thread - counting slowly to 100
0...1...2...3...
Main - Aborting Skilled thread
Skilled thread - can't catch the exception.
```

When the *Thread.Abort* method is called, the thread will not cease execution immediately. The runtime waits until the thread has reached what the documentation describes as a "safe point." Therefore, if the code is dependent on something happening after the abort and we must be sure the thread has stopped, we can use the *Thread.Join* method. This is a synchronous call, meaning that it will not return until the thread has been stopped. Lastly, note that once we abort a thread, it cannot be restarted. In that case, although we have a valid *Thread* object, we can't do anything useful with it in terms of executing code.

12.2.5 Scheduling Threads

Each thread has an associated priority level that tells the processor how it should be scheduled in relation to the other threads in the system. When the processor switches between threads once a given thread's timeslice has ended, the process of choosing which thread executes next is far from arbitrary. This priority level is defaulted to *Normal*—more on this shortly—for threads that are created within the runtime. For threads that are created outside the runtime, they retain their original priority. We use the *Thread.Priority* property to view and set this value. The *Thread.Priority* property's setter takes a value of type *Thread.ThreadPriority* that is an *enum* that defines these values: *Highest*, *AboveNormal*, *Normal*, *BelowNormal*, and *Lowest.*

Take a look at the following example to understand how priorities can affect even the simplest code, in which one Skilled thread counts from 1 to 10 and the other counts from 11 to 20. Note the nested loop within each *SkilledThread* method. Each loop is there to represent work the thread would be doing in a real application. Because these methods don't really do anything, not having those loops would result in each thread finishing its work in its first timeslice!

```
using System;
using System.Threading;

class ThreadMulApp
{
   public static void SkilledThreadMethod1()
   {
      Console.WriteLine("Skilled thread started");

      Console.WriteLine
         ("Skilled thread - counting slowly from 1 to 10");
      for (int i = 1; i < 11; i++)
```

```
    {
        for (int j = 0; j < 100; j++)
        {
            Console.Write(".");
            // Code to imitate work being done.
            int a;
            a = 15;
        }
        Console.Write("{0}", i);
    }

    Console.WriteLine("Skilled thread finished");
}

public static void SkilledThreadMethod2()
{
    Console.WriteLine("Skilled thread started");

    Console.WriteLine
        ("Skilled thread - counting slowly from 11 to 20");
    for (int i = 11; i < 20; i++)
    {
        for (int j = 0; j < 100; j++)
        {
            Console.Write(".");
            // Code to imitate work being done.
            int a;
            a = 15;
        }
        Console.Write("{0}", i);
    }

    Console.WriteLine("Skilled thread finished");
}

public static void Main()
{
    ThreadStart Skilled1 = new ThreadStart(SkilledThreadMethod1);
    ThreadStart Skilled2 = new ThreadStart(SkilledThreadMethod2);

    Console.WriteLine("Main - Creating Skilled threads");

    Thread t1 = new Thread(Skilled1);
    Thread t2 = new Thread(Skilled2);
```

```
        t1.Start();
        t2.Start();
    }
}
```

Running this application results in the following output. Note that in the interest of brevity I've truncated the output—most of the dots are removed.

```
Main - Creating Skilled threads
Skilled thread started
Skilled thread started
Skilled thread - counting slowly from 1 to 10
Skilled thread - counting slowly from 11 to 20
...1...11...2...12...3...13
```

Both threads are getting equal playing time with the processor. Now alter the *Priority* property for each thread as in the following code—I've given the first thread the highest priority allowed and the second thread the lowest—and we will see a much different result.

```
using System;
using System.Threading;

class ThreadMul2App
{
    public static void SkilledThreadMethod1()
    {
        Console.WriteLine ("Skilled thread started");

        Console.WriteLine
            ("Skilled thread - counting slowly from 1 to 10");
        for (int i = 1; i < 11; i++)
        {
            for (int j = 0; j < 100; j++)
            {               Console.Write(".");
                // Code to imitate work being done.
                int a;
                a = 15;
            }

            Console.Write("{0}", i);
        }

        Console.WriteLine("Skilled thread finished");
    }

    public static void SkilledThreadMethod2()
    {
        Console.WriteLine("Skilled thread started");
        Console.WriteLine
            ("Skilled thread - counting slowly from 11 to 20");
        for (int i = 11; i < 20; i++)
        {
            for (int j = 0; j < 100; j++)
            {
                Console.Write(".");
                // Code to imitate work being done.
                int a;
                a = 15;
            }
            Console.Write("{0}", i);
        }

        Console.WriteLine("Skilled thread finished");
    }

    public static void Main()
    {
        ThreadStart Skilled1 = new ThreadStart(SkilledThreadMethod1);
        ThreadStart Skilled2 = new ThreadStart(SkilledThreadMethod2);

        Console.WriteLine("Main - Creating Skilled threads");

        Thread t1 = new Thread(Skilled1);
        Thread t2 = new Thread(Skilled2);

        t1.Priority = ThreadPriority.Highest;
        t2.Priority = ThreadPriority.Lowest;

        t1.Start();
        t2.Start();
    }
}
```

The results should be as shown below. Notice that the second thread has been set to such a low priority that it does not receive any cycles until after the first thread has completed its work.

```
Main - Creating Skilled threads
Skilled thread started
Skilled thread started
Skilled thread - counting slowly from 1 to 10
Skilled thread - counting slowly from 11 to 20
......1......2......3......4......5......6......7......8......9......10......
11......12......13......14......15......16......17......18......19......20
```

Remember that when we tell the processor the priority we want a given thread to have, it is the operating system that eventually uses this value as part of its scheduling algorithm that it relays to the processor. All these values are used to create a numeric value that represents the priority of the thread having the highest value is the thread with the highest priority.

One last note on the subject of thread scheduling: use it with caution. Let us say we have a GUI application and a couple of Skilled threads that are off doing some sort of asynchronous work. If we set the Skilled thread priorities too high, the UI might become sluggish because the main thread in which the GUI application is running is receiving fewer CPU cycles. Unless we have a specific reason to schedule a thread with a high priority, it is best to let the thread's priority default to *Normal*.

12.2.6 Thread Safety and Synchronization

When programming for a single-threaded environment, it is common to write methods in such a way that at several points in the code the object is in a temporarily invalid state. Obviously, if only one thread is accessing the object at a time, we're guaranteed that each method will complete before another method is called—meaning that the object is always in a valid state to any of the object's clients. However, when multiple threads are thrown into the mix, we can easily have situations in which the processor switches to another thread while the object is in an invalid state. If that thread then also attempts to use this same object, the results can be quite unpredictable. Therefore, the term "thread safety" means that the members of an object always maintain a valid state when used concurrently by multiple threads.

How do we prevent this unpredictable state? Actually, as is common in programming, there are several ways to address this well-known issue. In this section, we will cover the most common means: synchronization. Through synchronization, we specify **critical sections** of code that can be entered by only one thread at a time, thereby guaranteeing that any temporary invalid states of Wer object are not seen by the object's clients.

12.2.7 Protecting Code by Using the Monitor Class

The *System.Monitor* class enables us to serialize the access to blocks of code by means of locks and signals. For example, we have a method that updates a Datacenter and that cannot be executed by two or more threads at the same time. If the work being performed by this method is especially time-consuming and we have multiple threads, any of which might call this method, we could have a serious problem on the hands. This is where the *Monitor* class comes in. Take a look at the following synchronization example. Here we have two threads, both of which will call the *Datacenter.SaveData* method.

```
using System;
using System.Threading;

class Datacenter
{
   public void SaveData(string text)
   {
      Console.WriteLine("Datacenter.SaveData - Started");

      Console.WriteLine("Datacenter.SaveData - Working");
      for (int i = 0; i < 100; i++)
      {
         Console.Write(text);
```

```
        }

        Console.WriteLine("\nDatacenter.SaveData - Ended");
    }
}

class ThreadMonitor1App

{
    public static Datacenter db = new Datacenter();

    public static void SkilledThreadMethod1()
    {
        Console.WriteLine("Skilled thread #1 - Started");

        Console.WriteLine
            ("Skilled thread #1 -Calling Datacenter.SaveData");
        db.SaveData("x");

        Console.WriteLine("Skilled thread #1 - Returned from Output");
    }

    public static void SkilledThreadMethod2()
    {
        Console.WriteLine("Skilled thread #2 - Started");

        Console.WriteLine
            ("Skilled thread #2 - Calling Datacenter.SaveData");
        db.SaveData("o");

        Console.WriteLine("Skilled thread #2 - Returned from Output");
    }

    public static void Main()
    {
        ThreadStart Skilled1 = new ThreadStart(SkilledThreadMethod1);
        ThreadStart Skilled2 = new ThreadStart(SkilledThreadMethod2);

        Console.WriteLine("Main - Creating Skilled threads");

        Thread t1 = new Thread(Skilled1);
        Thread t2 = new Thread(Skilled2);

        t1.Start();
        t2.Start();
```

```
   }
}
```

When we compile and execute this application, the resulting output will have a mixture of o's and x's, showing that the *Datacenter.SaveData* method is being run concurrently by both threads. (Note that once again I've abbreviated the output.)

```
Main - Creating Skilled threads

Skilled thread #1 - Started
Skilled thread #2 - Started

Skilled thread #1 - Calling Datacenter.SaveData
Skilled thread #2 - Calling Datacenter.SaveData

Datacenter.SaveData - Started
Datacenter.SaveData - Started

Datacenter.SaveData - Working
Datacenter.SaveData - Working
Xoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxoxox
Datacenter.SaveData - Ended
Datacenter.SaveData - Ended

Skilled thread #1 - Returned from Output
Skilled thread #2 - Returned from Output
```

Obviously, if the *Datacenter.SaveData* method needed to finish updating multiple tables before being called by another thread, we would have a serious problem.

To incorporate the *Monitor* class in this example, we use two of its static methods. The first method is called *Enter*. When executed, this method attempts to obtain a monitor lock on the object. If another thread already has the lock, the method will block until that lock has been released. Note that there is no implicit box operation performed here, so We can supply only a reference type to this method. The *Monitor.Exit* method is then called to release the lock. Here's the example rewritten to force the serialization of access to the *Datacenter.SaveData* method:

```
using System;
using System.Threading;
class Datacenter
{
   public void SaveData (string text)
   {
      Monitor. Enter (this);

      Console.WriteLine ("Datacenter.SaveData - Started");

      Console.WriteLine ("Datacenter.SaveData - Working");
      for (int i = 0; i < 100; i++)
      {
            Console.Write (text);
        }

        Console.WriteLine ("\nDatacenter. SaveData - Ended");

        Monitor.Exit (this);
    }
}

class ThreadMonitor2App
{
    public static Datacenter db = new Datacenter ();

    public static void SkilledThreadMethod1 ()
    {
        Console.WriteLine ("Skilled thread #1 - Started");

        Console.WriteLine
          ("Skilled thread #1 - Calling Datacenter.SaveData");
        db. SaveData ("x");

        Console.WriteLine("Skilled thread #1 - Returned from Output");
    }

    public static void SkilledThreadMethod2()
    {
        Console.WriteLine("Skilled thread #2 - Started");

        Console.WriteLine
            ("Skilled thread #2 - Calling Datacenter.SaveData");
        db.SaveData("o");

        Console.WriteLine("Skilled thread #2 - Returned from Output");
    }

    public static void Main()
    {
        ThreadStart Skilled1 = new ThreadStart(SkilledThreadMethod1);
        ThreadStart Skilled2 = new ThreadStart(SkilledThreadMethod2);

        Console.WriteLine("Main - Creating Skilled threads");

        Thread t1 = new Thread(Skilled1);
        Thread t2 = new Thread(Skilled2);
        t1.Start();
        t2.Start();
    }
}
```

Notice in the following output that even though the second thread called the *Datacenter.SaveData* method, the *Monitor.Enter* method caused it to block until the first thread had released its lock:

```
Main - Creating Skilled threads

Skilled thread #1 - Started
Skilled thread #2 - Started

Skilled thread #1 - Calling Datacenter.SaveData
Skilled thread #2 - Calling Datacenter.SaveData

Datacenter.SaveData - Started
Datacenter.SaveData - Working
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Datacenter.SaveData - Ended

Datacenter.SaveData - Started

Skilled thread #1 - Returned from Output

Datacenter.SaveData - Working
ooooooooooooooooooooooooooooooooooooooo
Datacenter.SaveData - Ended

Skilled thread #2 - Returned from Output
```

12.2.8 Using Monitor Locks with the C# lock Statement

Although the C# *lock* statement does not support the full array of features found in the **Monitor** class, it does enable us to obtain and release a monitor lock. To use the **lock** statement, simply specify the *lock* statement with the code being serialized in braces. The braces indicate the starting and stopping point of the code being protected, so there's no need for an ***unlock*** statement. The following code will produce the same synchronized output as the previous examples:

```
using System;
using System.Threading;

class Datacenter
{
    public void SaveData(string text)
    {
        lock(this)
        {
          Console.WriteLine("Datacenter.SaveData - Started");

          Console.WriteLine("Datacenter.SaveData - Working");
          for (int i = 0; i < 100; i++)
          {
            Console.Write(text);
          }

          Console.WriteLine("\nDatacenter.SaveData - Ended");
        }
    }
}

class ThreadLockApp
{
    public static Datacenter db = new Datacenter();

    public static void SkilledThreadMethod1()
    {
        Console.WriteLine("Skilled thread #1 - Started");

        Console.WriteLine
          ("Skilled thread #1 - Calling Datacenter.SaveData");
        db.SaveData("x");

        Console.WriteLine("Skilled thread #1 - Returned from Output");
    }

    public static void SkilledThreadMethod2()
    {
        Console.WriteLine("Skilled thread #2 - Started");

        Console.WriteLine
          ("Skilled thread #2 - Calling Datacenter.SaveData");
        db.SaveData("o");

        Console.WriteLine("Skilled thread #2 - Returned from Output");
    }

    public static void Main()
    {
        ThreadStart Skilled1 = new ThreadStart(SkilledThreadMethod1);
        ThreadStart Skilled2 = new ThreadStart(SkilledThreadMethod2);
        Console.WriteLine("Main - Creating Skilled threads");

        Thread t1 = new Thread(Skilled1);
        Thread t2 = new Thread(Skilled2);

        t1.Start();
        t2.Start();
    }
}
```

12.2.9 Synchronizing Code by Using the Mutex Class

The term **mutex** comes from **mutually exclusive**, and just as only one thread at a time can obtain a monitor lock for a given object, only one thread at a time can obtain a given mutex. The *Mutex* class—defined in the *System.Threading* namespace—is a run-time representation of the Win32 system primitive of the same name. We can use a mutex to serialize access to code as we can use a monitor lock, but mutexes are much slower because of their increased flexibility. We can create a mutex in C# with the following three constructors:

```
Mutex( )
Mutex(bool initiallyOwned)
Mutex(bool initiallyOwned, string mutexName)
```

The first constructor creates a mutex with no name and makes the current thread the owner of that mutex. Therefore, the mutex is locked by the current thread. The second constructor takes only a Boolean flag that designates whether the thread creating the mutex wants to own it (lock it). And the third constructor allows We to specify whether the current thread owns the mutex and to specify the name of the mutex. Let us now incorporate a mutex to serialize access to the *Datacenter.SaveData* method:

```
using System;
using System.Threading;

class Datacenter
{
    Mutex mutex = new Mutex(false);

    public void SaveData(string text)
    {
        mutex.WaitOne();
        Console.WriteLine("Datacenter.SaveData - Started");

        Console.WriteLine("Datacenter.SaveData - Working");
        for (int i = 0; i < 100; i++)
        {
            Console.Write(text);
        }
    Console.WriteLine("\nDatacenter.SaveData - Ended");

    mutex.Close();
  }
}

class ThreadMutexApp
{
  public static Datacenter db = new Datacenter();

  public static void SkilledThreadMethod1()
  {
    Console.WriteLine("Skilled thread #1 - Started");

    Console.WriteLine
      ("Skilled thread #1 - Calling Datacenter.SaveData");
    db.SaveData("x");

    Console.WriteLine("Skilled thread #1 - Returned from Output");
  }

  public static void SkilledThreadMethod2()
  {
    Console.WriteLine("Skilled thread #2 - Started");

    Console.WriteLine
      ("Skilled thread #2 - Calling Datacenter.SaveData");
    db.SaveData("o");

    Console.WriteLine("Skilled thread #2 - Returned from Output");
  }

  public static void Main()
  {
    ThreadStart Skilled1 = new ThreadStart(SkilledThreadMethod1);
    ThreadStart Skilled2 = new ThreadStart(SkilledThreadMethod2);

    Console.WriteLine("Main - Creating Skilled threads");

    Thread t1 = new Thread(Skilled1);
    Thread t2 = new Thread(Skilled2);

    t1.Start();
    t2.Start();
  }
```

}

Now, the *Datacenter* class defines a *Mutex* field. We don't want the thread to own the mutex just yet because we would have no way of getting into the *SaveData* method. The first line of the *SaveData* method shows we how We need to attempt to acquire the mutex—with the *Mutex.WaitOne* method. At the end of the method is a call to the *Close* method, which releases the mutex.

The *WaitOne* method is also overloaded to provide more flexibility in terms of allowing we to define how much time the thread will wait for the mutex to become available. Here are those overloads:

WaitOne ()

WaitOne(TimeSpan *time*, bool *exitContext*)

WaitOne (int milliseconds, bool exitContext)

The basic difference between these overloads is that the first version—used in the example—will wait indefinitely, and the second and third versions will wait for the specified amount of time, expressed with either a *TimeSpan* value or an *int* value.

12.3 When to Use Threads

We should use threads when we are striving for increased concurrency, simplified design, and better utilization of CPU time, as described in the following sections.

Increased Concurrency

Very often applications need to accomplish more than one task at a time. For example, when we write a document retrieval system for banks that accessed data from optical disks that were stored in optical disk jukeboxes. Imagine the massive amounts of data we are talking about here; picture a jukebox with one drive and 50 platters serving up gigabytes of data. It could sometimes take as long as 5 to 10 seconds to load a disk and find the requested document. Needless to say, it would not exactly be the definition of productivity if my application blocked user input while it did all this. Therefore, to deal with the user input case, I spun off another thread to do the physical work of retrieving the data, allowing the user to continue working. This thread would notify the main thread when the document was loaded. This is a great example of having independent activities—loading a document and handling the UI—that can be handled with two separate threads.

Simplified Design

A popular way to simplify the design of complex systems is to use queues and asynchronous processing. Using such a design, we would have queues set up to handle the different events that transpire in Wer system. Instead of methods being called directly, objects are created and placed in queues where they will be handled. At the other end of these queues are server programs with multiple threads that are set up to "listen" for messages coming in to these queues. The advantage of this type of simplified design is that it provides for reliable, robust, and extendable systems.

Better Utilization of CPU Time

Many times Wer application isn't really doing any work while it is still enjoying its timeslice. In my document retrieval example, one thread was waiting on the jukebox to load the platter. Obviously, this wait was a hardware issue and required no use of the CPU. Other examples of wait times include when we're printing a document or waiting on the hard disk or CD-ROM drive. In each case, the CPU is not being utilized. Such cases are candidates for being moved to background threads.

12.4 When Not to Use Threads

It is a common mistake for those new to threads to attempt to deploy them in every application. This can be much worse than not having them at all! As with any other tool in Wer programming arsenal, we should use threads only when it is appropriate to do so. We should avoid using multiple threads in Wer application in at least the following cases when costs outweigh benefits, when we haven't benchmarked both cases, and when we can't come up with a reason to use threads.

If we are new to multithreaded programming, it might surprise us to find that often the overhead required by CPU thread creation and scheduling to achieve modest gains in CPU utilization can actually result in single-threaded applications running faster! It all depends on what we are doing and if we are truly splitting independent tasks into threads. For example, if we have to read three files from disk, spawning three threads we dont do we any good because each has to use the same hard disk. Therefore, always be sure to benchmark a prototype of both a single-threaded and multithreaded version of the system before going through the extra time and cost of designing around a solution that might actually backfire in terms of performance.

Review Questions

1. Explain Multi threading?
2. What are the types of threads in C#. when will you use them?
3. What are the thread sysnchronising class in C#?
4. What do understand by context switching?
5. Explain briefly
   Destroying threads.
   Scheduling threads.

6. How will you use lock statement in C#?

UNIT - III

Chapter 13. Writing application with C#

13.1 Visual Studio .NET IDE

Visual Studio .NET is the comprehensive tool set for rapidly building Microsoft Windows based applications and Web solutions. In addition to providing intuitive visual designers, intelligent code editors, and rich data tools, Visual Studio .NET delivers an open tools development environment that supports enhancements and customization at every level of the product.

13.1.1An Open Tools Platform for Developers

Developers integrating products with Visual Studio .NET recognizes tangible benefits for themselves and their customers. Imagine a shorter time-to-market by focusing more time on your company's core value-add instead of building the **integrated development environment** (IDE). Imagine improvement in overall product quality and being able to focus more on testing internally developed code by building on the stable set of reliable IDE components in Visual Studio .NET.

Seamless integration and a consistent user interface (UI) is a boon to customers. Even better, the whole is greater than the sum of the parts: a developer who combines Visual Studio .NET with a modeling tool, a profiling tool, a legacy transaction interface, and a life cycle management solution can now work with a single environment that meets all development needs. The end user benefits by receiving a single, cohesive development environment for all aspects of the software development life cycle.

Currently, Visual Studio .NET integrators include development life-cycle tools vendors, programming language vendors, independent software vendors (ISVs), system integrators, corporations, and academics.Developers can customize the Visual Studio .NET IDE by writing macros, wizards, and add-ins that build upon the Visual Studio .NET automation model. With the addition of the free Visual Studio Help Integration Kit (VSHIK), authors and developers can extend help content, including adding new Dynamic Help.

13.1.2 Macros

Macros provide the fastest, easiest way to customize, extend, and automate the Visual Studio .NET IDE. Developers can start macro development by recording a sequence of actions, and then enhance the macro by writing additional Microsoft Visual Basic .NET code.

13.1.3 Add-ins and Wizards

In addition to macros, the automation model can be accessed by creating extensions to the IDE known as add-ins, which are compiled applications that manipulate the environment and automate tasks. Visual Studio .NET includes extensibility projects that help you write, build, and deploy add-ins.

13.2 Visual Studio Help Integration Kit

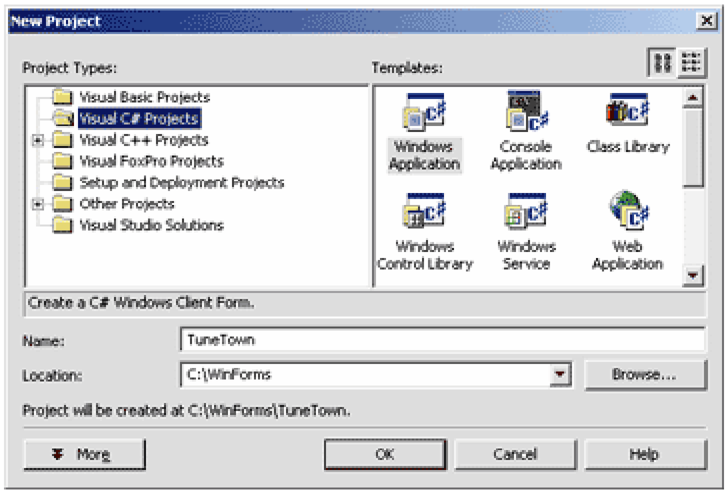
The Visual Studio .NET Help Integration Kit provides documentation and tools for software developers and Help authors to extend the help content in Visual Studio .NET enables component vendors, book authors, and others to provide documentation for components, add-ins, form elements, and libraries, or to extend or enhance the Help for Visual Studio .NET itself. We can Download the Visual Studio .NET Help Integration Kit from MSDN.

Visual Studio .NET is a complete set of development tools for building ASP Web applications, XML Web services, desktop applications, and mobile applications. Visual Basic .NET, Visual C++ .NET, and Visual C# .NET all use the same integrated development environment (IDE), which allows them to share tools and facilitates in the creation of mixed-language solutions. In addition, these languages leverage the functionality of the .NET Framework, which provides access to key technologies that simplify the development of ASP Web applications and XML Web services.

13.2.1 Creating a Windows Forms application

Step 1: Create a New Windows Forms Project

Choose the New Project command from the Visual Studio® File menu to create a new project. Choose Windows Application from the C# Projects folder as the project type. For the project name, enter TuneTown.



Create a C# Windows Client Form.

Name: TuneTown

Location: C:\WinForms

Project will be created at C:\WinForms\TuneTown.

Fig 2

Step 2: Design the Main Form

Once the new project is created, Visual Studio will drop you into the Visual Studio forms editor and present you with a blank form. Before you begin work on the form, change the file name Main1.cs to MainForm.cs in the Solution Explorer window. While you are at it, use the Class View window to change the form class's name from Form1 to MainForm.

Now go back to the forms editor and add a list view control and three push buttons to the form, as shown in **Figure**. Then, one by one, select each of the controls you added and use the Properties window to modify the control properties as described in the following paragraphs.

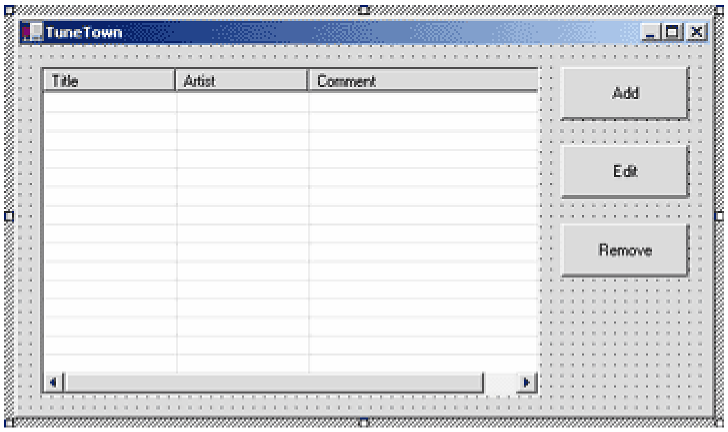


Fig3

For the list view control, edit the control properties as follows:

Set the FullRowSelect property to True.

Set the GridLines property to True.

Set the View property to Report.

Edit the Columns collection to add three columns to the header at the top of the list view: one whose Name is "TitleHeader", whose Text is "Title", and whose Width is 100; another whose Name is "ArtistHeader", Text is "Artist", and Width is 100; and a third whose Name is "CommentHeader", Text is "Comment", and Width is 200.

Set Multiselect to False.

Set HideSelection to False.

Set Sorting to Ascending.

Set TabIndex to 0.

Set Name to "TuneView".

Edit the properties of the push button controls as follows:

Set the Text property to "&Add", "&Edit", and "&Remove", respectively.

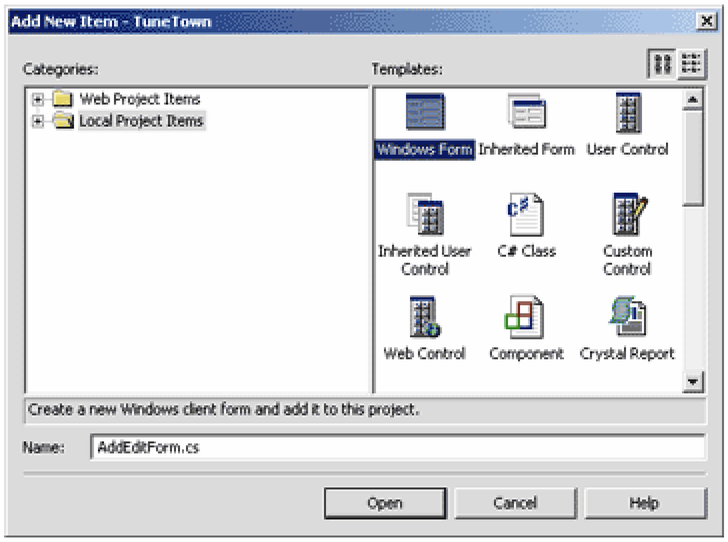
Set the Name property to "AddButton", "EditButton", and "RemoveButton", respectively.

Set the TabIndex property to 1, 2, and 3, respectively.

Finally, change the Text property of the form itself to "TuneTown". Doing so will change the title in the form's caption bar.

Step 3: Add Another Form

We need a second form that you can use to solicit input when the user clicks the Add or Edit button. In effect, this form will serve as a dialog box. To add the form to the project, go to the Project menu, select the Add Windows Form command, and choose Windows Form from the ensuing dialog



Create a new Windows client form and add it to this project.

Name: AddEditForm.cs

Fig4

In the Name box, type AddEditForm.cs.
Step 4: Design the Form
Edit the new form in the Visual Studio forms editor, so that it resembles the one shown in **Figure**. Modify the label controls' properties as follows:

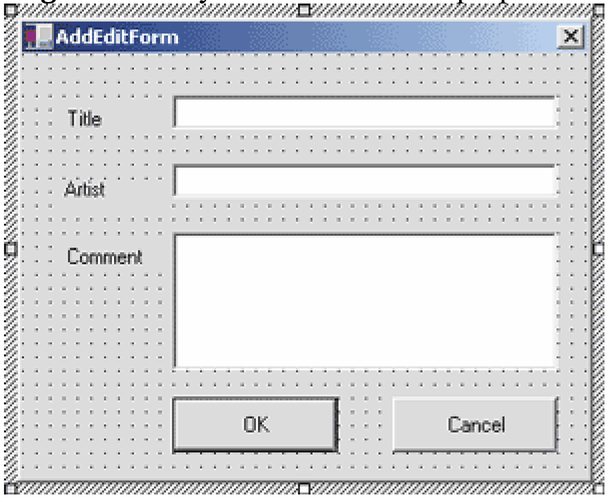


Fig5
Set the Text property to "&Title", "&Artist", and "&Comment", respectively.
Set the Name property to "TitleLabel", "ArtistLabel", and "CommentLabel", respectively.
Set TabIndex to 0, 2, and 4, respectively.
Modify the edit controls' properties this way:
Set all three controls' Text properties to null.
Set Name to "TitleBox", "ArtistBox", and "CommentBox", respectively.
Set TabIndex to 1, 3, and 5, respectively.
Change the third edit control's Multiline property from False to True.
Next, modify the properties of the two push buttons:
Set Text to "OK" and "Cancel", respectively.

Set DialogResult to OK and Cancel, respectively.
Set Name to "OKButton" and "NotOKButton", respectively.
Set TabIndex to 6 and 7, respectively.

Finally, edit the properties of the form itself:

Set BorderStyle to FixedDialog.
Set AcceptButton to OKButton.
Set CancelButton to NotOKButton.
Set MaximizeBox and MinimizeBox to False.
Set ShowInTaskbar to False.

The forms are complete; now it is time to write some code.

Step 5: Add Properties to AddEditForm

Open AddEditForm.cs and add the statements shown in Figure . The bulk of the code shown in Figure was created by Visual Studio. That code performs three important tasks:

It declares Button (System.WinForms.Button), TextBox (System.WinForms.TextBox), and Label (System.WinForms.Label) fields in the AddEditForm class to represent the form's controls. It also initializes these fields with references to instances of Button, TextBox, and Label.

It initializes the properties of each control and of the form itself.

It physically adds the controls to the form by calling Add on the form's Controls collection.

Most of this code is located in a method named InitializeComponent, which is called from the class constructor. The "Component" in InitializeComponent refers to the form itself. Most of the code that you see in the InitializeComponent method was generated at the time you added the controls to the form and changed the controls' properties. The statements you entered add Title, Artist, and Comment properties to the form class. These properties permit callers to access the text in the form's edit controls.

Step 6: Add Event Handlers to MainForm

The next step is to add event handlers to the MainForm class—handlers that will be called when the Add, Edit, or Remove button is clicked, or when an item in the list view control is double-clicked.

In Windows Forms, controls fire events in response to user input. MainForm connects each pushbutton's Click event to an XxxButtonClicked method, and the ListView's DoubleClick event to OnItemDoubleClicked. The handlers for the Add and Edit buttons instantiate AddEditForm and display it on the screen by calling its ShowDialog method. ShowDialog's return value tells you how the dialog was dismissed: DialogResult.OK if the OK button was clicked, DialogResult.Cancel if Cancel was clicked instead. To get data in and out of the dialog, the handlers read and write the properties you added to AddEditForm in Step 5. Once again, the forms editor provides the code that defines the form's appearance, and you provide the code that implements its behavior.

Now it is a good time to build the project if you haven't already. You don't have to go out to the command line; instead, you can select the Build command from the Visual Studio Build menu. Then you can run TuneTown.exe by selecting one of the Debug menu's Start commands.

Step 7: Add Anchors

TuneTown is almost complete, but there's an important element that's still missing—an element that highlights one of the coolest features in Windows Forms. To see what I mean, run the application and resize its main window. The window resizes just fine (that's because the form's BorderStyle property is set to Sizable; to prevent resizing, you could change the BorderStyle to

FixedDialog), but the controls stay put. Wouldn't it be nice if the controls flowed with the form, automatically resizing and repositioning themselves to utilize all the real estate available to them? To do that in a traditional Windows-based application, you'd have to handle WM_SIZE messages and programmatically move and resize the controls. In Windows Forms, you can do it with about one tenth of the effort. In fact, you don't have to write a single line of code.

Every Windows Form control inherits a property named Anchor from System.WinForms.RichControl. The Anchor property describes which edges of its parent a control's own edges should be glued to. If, for example, you set a control's Anchor property to AnchorStyles.Right, then the distance between the control's right edge and the right edge of the form will remain constant, even if the form is resized. By setting the Anchor properties of the controls in MainForm, you can easily configure the push buttons to move with the form's right edge and the list view control to stretch both vertically and horizontally to fill the remaining space in the form. Here's how.

Open MainForm in the forms editor and select the ListView's Anchor property. Initially, Anchor is set to TopLeft; change it to All. Then, one at a time, set the pushbuttons' Anchor properties to TopRight. These actions result in the following statements being added to MainForm's InitializeComponent method:

```
AddButton.Anchor = System.WinForms.AnchorStyles.TopRight;
EditButton.Anchor = System.WinForms.AnchorStyles.TopRight;
RemoveButton.Anchor = System.WinForms.AnchorStyles.TopRight;
TuneView.Anchor = System.WinForms.AnchorStyles.All;
```

Now, rebuild the application and resize the form again. This time, the controls should flow with the form. I'm sure you've been frustrated by dialogs whose controls are too small to show everything that you want them to show. Frustrations such as these will be a thing of the past when developers take advantage of Windows Forms anchoring.

Step 8: Add Persistence

The final step is to make the data entered into TuneTown persistent by writing the contents of the list view control to a disk file when TuneTown closes, and reading those contents back the next time TuneTown starts up. I'll use the .NET Framework class library's StreamWriter and StreamReader classes (members of the System.IO namespace) to make short work of inputting and outputting text strings.

The necessary changes are highlighted in Figure 20. OnClosing is a virtual method inherited from System.WinForms.Form that's called just before a form closes. MainForm's OnClosing implementation creates a text file named TuneTownData.ttd in the local user's application data path (SystemInformation.GetFolderPath (SpecialFolder.LocalApplicationData)). Then it writes the text of each item and subitem in the list view control to the file. Next time TuneTown starts up, MainForm's InitializeListView method opens the file, reads the strings, and writes them back to the list view.

Notice the calls to the StreamWriter and StreamReader objects' Close methods. The reason we included these calls has to do with deterministic versus nondeterministic destruction. Languages such as C++ employ deterministic destruction, in which objects go out of scope (and are destroyed) at precise points in time. However, the .NET CLR, which uses a garbage collector to reclaim resources, uses nondeterministic destruction, meaning that there's no guarantee when (or if) the garbage collector will kick in. Normally you do not care when garbage collection is performed. But if an object encapsulates a non-memory resource such as a file handle, you very much want destruction to be deterministic because you want that file handle closed as soon as it is no longer needed. For reasons that we will not go into here, the problem is exacerbated when StreamWriter objects are involved because a StreamWriter's Finalize method, which is called when a StreamWriter object is destroyed, does not bother to close any file handles or flush buffered data to disk. If you don't call Close, data could be lost. We took the extra step of using finally blocks to enclose TuneTown's calls to Close to be sure that Close is called, even after inopportune exceptions.

Review Questions

1. Explain VS.Net IDE?
2. Write the steps involved in creating windows forms using VS.NET IDE?

Chapter 14 Data access with ADO.NET

14.1 Files and Streams

The System.IO namespace contains types that allow synchronous and asynchronous reading and writing on data streams and files.

The following distinctions help clarify the differences between a file and a stream. A file is an ordered and named collection of a particular sequence of bytes having persistent storage. Therefore, with files, one thinks in terms of directory paths, disk storage, and file and directory names. In contrast, streams provide a way to write and read bytes to and from a backing store that can be one of several storage mediums. Just as there are several backing stores other than disks, there are several kinds of streams other than file streams. For example, there are network, memory, and tape streams.

14.1.1 Basic File I/O

The abstract base class Stream supports reading and writing bytes. **Stream** integrates asynchronous support. Its default implementations define synchronous reads and writes in terms of their corresponding asynchronous methods, and vice versa.

All classes that represent streams inherit from the **Stream** class. The **Stream** class and its derived classes provide a generic view of data sources and repositories, isolating the programmer from the specific details of the operating system and underlying devices.

Streams involve these fundamental operations:

Streams can be read from. Reading is the transfer of data from a stream into a data structure, such as an array of bytes.

Streams can be written to. Writing is the transfer of data from a data structure into a stream.

Streams can support seeking. Seeking is the querying and modifying of the current position within a stream.

Depending on the underlying data source or repository, streams might support only some of these capabilities. For example, NetworkStreams do not support seeking. The CanRead, CanWrite, and CanSeek properties of **Stream** and its derived classes determine the operations that various streams support.

I/O Classes Derived from System.Object

BinaryReader and BinaryWriter read and write encoded strings and primitive data types from and to **Streams**.

File provides static methods for the creation, copying, deletion, moving, and opening of files, and aids in the creation of FileStream objects. The **FileInfo** class provides instance methods.

Directory provides static methods for creating, moving, and enumerating through directories and subdirectories. The **DirectoryInfo** class provides instance methods.

Path provides methods and properties for processing directory strings in a cross-platform manner.

File, **Path**, and **Directory** are sealed (in Microsoft Visual Basic, **NotInheritable**) classes. You can create new instances of these classes, but they can have no derived classes.

System.IO.FileSystemInfo and Its Derived Classes

FileSystemInfo is the abstract base class for **FileInfo** and **DirectoryInfo** objects.

FileInfo provides instance methods for the creation, copying, deletion, moving, and opening of files, and aids in the creation of FileStream objects. The **File** class provides static methods.

DirectoryInfo provides instance methods for creating, moving, and enumerating through directories and subdirectories. The **Directory** class provides static methods.

FileInfo and **DirectoryInfo** are sealed (in Microsoft Visual Basic, **NotInheritable**) classes. You can create new instances of these classes, but they can have no derived classes.

Classes Derived from System.IO.Stream

FileStream supports random access to files through its Seek method. **FileStream** opens files synchronously by default, but supports asynchronous operation as well. **File** contains static methods, and **FileInfo** contains instance methods.

A MemoryStream is a nonbuffered stream whose encapsulated data is directly accessible in memory. This stream has no backing store and might be useful as a temporary buffer.

A NetworkStream represents a **Stream** over a network connection. Although **NetworkStream** derives from **Stream**, it is not part of the **System.IO** namespace, but is in the System.NET.Sockets namespace.

A CryptoStream links data streams to cryptographic transformations. Although **CryptoStream** derives from **Stream**, it is not part of the **System.IO** namespace, but is in the System.Security.Cryptography namespace.

A BufferedStream is a **Stream** that adds buffering to another **Stream** such as a **NetworkStream**. (**FileStream** already has buffering internally, and a **MemoryStream** does not need buffering). A **BufferedStream** object can be composed around some types of streams in order to improve read and write performance. A buffer is a block of bytes in memory used to cache data, thereby reducing the number of calls to the operating system.

System.IO.TextReader and Its Derived Classes

TextReader is the abstract base class for **StreamReader** and **StringReader** objects. While the implementations of the abstract **Stream** class are designed for byte input and output, the implementations of **TextReader** are designed for Unicode character output.

StreamReader reads characters from **Streams**, using Encoding to convert characters to and from bytes. **StreamReader** has a constructor that attempts to ascertain what the correct **Encoding** for a given **Stream** is, based on the presence of an **Encoding**-specific preamble, such as a byte order mark.

StringReader reads characters from **Strings**. **StringReader** allows you to treat **Strings** with the same API, so your output can be either a **Stream** in any encoding or a **String**.

System.IO.TextWriter and Its Derived Classes

TextWriter is the abstract base class for **StreamWriter** and **StringWriter** objects. While the implementations of the abstract **Stream** class are designed for byte input and output, the implementations of **TextWriter** are designed for Unicode character input.

StreamWriter writes characters to **Streams**, using Encoding to convert characters to bytes.

StringWriter writes characters to **Strings**. **StringWriter** allows you to treat **Strings** with the same API, so your output can be either a **Stream** in any encoding or a **String**.

14.2 Enumerators

The FileAccess, FileMode, and FileShare enumerations define constants used by some of the **FileStream** and IsolatedStorageFileStream constructors and some of the **File.Open** overloaded methods. These constants affect the way in which the underlying file is created, opened, and shared.

The SeekOrigin enumerator defines constants that specify the point of entry for random access to files. These constants are used with byte offsets.

14.2.1 I/O and Security

When using the classes in the **System.IO** namespace, operating system security requirements such as access control lists (ACLs) must be satisfied for access to be allowed. This requirement is in addition to any FileIOPermission requirements.

A backing store is a storage medium, such as a disk or memory. Each different backing store implements its own stream as an implementation of the Stream class. Each stream type reads and writes bytes from and to its given backing store. Streams that connect to backing stores are called base streams. Base streams have constructors that have the parameters necessary to connect the stream to the backing store. For example, FileStream has constructors that specify a path parameter, a parameter that specifies how the file will be shared by processes, and so on.

The design of the System.IO classes provides simplified stream composing. Base streams can be attached to one or more pass-through streams that provide the desired functionality. A reader or writer can be attached to the end of the chain so that the desired types can be read or written easily.

The following code example creates a **FileStream** object around the existing `MyFile.txt` in order to buffer `MyFile.txt`. (Note that **FileStreams** are buffered by default.) Next, a StreamReader is created to read characters from the **FileStream**, which is passed to the **StreamReader** as its constructor argument. ReadLine reads until Peek finds no more characters.

```
Option Explicit
Option Strict
Imports System
Imports System.IO
Public Class CompBuf
  Private Const FILE_NAME As String = "MyFile.txt"
  Public Shared Sub Main()
    If Not File.Exists(FILE_NAME) Then
      Console.WriteLine("{0} does not exist!", FILE_NAME)
      Return
    End If
    Dim fsIn As New FileStream(FILE_NAME, FileMode.Open, FileAccess.Read, FileShare.Read)
    ' Create a reader that can read characters from the FileStream.
    Dim sr As New StreamReader(fsIn)
    ' While not at the end of the file, read lines from the file.
    While sr.Peek() > -1
      Dim input As String = sr.ReadLine()
```

```
        Console.WriteLine(input)
      End While
      sr.Close()
    End Sub
End Class
using System;
using System.IO;
public class CompBuf {
    private const string FILE_NAME = "MyFile.txt";
    public static void Main(String[] args) {
      if (!File.Exists(FILE_NAME)) {
        Console.WriteLine("{0} does not exist!", FILE_NAME);
        return;
      }
      FileStream fsIn = new FileStream(FILE_NAME, FileMode.Open,
        FileAccess.Read, FileShare.Read);
      // Create a reader that can read characters from the FileStream.
      StreamReader sr = new StreamReader(fsIn);
      // While not at the end of the file, read lines from the file.
      while (sr.Peek()>-1) {
        String input = sr.ReadLine();
        Console.WriteLine (input);
      }
      sr.Close();
    }
}
```

The following code example creates a **FileStream** object around the existing `MyFile.txt` in order to buffer `MyFile.txt`. (Note that **FileStreams** are buffered by default.) Next, a **BinaryReader** is created to read bytes from the **FileStream**, which is passed to the **BinaryReader** as its constructor argument. ReadByte reads until PeekChar finds no more bytes.

```
Option Explicit
Option Strict
Imports System
Imports System.IO
Public Class ReadBuf
    Private Const FILE_NAME As String = "MyFile.txt"
    Public Shared Sub Main()
      If Not File.Exists(FILE_NAME) Then
        Console.WriteLine("{0} does not exist!", FILE_NAME)
        Return
      End If
      Dim f As New FileStream(FILE_NAME, FileMode.Open, FileAccess.Read, FileShare.Read)
      ' Create a reader that can read bytes from the FileStream.
      Dim sr As New BinaryReader(f)
      ' While not at the end of the file, read lines from the file.
      While sr.PeekChar() > - 1
        Dim input As Byte = sr.ReadByte()
        Console.WriteLine(input)
      End While
      sr.Close()
    End Sub
End Class
using System;
using System.IO;
public class ReadBuf {
    private const string FILE_NAME = "MyFile.txt";
    public static void Main(String[] args) {
      if (!File.Exists(FILE_NAME)) {
        Console.WriteLine("{0} does not exist!", FILE_NAME);
        return;
      }
      FileStream f = new FileStream(FILE_NAME, FileMode.Open,
        FileAccess.Read, FileShare.Read);
      // Create a reader that can read bytes from the FileStream.
      BinaryReader sr = new BinaryReader(f);
      // While not at the end of the file, read lines from the file.
      while (sr.PeekChar()>-1) {
        byte input = sr.ReadByte();
        Console.WriteLine (input);
      }
      sr.Close();
    }
}
```

14.3 Accessing Data with ADO.NET

As you develop applications using ADO.NET, you will have different requirements for working with data. In some cases, you might simply want to display data on a form. In other cases, you might need to devise a way to share information with another company.

No matter what you do with data, there are certain fundamental concepts that you should understand about the data approach in ADO.NET. You might never need to know some of the details of data handling — for example, you might never need to directly edit an XML file containing data — but it is very useful to understand the data architecture in ADO.NET, what the major data components are, and how the pieces fit together.

This introduction presents a high-level overview of these most important concepts. The topic deliberately skips over many details — for example, there is much more to datasets than what is mentioned here — in favor of simply introducing you to ideas behind data integration in ADO.NET.

ADO.NET Does Not Depend On Continuously Live Connections

In traditional client/server applications, components establish a connection to a database and keep it open while the application is running. For a variety of reasons, this approach is impractical in many applications:

Open database connections take up valuable system resources. In most cases, databases can maintain only a small number of concurrent connections. The overhead of maintaining these connections detracts from overall application performance.

Similarly, applications that require an open database connection are extremely difficult to scale up. An application that does not scale up well might perform acceptably with four users but will likely not do so with hundreds. ASP.NET Web applications in particular need to be easily scalable, because traffic to a Web site can go up by orders of magnitude in a very short period.

In ASP.NET Web applications, the components are inherently disconnected from each other. The browser requests a page from the server; when the server has finished processing and sending the page, it has no further connection with the browser until the next request. Under these circumstances, maintaining open connections to a database is not viable, because there is no way to know whether the data consumer (the client) requires further data access.

A model based on always-connected data can make it difficult and impractical to exchange data across application and organizational boundaries using a connected architecture. If two components need to share the same data, both have to be connected, or a way must be devised for the components to pass data back and forth.

For all these reasons, data access with ADO.NET is designed around an architecture that uses connections sparingly. Applications are connected to the database only long enough to fetch or update the data. Because the database is not holding on to connections that are largely idle, it can service many more users.

14.4 Database Interactions - Using Data Commands

To perform operations in a database, you execute SQL statements or stored procedures (which include SQL statements). You use SQL statements or stored procedures to read and write rows and perform aggregate functions, such as adding or averaging. You also use SQL statements or stored procedures to create or modify tables or columns, to perform transactions, and so on.

In ADO.NET you use data commands to package a SQL statement or stored procedure. For example, if you want to read a set of rows from the database, you create a data command and configure it with the text of a SQL Select statement or the name of a stored procedure that fetches records.

When you want to get the rows, you do the following:

Open a connection.

Call an execute method of the command, which in turn:

Executes the SQL statement or stored procedure referenced by the command.

Then closes the connection.

The connection stays open only long enough to execute the statement or stored procedure.

When you call a command's execute method, it returns a value. Commands that update the database return the number of rows affected; other types of commands return an error code. If the command queries the database with a SELECT statement, the command can return a set of rows. You can fetch these rows using a data reader, which acts as a very fast read-only, forward-only cursor. For more information, see Retrieving Data Using the DataReader.

If you need to perform more than one operation — for example, read some rows and then update them — you use multiple data commands, one for each operation. Each operation is performed separately. For example, to read the rows, you open the connection, read the rows, and then close the connection. When you want to update data, you again open the connection, perform the update, and then close the connection again.

Data commands can include parameters (specifically, a collection of parameter objects) that allow you to create parameterized queries such as the following:

Select * From customers Where (customer_id = @customerid)

You can then set the parameters at run time and execute the command to return or update the data you want.

14.5 Caching data in Datasets

The most common data task is to retrieve data from the database and do something with it: display it, process it, or send it to another component. Very frequently, the application needs to process not just one record, but a set of them: a list of customers or today's orders, for example. Often the set of records that the application requires comes from more than one table: my customers and all their orders; all authors named "Smith" and the books they have written; and other, similar, sets of related records.

Once these records are fetched, the application typically works with them as a group. For example, the application might allow the user to browse through all the authors named "Smith" and examine the books for one Smith, then move to the next Smith, and so on.

In many cases, it is impractical to go back to the database each time the application needs to process the next record. (Doing so can undo much of the advantage of minimizing the need for open connections.) A solution, therefore, is to temporarily store the records retrieved from the database and work with this temporary set.

This is what a dataset is. A dataset is a cache of records retrieved from a data source. It works like a virtual data store: A dataset includes one or more tables based on the tables in the actual database, and it can include information about the relationships between those tables and constraints on what data the tables can contain.

The data in the dataset is usually a much-reduced version of what is in the database. However, you can work with it in much the same way you do the real data. While you are doing so, you remain disconnected from the database, which frees it to perform other tasks.

Of course, you often need to update data in the database (although not nearly as often as you retrieve data from it). You can perform update operations on the dataset, and these can be written through to the underlying database.

An important point is that the dataset is a passive container for the data. To actually fetch data from the database and (optionally) write it back, you use data adapters. A data adapter contains one or more data commands used to populate a single table in the dataset and update the corresponding table in the database. (A data adapter typically contains four commands, one each to select, insert, update, and delete rows in the database.) Therefore, a data adapter's **Fill** method might execute a SQL statement such as SELECT au_id, au_lname, au_fname FROM authors whenever the method is called.

Because a dataset is effectively a private copy of the database data, it does not necessarily reflect the current state of the database. If you want to see the latest changes made by other users, you can refresh the dataset by calling the appropriate **Fill** method.

One of the advantages of using datasets is that components can exchange them as required. For example, a business object in the middle tier might create and populate a dataset, then send it to another component elsewhere in the application for processing. This facility means that components do not have to individually query the database.

14.5.1 Datasets Are Independent of Data Sources

Although a dataset acts as a cache for data drawn from a database, the dataset has no actual relationship with the database. The dataset is a container; it is filled by SQL commands or stored procedures executed from a data adapter.

Because a dataset is not directly tied to a data source, it is a good integration point for data coming from multiple sources. For example, some of the data in a dataset might come from your database, whereas other parts of it might come from a different database or a non-database source such as a spreadsheet. Some of the data in a dataset might arrive in a stream sent by another component. Once the data is in a dataset, you can work with it using a consistent object model, regardless of its original source.

14.5.2 Data Is Persisted as XML

Data needs to be moved from the data store to the dataset, and from there to various components. In ADO.NET the format for transferring data is XML. Similarly, if data needs to be persisted (for example, into a file), it is stored as XML. If you have an XML file, you can use it like any data source and create a dataset out of it.

In fact, in ADO.NET, XML is a fundamental format for data. The ADO.NET data APIs automatically create XML files or streams out of information in the dataset and send them to another component. The second component can invoke similar APIs to read the XML back into a dataset. (The data is not stored in the dataset as XML — for example, you cannot parse data in a dataset using an XML parser — but instead in a more efficient format.)

Basing data protocols around XML offers a number of advantages:

XML is an industry-standard format. This means that your application's data components can exchange data with any other component in any other application, as long as that component understands XML. Many applications are being written to understand XML, which provides an unprecedented level of exchange between disparate applications.

XML is text-based. The XML representation of data uses no binary information, which allows it to be sent via any protocol, such as HTTP. Most firewalls block binary information, but by formatting information in XML, components can still easily exchange the information.

For most scenarios, you do not need to know XML in order to use data in ADO.NET. ADO.NET automatically converts data into and out of XML as needed; you interact with the data using ordinary programming methods.

14.5.3 Schemas Define Data Structures

Although you do not need to know anything about XML to read and write to the database and work with datasets, there are situations in which working with XML is precisely the goal you are after. These are situations in which you are not accessing data, but instead, working with the design of data. To put it another way, in ADO.NET you use XML directly when you are working with metadata.

Datasets are represented as XML. The structure of the dataset — the definition of what tables, columns, data types, constraints, and so on are in the dataset — is defined using an XML Schema based on the XML Schema definition language (XSD). Just as data contained by a dataset can loaded from and serialized as XML, the structure of the dataset can be loaded from and serialized as XML Schema.

For most of the work you do with data in ADO.NET, you do not have to delve deeply into schemas. Typically, the Visual Studio .NET tools will generate and update schemas as needed, based on what you do in visual designers. For example, when you use the tools to create a dataset representing tables in your database, Visual Studio .NET generates an XML Schema describing the structure of the dataset. The XML Schema is then used to generate a typed dataset, in which data elements (tables, columns, and so on) are available as first-class members.

However, there are times when you want to create or edit schemas yourself. A typical example is developing a schema in conjunction with a partner or client. In that case, the schema serves as a contract between you and the partner regarding the shape of the XML-based data you will exchange. In that situation, you will often have to map the schema elements to the structure of your own database.

14.6 Components of ADO.NET

The following illustration shows the major components of an ADO.NET application.

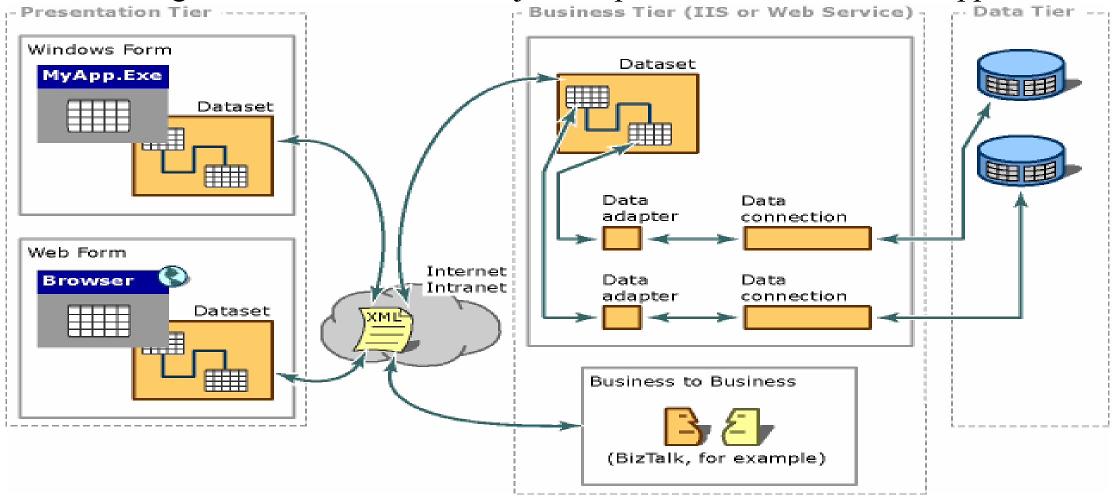


Fig 6

14.6.1 Using .NET Data Providers to Access Data

A data provider in the .NET Framework serves as a bridge between an application and a data source. A data provider is used to retrieve data from a data source and to reconcile changes to that data back to the data source.

The following table lists the .NET data providers that are included in the .NET Framework.

.NET data provider	Description
SQL Server .NET Data Provider	For Microsoft® SQL Server™ version 7.0 or later.
OLE DB .NET Data Provider	For data sources exposed using OLE DB.

An Open Database Connectivity (ODBC) .NET Data Provider is available as a separate download at http://msdn.microsoft.com/downloads. The download includes documentation on the classes that make up the ODBC .NET Data Provider. However, the implementation has the same architecture as both the SQL Server .NET Data Provider and the OLE DB .NET Data Provider. As a result, you can apply the information found in this section to the ODBC .NET Data Provider as well.

The **Connection**, **Command**, **DataReader**, and **DataAdapter** objects represent the core elements of the .NET data provider model. The following table describes these objects.

Object	Description
Connection	Establishes a connection to a specific data source.
Command	Executes a command against a data source.
DataReader	Reads a forward-only, read-only stream of data from a data source.
DataAdapter	Populates a DataSet and resolves updates with the data source.

Along with the core classes listed in the preceding table, a .NET data provider also contains the classes listed in the following table.

Object	Description
Transaction	Enables you to enlist commands in transactions at the data source.
CommandBuilder	A helper object that will automatically generate command properties of a DataAdapter or will derive parameter information from a stored procedure and populate the Parameters collection of a Command object.
Parameter	Defines input, output, and return value parameters for commands and stored procedures.
Exception	Returned when an error is encountered at the data source. For an error encountered at the client, .NET data providers throw a .NET Framework exception.
Error	Exposes the information from a warning or error returned by a data source.
ClientPermission	Provided for .NET data provider code access security attributes.

Some of the packaging and deployment story for the .NET Framework is described in other sections of the .NET Framework SDK documentation. These sections provide information about the self-describing units called assemblies, which require no registry entries, strong-named assemblies, which ensure name uniqueness and prevent name spoofing, and assembly versioning, which addresses many of the problems associated with DLL conflicts. This section provides information about packaging and distributing .NET Framework applications.

14.6.2 Packaging and Deployment

The .NET Framework provides a number of basic features that make it easier to deploy a variety of applications. These features include:

No-impact applications : This feature provides application isolation and eliminates DLL conflicts. By default, components do not affect other applications.

Private components by default: By default, components are deployed to the application directory and are visible only to the containing application.

Controlled code sharing: Code sharing requires you to explicitly make code available for sharing rather than being the default behavior.

Side-by-side versioning: Multiple versions of a component or application can coexist, you can choose which versions to use, and the common language runtime enforces versioning policy.

XCOPY deployment and replication: Self-described and self-contained components and applications can be deployed without registry entries or dependencies.

On-the-fly updates: Administrators can use hosts, such as ASP.NET, to update program DLLs, even on remote computers.

Integration with the Microsoft Windows Installer: Advertisement, publishing, repair, and install-on-demand are all available when deploying your application.

Enterprise deployment: This feature provides easy software distribution, including using Active Directory.

Downloading and caching: Incremental downloads keep downloads smaller, and components can be isolated for use only by the application for zero-impact deployment.

Partially trusted code: Identity is based on the code rather than the user, policy is set by the administrator, and no certificate dialog boxes appear.

14.6.3 Packaging

The .NET Framework provides the following options for packaging applications:

As a single assembly or as a collection of assemblies : With this option, you simply use the .dll or .exe files as they were built.

As cabinet (CAB) files: With this option, you compress files into .cab files to make distribution or download less time consuming.

As a Microsoft Windows Installer 2.0 package or in other installer formats: With this option, you create .msi files for use with the Windows Installer or you package your application for use with some other installer.

14.6.4 Distribution

The .NET Framework provides the following options for distributing applications:

Use XCOPY or FTP: Because common language runtime applications are self-describing and require no registry entries, you can use XCOPY or FTP to simply copy the application to an appropriate directory. The application can then be run from that directory.

Use code download: If you are distributing your application over the Internet or through a corporate intranet, you can simply download the code to a computer and run the application there.

Use an installer program such as Windows Installer 2.0: Windows Installer 2.0 can install, repair, or remove Microsoft .NET Framework assemblies in the global assembly cache and in private directories.

This section describes several possible deployment scenarios that can be used for common language runtime applications. You package and deploy your application differently depending on your deployment requirements. Note that these are only suggested scenarios for particular types of applications. Your deployment needs might dictate using another method. The following are typical deployment scenarios:

14.7 Deploying an ASP.NET application.

Packaging: Application and DLLs

Distribution: XCOPY or FTP distribution

You can use XCOPY or FTP to deploy an ASP.NET application onto a server. You can then run one version of the application side-by-side with another version, and you can also update the application without closing it. The common language runtime makes it easy for you to run the application simultaneously with other applications, without DLL conflicts.

Deploying a Windows Forms application.

Packaging: Microsoft Windows Installer package (.msi)

Distribution: Windows Installer

Distributing a Windows Forms application using the Windows Installer allows you to leverage both the Installer and Windows 2000 Application Management. You can also advertise the application's availability, publish the application, use the **Add/Remove Programs** option in Control Panel to install or remove the application, and easily repair the application, if necessary.

Deploying a Windows Forms Control or other code by downloading.

Packaging: Compressed CAB files (.cab) or compiled libraries (.dll)

Distribution: Code download

Distributing a Windows Forms control can be as simple as making the application available for download on a Web host. You can compress the files that make up the application for quicker download.

<u>Review Questions</u>

1. What are the fundamental operationd of the streams?
2. What are the classes derived from System.IO sream?
3. Differentiate the legacy connection and the data access with ADO.Net?
4. What bis the format used to transfer data in ADO.Net. What are its advantages?
5. How will you cache data in a data set?
6. Explain the components od ADO.Net. Illustrate with help of a block diagram?
7. What are the options provided by .Net frame work in
   Packaging Application.
   Distribution application.

Chapter 15 - Graphics

When you are working with an operating system, such as Windows that is based on a graphical user interface (GUI) you are naturally confronted with the problem of completing graphical tasks. Working with graphics in the windows world involves such tasks as drawing lines, arcs, and figures, as you would expect. However, it also involves less obvious tasks, like putting images on the screen and displaying text in fancy fonts. Although all these tasks come under the heading of "working with graphics," they are rarely considered that way by the average programmer. The C# system in .NET supports a wide variety of graphics functionality, including putting images into control easily. Fonts, colors, lines, and figures are easily displayed on the screen and manipulated just as they were in the "good old days" of windows programming, when the only available options were the windows API functions.

You will find differences between the graphics system of the older versions of windows and that .NET. For one thing, the graphical interface in C# is much easier to user and is much more consistent than the old-style device context functions of windows. Rather than have to worry about getting and releasing device contexts, you now work with a graphics object that contains all the functionality as a series of methods that can be used to change the display. Colors and fonts are now properties of the controls they affect rather than strange and arbitrary windows messages that are sent to handlers for the controls although these changes are all for the best they can cause difficulty if you are accustomed to working with the old-style programming interface. For this reason, I walk you through the graphics system in C# step-by-step, looking at how it is all put together and how you can use it to your best advantage.

The first step on your tour of the graphics system in C# is with the GDI+ library. If you are an old-time windows programmer, you are probably accustomed to working with the **graphics device interface (GDI)**. This system was created to abstract (get away from) the work of displaying graphics on various graphics devices, such as monitors and printers. In "classic" windows programming, the GDI was reached by retrieving a handle to a device context (known as HDC) and passing that handle to various functions to perform such actions as drawing lines or displaying text on the screen. The problem was always that the number of device context handles was limited on the system. Forgetting to release a handle was a common problem in windows programming and often led to system slowdowns at best – and system crashes at worst.

15.1 The GDI+ Interface

The GDI+ interface is a change from the old GDI functionality of windows and from the aaa GDI functionality of earliuer versions of C# and the CLR. Rather than assign properties to each and every element uyou want to change in the graphics system, you instead utilize those properties to a Graphics object method. For example, if you are accustomed to working with the old-style GDI, you might write:

```
Graphics g = new Graphics();
g.Font = new Font("Times New Roman", 22);
g.Forecolor = new Color(Color.Blue);
```

```
g.Backcolor = new Color(Color.White);
g.DrawString("Hello World", 0,0);
```
In the old-style windows programming, this block of code would look something like this:
```
HDC hDC = GetDC();
SelectObject(hDC,&font);
TextOut (hdc, 0, 0, "Hello World", strlen("Hello World"));
```

In the new GDI, however, the code looks like this:
```
Graphics g = new Graphics();
g.DrawString("Hello World", new Font("Times New Roman", 22),
new SolidBrush( new Color(Color.Blue)), 0, 0);
```

The most important thing to notice about this particular block is that you don't pay much attention to the objects you pass in after you are done with them. You don't need to free up a device context handle or delte the font object or worry abouyt the bvrsuh object you are passing into the DrawString method. All these tasks are done automatically by the garbage-collection system, at an appropriate time in the application. This geqture is by far one of the most useful about working with C#. you lose the danger of leaking resources and causing dangerous system memory leaks.
The GDI+ classes "live" in the **System.Drawing**,**System.Drawing.2D**, **System.Drawing.Imaging**, and **System.Drawing.Text** namespaces and are contained in the System.Drawiung.DLL reference library. By including this reference in your project in Visual Studio.NET, you have access to all the drawing functionality that is inherent in the C# development system.

15.2 The Graphics Object

The core of the graphics functionality in C# is provided by the Graphics class. It contains all the methods you need in order to render images; draw lines, arcs, or figures; or display text on the screen. As you see a bit later in this chapter, this class also contains all the functionality needed to do grapohics transformation, which is necessary for such tasks as rotating text displays or rotating 3D objects on the screen.

You create a graphics object by instantiating an object of the **Graphics** class. Use the following single line of code:

```
Graphics g = new Graphics();
```

A *graphics object* is simply a "surface" on which you can render text, graphics, colors, and fonts. Rather than use the old style of selecting the objects you want to use in the method calls you make to the object. For example, you could create a line on a graohics display by using this code:

```
Graphics g = new Graphics();
Pen aPen = new Pen(Color.Blue);
Point startPoint = new Point(0,0);
Point endpoint = new Point(100,100);
```

```
g.DrawLine( aPen, startPoint, endpoint);
```

This snippet of code creates a new graphics object, creates a new blue pen, and draws a line from 0,0 to 100,100 in the current graphics coordinate system. When the function that calls this snippet ends, the Pen, startPoint, and endpoint objects (in addition to the Graphics object) all simply go away and release the memory associated with them. Unlike with previous Windows programming issues, the window associated with this graphics object is not affected by the objects disappearance.

You can normally create graphics objects in one of three ways:

You may be given a graphics object as part of a Paint event.

You may create an "off-screen" graphics object so that you can create a bitmap for immediate display on the screen with no flicker or delay.

You may get a graphics object to user for printing.

You do not really see a difference between drawing on a screen graphics object and a printer graphics object, so I don't talk about the differences between these two approaches. However, some minor changes have taken place for working with off-screen bitmaps. Next, take a look at a simple example of creating such an off-screen bitmap, often called double buffering in the graphics world.

This piece of code creates a bitmap and fills it with color. it then draws a text string in the middle of the bitmap:

```
Bitmap the Bitmap = new Bitmap (nWidth, nHeight, PixelFormat, Format32bppArgb);
Graphics grfx = Graphics.FormImage(theBitmap);
Font theFont = new Font ("MS Sans Serif", 18);
Grfx.FillRectangle (new SolidBrush(Color.Blue),
New Rectangle(0,0,nWidth,nHeight));
Grfx.DrawString ("I'm an image", the Font,
New SolidBrush(Color.White),0 , 0);
```

After you have a bitmap, you can do whatever you want with it. You can display a image, such as a bitmap, in a piece of control. Alternatively, as you see in "Creating a PNG File on the Fly" in the "Immediate Solutions" section later in this chapter, you can save this bitmap to a file in a different format that can be loaded into the C# editor or another application.

A graphics object supports all the functionality you have come to expect in working with GUI-based applications. However, a graphics object by itself is limited in that it does not support colors, brushes, or pens. These items make the display exciting for users and attract the human eye (and good reviews) to an application.

15.2.1 Brushes

The brush is the instrument of choice for anyone wanting to paint the backgrounds of windows or controls. Brushes can also be used to create interesting and beautiful textures for filling in figures displayed on a form. Gradients can be quickly and easily displayed using a brush, making them indispensable for working with form backgrounds and title bars.

C# has two distinct types of brushes, implemented in two separate classes. The first one is the **SolidBrush** class, which implements a brush that is a single, uniform color across its display. For example, to create a brush that is a uniform shade of blue and uses it to fill in a rectangle on the drawing surface, you write this code:

```
Graphics g;  //Get a Graphics object
SolidBrush sb = new SolidBrush ( Color.Blue);
Rectangle r = new Rectangle (0,0, Width, Height);
g.FillRectangle (sb,r);
```

This code fills a rectangle (of size Width, Height) with a blue color. Rather than have to create a new brush from scratch each time you want one, you could use the **System** brushes. This collection contains all the standard precreated brushes, requiring no overhead of creation and destruction. You could rewrite the preceding code, therefore, as:

```
Graphics g;
Rectangle r = new Rectangle(0,0, Width, Height);
g.FillRectangle (Brushes.Blue, r);
```

These two pieces of code have exactly the same effect on the screen. The second one is slightly faster because of the lack of overhead of the creation and destruction of the graphics onjects. Unless you are creating a large number of brushes and using them in your code, however, you are unlikely to notice any real speed difference. The C# garbage-collection facilities are quite good and hide the over-head time.

In terms of methods, the **Brush** class itself has none to speak of. It has a single public property, **Color**, that represents the color of the solid brush. For methods, the **Brush** class implements all the standard methods of the **Objects** class (**Clone** and **Equals**, for example), but none of its own. A brush is a wrapper around a simple windows brush handle (HBRUSH).

The second kind of brush is the textured brush type. These objects – represented by the classes **TextureBrush**, **LinearGradient**, and **PathGradient** and **PathGradientBrush** classes, on the other hand, allow you to draw a gradient on the surface of a figure or form. A gradient is a gradual color change from one color to another, which looks like a three-dimensional surface because of its shadow-like color changes.

To create a **TextureBrush** object and use it to fill an area, you need to leave an image to work with. For example, to "paint" a bitmap across the background of a form, you could see the following code:

```
Graphics g;
Image I = new Bitmap("myImage.gif");
TextureBrush tb = new TextureBrush(i);
g.FillRectangle (tb,ClientRectangle);
```

This code results in the image contained in Myimage.gif being displayed across the background of the form window. You could then mute (soften) the colors of the image by "whitewashing" the image using the **SolidBrush** class;

```
g.FillRectangle( new SolidBrush(Color.FromArgb(180, Color.White)),
        ClientRectangle);
```

The **LinerGradient** brush (which is of type **TexturedBrush**), on the other hand, allows you to define you to define a straight gradient from one side or point in a rectangle to another on the form. Suppose that you want to paint a gradient across the background of a form. You could write this code in the Paint handler for the form:

```
LinearGradientBrush lgb = new LinearGradientBrush(
        ClientRectangle, Color.Blue,
        Color.Green, 45, true);
e.Graphics.FillRectangle( lgb, ClientRectangle );
```

The 45 in the constructor call is the angle through which the gradient is drawn. The angle at which the lines pass for the gradient is 45 degrees from the top. The true argument simply indicated that the angle is scalable. Unless you are working with transformations in this graphics display, which I talk about later in this chapter, the scalability means nothing when you are drawing the gradient.

Another kind of textured brush is **PathGradientBrush**. This brush is similar to the **LinearGradientBrush** class, but allows you to do much more complex tasks with the display:

```
Using System;
Using System.Drawing;
Using System.Collections;
Using System.ComponentModel;
Using System.Windows.Forms;
Using System.Data;
Using System.Drawing.Imaging;
Using System.Drawing.Drawing2D;

Namespace CH!!_Gradient
{
        public class Form1 : System.Windows.Forms.Form
{
                private System.ComponentModel.Container components = null;

                public Form()
                {
                        InitializeComponent();
                }
                protected override void Dispose( bool disposing )
                {
                        if( disposing )
                        {
                                if (components != null)
                                {
                                        components.Dispose();
                                }
                        }
                        base.Dispose (disposing);
                }

                #region Windows Form Designer generated code
                private void InitializeComponent()
                {
                        this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
                        this.ClientSize = new System.Drawing.Size(768, 533);
```

```
			this.Name = “Form1”;
			this.Text = “Form1”;
			this.Paint += new
System.Windows.Forms.PaintEventHandler(this.OnPaint);
		}

		#endregion

		[STAThread]
		static void Main()
		{
			Application.Run(new Form1());
		}

		private void OnPaint (object sender,
System.Windows.Forms.PaintEventArgs e)
		{
			Rectangle rect = this.ClientRectangle;
			GraphicsPath path = new GrapohicsPath (new Point[] {
				new Point (40, 140), new Point(275, 200),
				new Point (105,225), new Point
					(190, ClientRectangle.Bottom),
				new Point(50, ClientRectangle.Bottom),
					new Point(20, 180), },
				new byte[] {
				(byte)PathPointType.Start,
(byte)PathPointType.Bezier,
(byte)PathPointType.Bezier,
(byte)PathPointType.Bezier,
(byte)PathPointType.Line,
(byte)PathPointType.Line,
			});

			PathGradientBrush pgb = new PathGradientBrush(path);
			Pgb.SurroundColors = new Color[] {
				Color.Green,Color.Yellow,Color.Red,
					Color.Blue,Color.Orange, Color.White, };
			e.Graphics.FillPath (pgb, path);

		}
	}
}
```

15.2.2 Pens

The pen is the line-drawing tool of the graphics toolbox. Whenever you draw lines, display text, or draw figures on the screen you are affected by the pen selected to do the job. Pens can be solid, dashed, dotted, or textured (using a **TexturedBrush** class type) to display the lines they are used to draw. Whether you're drawing a straight line or an arc, the pen faithfully renders each point along the path of the figure.

Pens have either a color or a texture and a width. You can create a default pen with a single pixel width by using the standard constructor for the pen:

Pen p = new Pen(Color.Black);

This code creates a pen that draws its lines in black with a width of one pixel across (the default) the drawing surface. On the other hand, you can create a pen three pixels wide by suing this line:

Pen pl = new Pen(Color.Black, 3);

This line permits you to do some rather fancy drawing on the screen. It also allows you to scale up objects by making them thicker, which makes them appear larger on the screen.

15.2.3 Images

Images are the core of what people think about when they talk about graphics. Unlike pens, brushes, and other graphics tools, images are the part of the systems that generally see the most. In C#, many different image can be quickly and easily loaded into an **Image** class component and then displayed on the screen in a variety of ways.

To load an image, you can use either the constructor for the image type with the file name, like this

Image bmp = new Bitmap("mybitmap.bmp");

or the **FromFile** method of the **Image** class to load the image directly from a file at runtime. Here's a simple example of using this method to load an image.

Image bmp = Image.FromFile ("mybitmap.bmp");

The **FromFile** method and the constructor are functionally equivalent.

Transformations

The next graphics subject to consider is transformations. To understand what transformations are, you must first understand the coordinate system used by the GDI+ drawing classes. The GDI+ coordinate system is made up of two views world view and user view. Normally, these two views are aligned so that the upper-left corner is always 0,0 in either coordinate system. However, before any graphics are drawn, any user-defined transformations are applied and the user coordinates system is shifted by that amount. When the results are applied , they are drawn in world coordinate view, moving the drawing around as the transformations are applied. This system permits you, as a developer, to shift, rotate, and skew the graphics you want displayed on the user's screen.

Three basic types of transformations are allowed in the GDI+ drawing system: **ScaleTransform, RotateTransform,** and **TranlateTransform.** Transformations can be applied singly or in multiples for any given display.

The **ScaleTransform** method scales the entire display by a given amount in either the x or y direction or in both directions. To use the **ScaleTransform** method of the graphics object, you simply pass in the x and y adjustments. For example, the following snippet of code scales the display to be twice as high as you originally drew it.

g.ScaleTransform(1, 2);

The **RotateTransform** method simply rotates the graphics display about the origin of the user coordinates system. To rotate the entire drawing 45 degrees about the upper-left corner, for example, add the following line:

```
g.RotateTransform(45);
```

15.2.4 Text and Drawing

Neither is a complicated affair, but you should understand all the functionality you have at your fingertips when using the GDI+ system. The text-display functions consists of the **Font** object and the text-drawing functions. The graphics primitive functions consists of the **Graphics** class methods to draw lines, arcs, polygons, and other figures.

The powerful **Font** class in the GDI+ library lets you easily select a given font or create you variation of a font on the flyh. For example, to create a font that represents a 45-em (a printer's measure) Arial font, you write the code:

```
Font f = new Font(”Arial”, 45);
```

Alternatively, you can be fancy with your fonts and specify virtually every aspect of their display and behaviour: Select the **FontStyle** attribute version of the constructor for the **Font** object. For example, this font is in 50-ems Arial style and is also in bold italic style:

```
Font f = new Font(“Arial”, 45, FontStyle.Bold | FontStyle.Italic );
```

The possible enumerations of the **FontStyle** type are shown below. Most of the values can be OR'd together to form a combination, as shown in this example.

To display text, you see the **DrawString** method of the **Graphics** class. This method permits you to display text, position it on the screen, and select its font and color. The general form of the **DrawString** method is

```
g.DrawString ( string, font, brush, bounding_rectangle );
```

Style	Meaning
Bold	A bold (wide) format for the font
Italic	A slanted format for the font
Regular	Normal text for the font
Striekout	Text with a line through the center of it
Underline	Text that is underlined

where these conditions are true:

string is the string you want to display on the screen.

font is the font object you want to use to display the text string.

Brush is the brush object (any of the brush classes I have discussed) used to display the text string.

bounding_rectangle is the area in which you want the text to appear in user coordinates of the form window.

Sometimes, of course, you don't know the bounding rectangle for the string. All you are likely to know is where you want the text to begin on the screen. In these cases, the **Graphics** class provides the **MeasureString** method, which allows you to find the size of a given string using the current settings of the graphics object and the font in which you want the string displayed. The general form of the **MeasuringString** method is:

```
SizeF size = g.MeasureString ( string, fontToUse);
```

The returned **size** object specifies the width and length of the string for this particular font given the transformations applied to the graphics object at the time the method is called.

Graphics Primitives

The final topic in this section is the list of possible graphic primitives available to you in the **Graphics** class. These primitives form the backbone of the GDI+ methods for rendering graphics for users to view. The primitives are relatively self-explanatory. The complete list of available primitives is shown in the Table below.

In addition to simple primitives, a version of each solid-figure-drawing primitive also exists. These methods permits you to draw a rectangle, for example, and fill it with a given color (**FillRectanlge**, for example).

Primitive	Purpose
DrawArc	Draws a segment of a circle
DrawBezier	Draws a curve based on a set of points
DrawBeziers	Draws a series of curves
DrawClosedCurve	Draws a closed curve (the starting point and end point meet) based on an arbitrary set of points
DrawCurve	Draws a simple curve based on a set of points
DrawEllipse	Draws an ellipse (and alternatively a circle)
DrawIcon	Renders an icon on the screen
DrawIconUnstretched	Oddly, stretches an icon to fit user coordinates (and is named Unstretched because the transformation is done based on user requirements rather than on the system)
DrawImage	Draws an image (any Image-based class) on the screen
DrawLine	Draws a line between two points
DrawLines	Draws a series of lines, each defined by two points on the screen
DrawPath	Draws a series of lines connected by end points
DrawPie	Draws a pie chart section
DrawPolygon	Draws a closed polygon from a set of points
DrawRectangle	Draws a single rectangle from a set of points
DrawRectangles	Draws a series of rectangles that are all unconnected

Review Questions

1. Explain briefly the GDI + Interface in C# ?
2. What is a Graphic class. Illustrate with an example?
3. Write short notes on
   Brushes
   Pens
   Image class
4. Explain Transformations?
5. Explain Graphic Primitives of Graphic class?

UNIT IV

Chapter 16. Advance C#

16.1 Typed Dataset

Datasets store data in a disconnected cache. The structure of a dataset is similar to that of a relational database; it exposes a hierarchical object model of tables, rows, and columns. In addition, it contains constraints and relationships defined for the dataset.

Note You use datasets if you want to work with a set of tables and rows while disconnected from the data source. Using a dataset is not always an optimal solution for designing data access.

You can create and manipulate datasets using the following portions of the .NET Framework namespaces.

16.1.1 Dataset Namespace

The fundamental parts of a dataset are exposed to you through standard programming constructs such as properties and collections. For example:

The DataSet class includes the Tables collection of data tables and the Relations collection of DataRelation objects.

The DataTable class includes the Rows collection of table rows, the Columns collection of data columns, and the ChildRelations and ParentRelations collections of data relations.

The DataRow class includes the RowState property, whose values indicate whether and how the row has been changed since the data table was first loaded from the database. Possible values for the RowState property include Deleted, Modified, New, and Unchanged.

16.1.2 Datasets, Schemas, and XML

An ADO.NET dataset is one view — a relational view — of data that can be represented in XML. In Visual Studio and the .NET Framework, XML is the format for storing and transmitting data of all kinds. As such, datasets have a close affinity with XML. This relationship between datasets and XML enables you to take advantage of the following features of datasets:

The structure of a dataset — its tables, columns, relationships, and constraints — can be defined in an XML Schema. XML Schemas are a standards-based format of the W3C (World Wide Web Consortium) for defining the structure of XML data. Datasets can read and write schemas that store structured information using the ReadXmlSchema and WriteXmlSchema methods. If no schema is available, the dataset can infer one (via its InferXmlSchema method) from data in an XML document that is structured in a relational way. For more information about datasets and schemas, You can generate a dataset class that incorporates schema information to define its data structures (such as tables and columns) as class members. You can read an XML document or stream into a dataset using the dataset's ReadXML method and write a dataset out as XML using the dataset's WriteXML method. Because XML is a standard interchange format for data between different applications, this means that you can load a dataset with XML-formatted information sent by other applications. Similarly, a dataset can write out its data as an XML stream or document, to be shared with other applications or simply stored in a standard format.

You can create an XML view (an XMLDataDocument object) of the contents of a dataset, and then view and manipulate the data using either relational methods (via the dataset) or XML methods. The two views are automatically synchronized as they are changed.

16.1.3 Typed versus Untyped Datasets

Datasets can be typed or untyped. A typed dataset is a dataset that is first derived from the base **DataSet** class and then uses information in an XML Schema file (an .xsd file) to generate a new class. Information from the schema (tables, columns, and so on) is generated and compiled into this new dataset class as a set of first-class objects and properties.

Because a typed **DataSet** class inherits from the base **DataSet** class, the typed class assumes all of the functionality of the **DataSet** class and can be used with methods that take an instance of a **DataSet** class as a parameter

An untyped dataset, in contrast, has no corresponding built-in schema. As in a typed dataset, an untyped dataset contains tables, columns, and so on — but those are exposed only as collections. (However, after manually creating the tables and other data elements in an untyped dataset, you can export the dataset's structure as a schema using the dataset's WriteXmlSchema method.)

You can use either type of dataset in your applications. However, Visual Studio has more tool support for typed datasets, and they make programming with the dataset easier and less error-prone.

16.1.4 Contrasting Data Access in Typed and Untyped Datasets

The class for a typed dataset has an object model in which its tables and columns become first-class objects in the object model. For example, if you are working with a typed dataset, you can reference a column using code such as the following:

```
' Visual Basic
' This accesses the CustomerID column in the first row of
' the Customers table.
Dim s As String
s = dsCustomersOrders1.Customers(0).CustomerID

// C#
// This accesses the CustomerID column in the first row of
// the Customers table.
string s;
s = dsCustomersOrders1.Customers[0].CustomerID;
```

In contrast, if you are working with an untyped dataset, the equivalent code is:

```
' Visual Basic
Dim s As String
s = CType(dsCustomersOrders1.Tables("Customers").Rows(0).Item("CustomerID"), String)

// C#
        string s = (string) dsCustomersOrders1.Tables["Customers"].Rows[0]["CustomerID"];
```

Typed access is not only easier to read, but is fully supported by IntelliSense in the Visual Studio Code Editor. In addition to being easier to work with, the syntax for the typed dataset provides type checking at compile time, greatly reducing the possibility of errors in assigning values to dataset members. Access to tables and columns in a typed dataset is also slightly faster at run time because access is determined at compile time, not through collections at run time.

Even though typed datasets have many advantages, there are a variety of circumstances under which an untyped dataset is useful. The most obvious scenario is that no schema is available for the dataset. This might occur, for example, if your application is interacting with a component that returns a dataset, but you do not know in advance what its structure is. Similarly, there are times when you are working with data that does not have a static, predictable structure; in that case, it is impractical to use a typed dataset, because you would have to regenerate the typed dataset class with each change in the data structure.

More generally, there are many times when you might create a dataset dynamically without having a schema available. In that case, the dataset is simply a convenient structure in which you can keep information, as long as the data can be represented in a relational way. At the same time, you can take advantage of the dataset's capabilities, such as the ability to serialize the information to pass to another process, or to write out an XML file.

16.1.5 Dataset Case Sensitivity

Within a dataset, table and column names are by default case-insensitive — that is, a table in a dataset called "Customers" can also be referred to as "customers." This matches the naming conventions in many databases, including SQL Server, where the names of data elements cannot be distinguished only by case.

Note Unlike datasets, XML documents are case-sensitive, so the names of data elements defined in schemas are case-sensitive. For example, schema protocol allows the schema to contain define a table called "Customers" and a different table called "customers." This can result in name collisions when a schema is used to generate a dataset class.

However, case sensitivity can be a factor in how data is interpreted within the dataset. For example, if you filter data in a dataset table, the search criteria might return different results depending on whether the comparison is case-sensitive or not. You can control the case sensitivity of filtering, searching, and sorting by setting the dataset's CaseSensitive property. All the tables in the dataset inherit the value of this property by default. (You can override this property for each individual table.)

16.1.6 Populating Datasets

A dataset is a container; therefore, you need to fill it with data. When you populate a dataset, various events are raised, constraint checking applies, and so on. You can populate a dataset in a variety of ways:

Call the Fill method of a data adapter. This causes the adapter to execute an SQL statement or stored procedure and fill the results into a table in the dataset. If the dataset contains multiple tables, you probably have separate data adapters for each table, and must therefore call each adapter's Fill method separately.

Manually populate tables in the dataset by creating DataRow objects and adding them to the table's Rows collection. (You can only do this at run time; you cannot set the Rows collection at design time.) For more information, see Adding Data to a Table.

Read an XML document or stream into the dataset. For more information, see the ReadXml method.

Merge (copy) the contents of another dataset. This scenario can be useful if your application gets datasets from different sources (different XML Web services, for example), but needs to consolidate them into a single dataset. For more information, see the DataSet.Merge method.

16.1.7 Record Position and Navigation in Datasets

Because a dataset is a fully disconnected container for data, datasets (unlike ADO recordsets) do not need or support the concept of a current record. Instead, all records in the dataset are available.Because there is no current record, there is no specific property that points to a current record and there are no methods or properties for moving from one record to another. (In contrast, ADO recordsets support an absolute record position and methods to move from one record to the next.) You can access individual tables in the dataset as objects; each table exposes a collection of rows. You can treat this like any collection, accessing rows via the collection's index or using collection-specific statements in your programming language.

Note If you are binding controls in a Windows Forms to a dataset, you can use the form's binding architecture to simplify access to individual records.

16.1.8 Related Tables and DataRelation Objects

If you have multiple tables in a dataset, the information in the tables might be related. A dataset has no inherent knowledge of such relationships; to work with data in related tables, therefore, you can create DataRelation objects that describe the relations between the tables in the dataset. **DataRelation** objects can be used to programmatically fetch related child records for a parent record, and a parent record from a child record.

For example, imagine customer and order data such as that in the Northwind database. The Customers table might contain records such as the following:

CustomerID	CompanyName	City
ALFKI	Alfreds Futterkiste	Berlin
ANTON	Antonio Moreno Taquerias	Mexico D.F.
AROUT	Around the Horn	London

The dataset might also contain another table with order information. The Orders table contains a customer ID as a foreign key column. Selecting only some of the columns in the Orders table, it might look like the following:

OrderId	CustomerID	OrderDate
10692	ALFKI	10/03/1997
10702	ALFKI	10/13/1997
10365	ANTON	11/27/1996
10507	ANTON	4/15/1997

Because each customer can have more than one order, there is a one-to-many relationship between customers and orders. For example, in the table above, the customer ALFKI has two orders.

You can use a **DataRelation** object to get related records from a child or parent table. For example, when working with the record describing the customer ANTON, you can get the collection of records describing the orders for that customer. Similarly, if you are working with the record describing order number 10507, you can use a **DataRelation** object to get the record describing the customer for that order (ANTON).

16.2 Constraints

As in most databases, datasets support constraints as a way to ensure the integrity of data. Constraints are rules that are applied when rows are inserted, updated, or deleted in a table. You can define two types of constraints:

A unique constraint that checks that the new values in a column are unique in the table.

A foreign-key constraint that defines rules for how related child records should be updated when a record in a master table is updated or deleted.

In a dataset, constraints are associated with individual tables (foreign-key constraints) or columns (a unique constraint, one that guarantees that column values are unique). Constraints are implemented as objects of type **UniqueConstraint** or **ForeignKeyConstraint**. They are then added to a table's Constraints collection. A unique constraint can alternatively be specified by simply setting a data column's Unique property to **true**.

The dataset itself supports a Boolean EnforceConstraints property that specifies whether constraints will be enforced or not. By default, this property is set to **true**. However, there are times when it is useful to temporarily turn constraints off. Most often, this is when you are changing a record in such a way that it will temporarily cause an invalid state. After completing the change (and thereby returning to a valid state), you can re-enable constraints.

In Visual Studio, you create constraints implicitly when you define a dataset. By adding a primary key to a dataset, you implicitly create a unique constraint for the primary-key column. You can specify a unique constraint for other columns by setting their Unique property to **true**.

You create foreign-key constraints by creating a **DataRelation** object in a dataset. In addition to allowing you to programmatically get information about related records, a **DataRelation** object allows you to define foreign-key constraint rules.

16.2.1 Updating Datasets and Data Stores

When changes are made to records in the dataset, the changes have to be written back to the database. To write changes from the dataset to the database, you call the **Update** method of the data adapter that communicates between the dataset and its corresponding data source.

The **DataRow** class used to manipulate individual records includes the **RowState** property, whose values indicate whether and how the row has been changed since the data table was first loaded from the database. Possible values include **Deleted**, **Modified**, **New**, and **Unchanged**. The **Update** method examines the value of the **RowState** property to determine which records need to be written to the database and what specific database command (add, edit, delete) should be invoked.

16.3 Preprocessor

This section discusses the C# language's preprocessor directives:

- **#if**
- **#else**
- **#elif**

- **#endif**
- **#define**
- **#undef**
- **#warning**
- **#error**
- **#line**
- **#region**
- **#endregion**

While the compiler does not have a separate preprocessor, the directives described in this section are processed as if there was one; these directives are used to aid in conditional compilation. Unlike C and C++ directives, you cannot use these directives to create macros.A preprocessor directive must be the only instruction on a line.

#if lets you begin a conditional directive, testing a symbol or symbols to see if they evaluate to true. If they do evaluate to true, the compiler evaluates all the code between the #if and the next directive.

#if symbol [operator symbol]...where: symbol

The name of the symbol you want to test. You can also use true and false. symbol can be prefaced with the negation operator. For example, !true will evaluate to false. operator (optional)

You can use the following operators to evaluate multiple symbols:

== (equality)
!= (inequality)
&& (and)
|| (or)

You can group symbols and operators with parentheses.

#if, along with the #else, #elif, #endif, #define, and #undef directives, lets you include or exclude code based on the condition of one or more symbols. This can be most useful when compiling code for a debug build or when compiling for a specific configuration.

A conditional directive beginning with a #if directive must explicitly be terminated with a #endif directive.

Example

```
// preprocessor_if.cs
#define DEBUG
#define VC_V6
using System;
public class MyClass
{
  public static void Main()
  {

    #if (DEBUG && !VC_V6)
      Console.WriteLine("DEBUG is defined");
    #elif (!DEBUG && VC_V6)
      Console.WriteLine("VC_V6 is defined");
    #elif (DEBUG && VC_V6)
      Console.WriteLine("DEBUG and VC_V6 are defined");
    #else
      Console.WriteLine("DEBUG and VC_V6 are not defined");
    #endif
  }
}
```

Output

```
DEBUG and VC_V6 are defined
```

#else lets you create a compound conditional directive, such that, if none of the expressions in the preceding #if or (optional) #elif directives did not evaluate to true, the compiler will evaluate all code between #else and the subsequent #endif.

#else

#endif must be the next preprocessor directive after #else.

#elif lets you create a compound conditional directive. The #elif expression will be evaluated if neither the preceding #if nor any preceding (optional) #elif directive expressions evaluate to true. If a #elif expression evaluates to true, the compiler evaluates all the code between the #elif and the next directive.

#elif symbol [operator symbol]... where: symbol

The name of the symbol you want to test. You can also use true and false. symbol can be prefaced with the negation operator. For example, !true will evaluate to false.

operator (optional) You can use the following operators to evaluate multiple symbols:

== (equality)

!= (inequality)

&& (and)

|| (or)

You can group symbols and operators with parentheses.

#elif is equivalent to using:

#else

#if

But using #elif is simpler because each #if requires a #endif, whereas a #elif can be used without a matching #endif.

Example

See #if for an example of how to use #elif.

#endif specifies the end of a conditional directive, which began with the #if directive.

#endif

A conditional directive, beginning with a #if directive, must explicitly be terminated with a #endif directive.

Example

See #if for an example of how to use #endif.

#define lets you define a symbol, such that, by using the symbol as the expression passed to the #if directive, the expression will evaluate to true.

#define symbolwhere: symbol

The name of the symbol to define.

Symbols can be used to specify conditions for compilation. You can test for the symbol with either #if or #elif. You can also use the conditional attribute to perform conditional compilation.

You can define a symbol, but you cannot assign a value to a symbol. The #define directive must appear in the file before you use any instructions that are not also directives.

You can also define a symbol with the /define compiler option. You can undefine a symbol with #undef.

A symbol that you define with /define or with #define does not conflict with a variable of the same name. That is, a variable name should not be passed to a preprocessor directive and a symbol can only be evaluated by a preprocessor directive.

The scope of a symbol created with #define is the file in which it was defined.

Example

See #if for an example of how to use #define.

#undef lets you undefine a symbol, such that, by using the symbol as the expression in a #if directive, the expression will evaluate to false.

#undef symbol where: symbol

The name of the symbol you want to undefine.

A symbol can be defined either with the #define directive or the /define compiler option. The #undef directive must appear in the file before you use any statements that are not also directives.

Example

```
// preprocessor_undef.cs
// compile with: /d:DEBUG
#undef DEBUG
using System;
public class MyClass
{
  public static void Main()
  {
    #if DEBUG
      Console.WriteLine("DEBUG is defined");
    #else
      Console.WriteLine("DEBUG is not defined");
    #endif
  }
}
```

Output

```
DEBUG is not defined
```

#warning lets you generate a level one warning from a specific location in your code.

#warning text where: text

The text of the warning that should appear in the compiler's output.

A common use of #warning is in a conditional directive. It is also possible to generate a user-defined error with #error.

Example

```
// preprocessor_warning.cs
// CS1030 expected
#define DEBUG
public class MyClass
{
  public static void Main()
  {
    #if DEBUG
    #warning DEBUG is defined
  #endif
  }
}
```

#error lets you generate an error from a specific location in your code.

#error text

where:

text

The text of the error that should appear in the compiler's output.

Remarks

A common use of #error is in a conditional directive. It is also possible to generate a user-defined warning with #warning.

Example

```
// preprocessor_error.cs
// CS1029 expected
#define DEBUG
public class MyClass
{
  public static void Main()
  {
    #if DEBUG
    #error DEBUG is defined
  #endif
  }
}
```

#line lets you modify the compiler's line number and (optionally) the file name output for errors and warnings.

#line [number ["file_name"] | default] where: number

The number you want to specify for the following line in a source code file.

"file_name" (optional)

The file name you want to appear in the compiler output. By default, the actual name of the source code file is used. The file name must be in double quotation marks ("").

default

Resets the line numbering in a file.

#line might be used by an automated, intermediate step in the build process. For example, if the intermediate step removed lines from the original source code file, but if you still wanted the compiler to generate output based on the original line numbering in the file, you could remove lines and then simulate the original line numbering with #line.

A source code file can have any number of #line directives.

Example

```
// preprocessor_line.cs
public class MyClass2
{
  public static void Main()
  {
    #line 200
    int i;   // CS0168 on line 200
    #line default
    char c;   // CS0168 on line 8
  }
}
```

#region lets you specify a block of code that you can expand or collapse when using the outlining feature of the Visual Studio Code Editor.

#region name where: name

The name you want to give to the region, which will appear in the Visual Studio Code Editor.

A #region block must be terminated with a #endregion directive.

A #region block cannot overlap with a #if block. However, a #region block can be nested in a #if block, and a #if block can be nested in a #region block.

Example

```
// preprocessor_region.cs
#region MyClass definition
public class MyClass
{
  public static void Main()
  {
  }
}
#endregion
```

#endregion marks the end of a #region block.

#endregion

16.4 Unsafe code

The use of pointers is rarely required in C#, but there are some situations that require them. As examples, using an unsafe context to allow pointers is warranted by the following cases:

- Dealing with existing structures on disk
- Advanced COM or Platform Invoke scenarios that involve structures with pointers in them
- Performance-critical code

The use of unsafe context in other situations is discouraged. Specifically, an unsafe context should not be used to attempt to "write C code in C#." Code written using an unsafe context cannot be verified to be safe, so it will be executed only when the code is fully trusted.

This tutorial includes the following examples:

- Uses pointers to copy an array of bytes. (Example 1)
- Shows how to call the Windows **ReadFile** function. (Example 2)
- Shows how to print the Win32 version of the executable file. (Example 3

Uses pointers to copy an array of bytes (Example 1)

The following example uses pointers to copy an array of bytes from `src` to `dst`. Compile the example with the /unsafe option.

```
// fastcopy.cs
// compile with: /unsafe
using System;
class Test
{
   // The unsafe keyword allows pointers to be used within
   // the following method:
   static unsafe void Copy(byte[] src, int srcIndex,
      byte[] dst, int dstIndex, int count)
   {
      if (src == null || srcIndex < 0 ||
         dst == null || dstIndex < 0 || count < 0)
      {
         throw new ArgumentException();
      }
      int srcLen = src.Length;
      int dstLen = dst.Length;
      if (srcLen - srcIndex < count ||
         dstLen - dstIndex < count)
      {
         throw new ArgumentException();
      }
      // The following fixed statement pins the location of
      // the src and dst objects in memory so that they will
      // not be moved by garbage collection.
      fixed (byte* pSrc = src, pDst = dst)
      {
         byte* ps = pSrc;
         byte* pd = pDst;
         // Loop over the count in blocks of 4 bytes, copying an
```

```
            // integer (4 bytes) at a time:
            for (int n = count >> 2; n != 0; n--)
            {
               *((int*)pd) = *((int*)ps);
               pd += 4;
               ps += 4;
            }
            // Complete the copy by moving any bytes that weren't
            // moved in blocks of 4:
            for (count &= 3; count != 0; count--)
            {
               *pd = *ps;
               pd++;
               ps++;
            }
         }
      }

      static void Main(string[] args)
      {
         byte[] a = new byte[100];
         byte[] b = new byte[100];
         for(int i=0; i<100; ++i)
            a[i] = (byte)i;
         Copy(a, 0, b, 0, 100);
         Console.WriteLine("The first 10 elements are:");
         for(int i=0; i<10; ++i)
            Console.Write(b[i] + "{0}", i < 9 ? " " : "");
         Console.WriteLine("\n");
      }
   }
```

Example Output

```
The first 10 elements are:
0 1 2 3 4 5 6 7 8 9
```

Code Discussion

- Notice the use of the **unsafe** keyword, which allows pointers to be used within the `Copy` method.
- The **fixed** statement is used to declare pointers to the source and destination arrays. It pins the location of the `src` and `dst` objects in memory so that they will not be moved by garbage collection. The objects will be unpinned when the **fixed** block completes

- The reason why unsafe code is beneficial here is that it allows copying the array in chunks of 4 bytes at a time, and directly by incrementing pointers, thereby getting rid of array bounds checks and address calculations in each iteration.

Shows how to call the Windows ReadFile function.(Example 2)

This example shows how to call the Windows **ReadFile** function from the *Platform SDK*, which requires the use of an unsafe context because the read buffer requires a pointer as a parameter.

```
// readfile.cs
// compile with: /unsafe
// arguments: readfile.txt

// Use the program to read and display a text file.
using System;
using System.Runtime.InteropServices;
using System.Text;

class FileRead
{
  const uint GENERIC_READ = 0x80000000;
  const uint OPEN_EXISTING = 3;
  int handle;

  public FileRead(string filename)
  {
     // opens the existing file...
     handle = CreateFile(filename,
                  GENERIC_READ,
                  0,
                  0,
                  OPEN_EXISTING,
                  0,
                  0);
  }

  [DllImport("kernel32", SetLastError=true)]
  static extern unsafe int CreateFile(
    string filename,
    uint desiredAccess,
    uint shareMode,
    uint attributes,   // really SecurityAttributes pointer
    uint creationDisposition,
    uint flagsAndAttributes,
    uint templateFile);

  [DllImport("kernel32", SetLastError=true)]
  static extern unsafe bool ReadFile(int hFile,
    void* lpBuffer, int nBytesToRead,
    int* nBytesRead, int overlapped);

  [DllImport("kernel32", SetLastError=true)]
  static extern int GetLastError();

  public unsafe int Read(byte[] buffer, int index, int count)
  {
    int n = 0;
    fixed (byte* p = buffer)
    {
      ReadFile(handle, p + index, count, &n, 0);
    }
    return n;
  }
}

class Test
{
  public static void Main(string[] args)
  {
     FileRead fr = new FileRead(args[0]);

     byte[] buffer = new byte[128];
     ASCIIEncoding e = new ASCIIEncoding();

    // loop through, read until done...
    Console.WriteLine("Contents");
    while (fr.Read(buffer, 0, 128) != 0)
    {
       Console.Write("{0}", e.GetString(buffer));
    }
  }
}
```

Example Input

The following input from readfile.txt produces the output shown in Example Output when you compile and run this sample.

```
line 1
line 2
```

Example Output

```
Contents
line 1
line 2
```

Code Discussion

The byte array passed into the **Read** function is a managed type. This means that the common language runtime garbage collector could relocate the memory used by the array at will. The **fixed** statement allows you to both get a pointer to the memory used by the byte array and to mark the instance so that the garbage collector won't move it.

At the end of the **fixed** block, the instance will be marked so that it can be moved. This capability is known as declarative pinning. The nice part about pinning is that there is very little overhead unless a garbage collection occurs in the **fixed** block, which is an unlikely occurrence.

Shows how to print the Win32 version of the executable file.Example 3

This example reads and displays the Win32 version number of the executable file, which is the same as the assembly version number in this example. The executable file, in this example, is printversion.exe. The example uses the *Platform SDK* functions VerQueryValue, GetFileVersionInfoSize, and GetFileVersionInfo to retrieve specified version information from the specified version-information resource.

This example uses pointers because it simplifies the use of methods whose signatures use pointers to pointers, which are common in the Win32 APIs.

```
// printversion.cs
// compile with: /unsafe
using System;
using System.Reflection;
using System.Runtime.InteropServices;

// Give this assembly a version number:
[assembly:AssemblyVersion("4.3.2.1")]

public class Win32Imports
{
  [DllImport("version.dll")]
  public static extern bool GetFileVersionInfo (string sFileName,
    int handle, int size, byte[] infoBuffer);
  [DllImport("version.dll")]
  public static extern int GetFileVersionInfoSize (string sFileName,
    out int handle);

  // The 3rd parameter - "out string pValue" - is automatically
  // marshaled from Ansi to Unicode:
  [DllImport("version.dll")]
  unsafe public static extern int VerQueryValue (byte[] pBlock,
    string pSubBlock, out string pValue, out uint len);
  // This VerQueryValue overload is marked with 'unsafe' because
  // it uses a short*:
  [DllImport("version.dll")]
  unsafe public static extern int VerQueryValue (byte[] pBlock,
    string pSubBlock, out short *pValue, out uint len);
}

public class C
{
  // Main is marked with 'unsafe' because it uses pointers:
  unsafe public static int Main ()
  {
    try
    {
      int handle = 0;
      // Figure out how much version info there is:
      int size =
        Win32Imports.GetFileVersionInfoSize("printversion.exe",
        out handle);
      // Be sure to allocate a little extra space for safety:
      byte[] buffer = new byte[size+2];
      Win32Imports.GetFileVersionInfo ("printversion.exe",
        handle, size, buffer);
      short *subBlock = null;
      uint len = 0;
      // Get the locale info from the version info:
      Win32Imports.VerQueryValue (buffer,
        "\\VarFileInfo\\Translation", out subBlock, out len);
      string spv =
        "\\StringFileInfo\\" + subBlock[0].ToString("X4")
        + subBlock[1].ToString("X4") + "\\ProductVersion";
      byte *pVersion = null;
      // Get the ProductVersion value for this program:
      string versionInfo;
      Win32Imports.VerQueryValue (buffer, spv,
        out versionInfo, out len);
      Console.WriteLine ("ProductVersion == {0}", versionInfo);
    }
    catch (Exception e)
    {
      Console.WriteLine ("Caught unexpected exception " +
        e.ToString() + "\n\n" + e.Message);
    }

    return 0;
  }
}
```

Example Output

```
ProductVersion == 4.3.2.1
```

16.5 Assemblies and Versioning

Assemblies are the building blocks of .NET Framework applications; they form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. An assembly provides the common language runtime with the information it needs to be aware of type implementations. To the runtime, a type does not exist outside the context of an assembly.Assemblies are a fundamental part of programming with the .NET Framework. An assembly performs the following functions:

It contains code that the common language runtime executes. Microsoft intermediate language (MSIL) code in a portable executable (PE) file will not be executed if it does not have an associated assembly manifest. Note that each assembly can have only one entry point (that is, DllMain, WinMain, or Main). It forms a security boundary. An assembly is the unit at which permissions are requested and granted. For more information about security boundaries as they apply to assemblies,

It forms a type boundary. Every type's identity includes the name of the assembly in which it resides. A type called MyType loaded in the scope of one assembly is not the same as a type called MyType loaded in the scope of another assembly. It forms a reference scope boundary. The assembly's manifest contains assembly metadata that is used for resolving types and satisfying resource requests. It specifies the types and resources that are exposed outside the assembly. The manifest also enumerates other assemblies on which it depends. It forms a version boundary. The assembly is the smallest versionable unit in the common language runtime; all types and resources in the same assembly are versioned as a unit. The assembly's manifest describes the version dependencies you specify for any dependent assemblies.

It forms a deployment unit. When an application starts, only the assemblies that the application initially calls must be present. Other assemblies, such as localization resources or assemblies containing utility classes, can be retrieved on demand. This allows applications to be kept simple and thin when first downloaded.

It is the unit at which side-by-side execution is supported.

Assemblies can be static or dynamic.

Static assemblies can include .NET Framework types (interfaces and classes), as well as resources for the assembly (bitmaps, JPEG files, resource files, and so on). Static assemblies are stored on disk in PE files. You can also use the .NET Framework to create dynamic assemblies, which are run directly from memory and are not saved to disk before execution. You can save dynamic assemblies to disk after they have executed.

There are several ways to create assemblies. You can use development tools, such as Visual Studio .NET, that you have used in the past to create .dll or .exe files. You can use tools provided in the .NET Framework SDK to create assemblies with modules created in other development environments. You can also use common language runtime APIs, such as Reflection.Emit, to create dynamic assemblies.

All versioning of assemblies that use the common language runtime is done at the assembly level. The specific version of an assembly and the versions of dependent assemblies are recorded in the assembly's manifest. The default version policy for the runtime is that applications run only with the versions they were built and tested with, unless overridden by explicit version policy in configuration files (the application configuration file, the publisher policy file, and the computer's administrator configuration file).

Note Versioning is done only on assemblies with strong names.

The runtime performs several steps to resolve an assembly binding request:

Checks the original assembly reference to determine the version of the assembly to be bound.

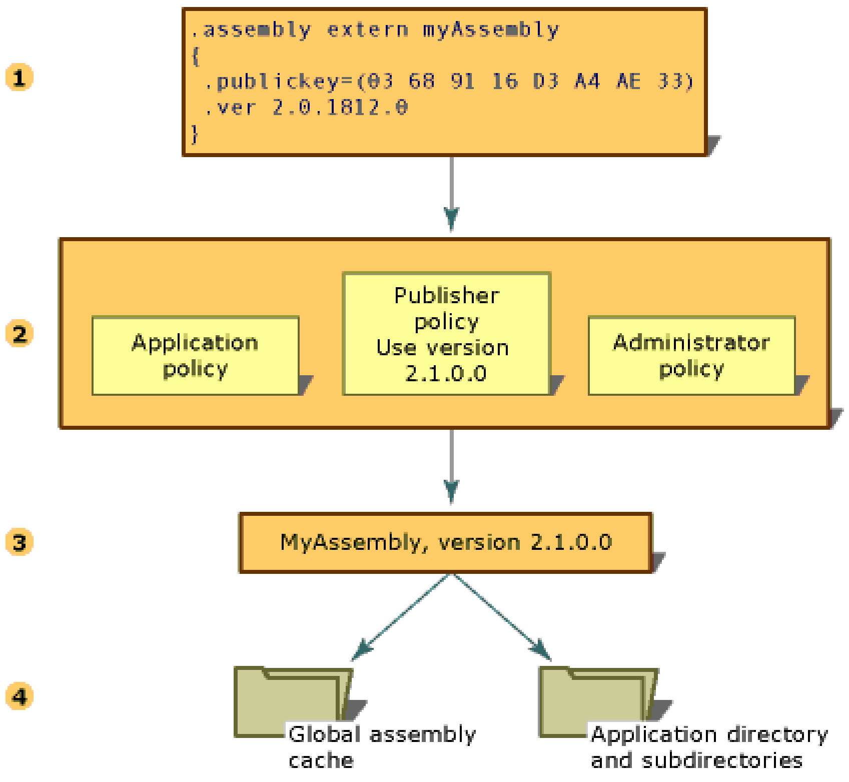
Checks for all applicable configuration files to apply version policy.

Determines the correct assembly from the original assembly reference and any redirection specified in the configuration files, and determines the version that should be bound to the calling assembly.

Checks the global assembly cache, codebases specified in configuration files, and then checks the application's directory and subdirectories using the probing rules

The following illustration shows these steps.

Resolving an assembly binding request



```
.assembly extern myAssembly
{
 .publickey=(03 68 91 16 D3 A4 AE 33)
 .ver 2.0.1812.0
}
```

1
2
Application policy
Publisher policy Use version 2.1.0.0
Administrator policy
3
MyAssembly, version 2.1.0.0
4
Global assembly cache
Application directory and subdirectories

Assemblies contain modules, modules contain types, and types contain members. Reflection provides objects that encapsulate assemblies, modules, and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object. You can then invoke the type's methods or access its fields and properties. Typical uses of reflection include the following:

Use Assembly to define and load assemblies, load modules that are listed in the assembly manifest, and locate a type from this assembly and create an instance of it. Use Module to discover information such as the assembly that contains the module and the classes in the module. You can also get all global methods or other specific, nonglobal methods defined on the module. Use ConstructorInfo to discover information such as the name, parameters, access modifiers (such as public or private), and implementation details (such as abstract or virtual) of a constructor. Use the GetConstructors or GetConstructor method of a Type object to invoke a specific constructor.

Use MethodInfo to discover information such as the name, return type, parameters, access modifiers (such as public or private), and implementation details (such as abstract or virtual) of a method. Use the GetMethods or GetMethod method of a Type object to invoke a specific method. Use FieldInfo to discover information such as the name, access modifiers (such as public or private) and implementation details (such as static) of a field, and to get or set field values. Use EventInfo to discover information such as the name, event-handler data type, custom attributes, declaring type, and reflected type of an event, and to add or remove event handlers. Use PropertyInfo to discover information such as the name, data type, declaring type, reflected type, and read-only or writable status of a property, and to get or set property values. Use ParameterInfo to discover information such as a parameter's name, data type, whether a parameter is an input or output parameter, and the position of the parameter in a method signature.

The classes of the System.Reflection.Emit namespace provide a specialized form of reflection that enables you to build types at run time. Reflection can also be used to create applications called type browsers, which enable users to select types and then view the information about those types.There are other uses for reflection. Compilers for languages such as JScript use reflection to construct symbol tables. The classes in the System.Runtime.Serialization namespace use reflection to access data and to determine which fields to persist. The classes in the System.Runtime.Remoting namespace use reflection indirectly through serialization.

16.6 Marshalling

Marshaling is the process of packaging and unpackaging parameters so a remote procedure call can take place.

Different parameter types are marshaled in different ways. For example, marshaling an integer parameter involves simply copying the value into the message buffer. (Although even in this simple case, there are issues such as byte ordering to deal with in cross-machine calls.) Marshaling an array, however, is a more complex process. Array members are copied in a specific order so that the other side can reconstruct the array exactly. When a pointer is marshaled, the data that the pointer is pointing to is copied following rules and conventions for dealing with nested pointers in structures. Unique functions exist to handle the marshaling of each parameter type.

With standard marshaling, the proxies and stubs are systemwide resources for the interface and they interact with the channel through a standard protocol. Standard marshaling can be used both by standard COM-defined interfaces and by custom interfaces, as follows:

In the case of most COM interfaces, the proxies and stubs for standard marshaling are in-process component objects which are loaded from a systemwide DLL provided by COM in ole32.dll.

In the case of custom interfaces, the proxies and stubs for standard marshaling are generated by the interface designer, typically with MIDL. These proxies and stubs are statically configured in the registry, so any potential client can use the custom interface across process boundaries. These proxies and stubs are loaded from a DLL that is located via the system registry, using the interface ID (IID) for the custom interface they marshal.

An alternative to using MIDL to generate proxies and stubs for custom interfaces, a type library can be generated instead and the system provided, type-library–driven marshaling engine will marshal the interface.

As an alternative to standard marshaling, an interface (standard or custom) can use custom marshaling. With custom marshaling, an object dynamically implements the proxies at run time for each interface that it supports. For any given interface, the object can select COM-provided standard marshaling or custom marshaling. This choice is made by the object on an interface-by-interface basis. Once the choice is made for a given interface, it remains in effect during the object's lifetime. However, one interface on an object can use custom marshaling while another uses standard marshaling.

Custom marshaling is inherently unique to the object that implements it. It uses proxies implemented by the object and provided to the system on request at run time. Objects that implement custom marshaling must implement the_IMarshal interface, whereas objects that support standard marshaling do not.

If you decide to write a custom interface, you must provide marshaling support for it. Typically, you will provide a standard marshaling DLL for the interface you design. You can use the tools contained in the Platform SDK CD to create the proxy/stub code and the proxy/stub DLL. Alternatively, you can use these tools to create a type library which COM will use to do data-driven marshaling (using the data in the type library).

For a client to make a call to an interface method in an object in another process involves the cooperation of several components. The standard proxy is a piece of interface-specific code that resides in the client's process space and prepares the interface parameters for transmittal. It packages, or marshals, them in such a way that they can be re-created and understood in the receiving process. The standard stub, also a piece of interface-specific code, resides in the server's process space and reverses the work of the proxy. The stub unpackages, or unmarshals, the sent parameters and forwards them to the object application. It also packages reply information to send back to the client.

Note Readers more familiar with RPC than COM may be used to seeing the terms client stub and server stub. These terms are analogous to proxy and stub.

The following diagram shows the flow of communication between the components involved. On the client side of the process boundary, the client's method call goes through the proxy and then onto the channel, which is part of the COM library. The channel sends the buffer containing the marshaled parameters to the RPC run-time library, which transmits it across the process boundary. The RPC run time and the COM libraries exist on both sides of the process. The distinction between the channel and the RPC run time is a characteristic of this implementation and is not part of the programming model or the conceptual model for COM client/server objects. COM servers see only the proxy or stub and, indirectly, the channel. Future implementations may use different layers below the channel or no layers.

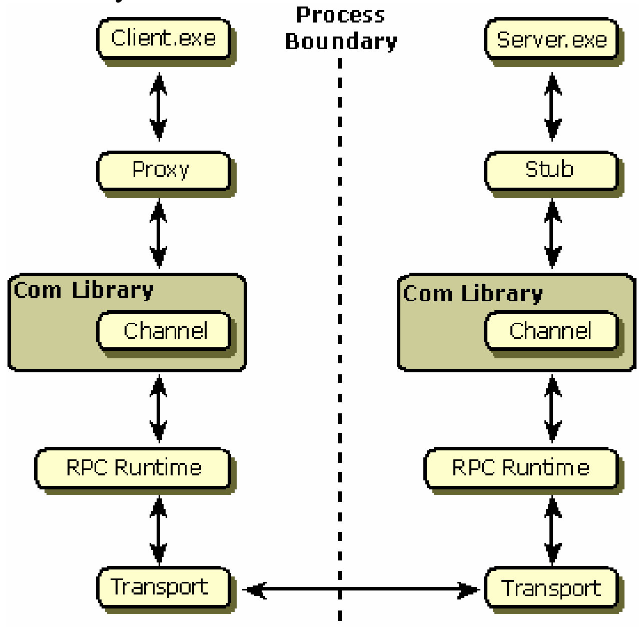


Fig 8

16.7 Remoting

Microsoft .NET Remoting technology provides a framework for distributing objects across different process boundaries and machine boundaries. The .NET Remoting technology offers a powerful yet simple programming model and runtime support that make these interactions transparent.
You should be familiar with the following .NET Remoting concepts before proceeding:

16.7.1 Channels

Typical .NET applications and application domains communicate with each other using messages. For more information, see *What is an application domain?* in the Microsoft .NET Framework FAQ. The .NET Channel Services provide the underlying transport mechanism for this communication.

The .NET Framework supplies the HTTP and TCP channels but third parties can write and plug in their own channels. The HTTP channel uses SOAP by default to communicate, whereas the TCP channel uses Binary payload by default.

Formatters

The .NET serialization formatters encode and decode messages between .NET applications and application domains. There are two native formatters in the .NET runtime, namely Binary (System.Runtime.Serialization.Formatters.Binary)andSOAP (System.Runtime.Serialization.Formatters.Soap).

16.7.2 Remoting Context

A context is a boundary that contains objects that share common runtime properties. Any call from an object in one context to an object in another context passes through a context proxy and can be affected by the policy that the combined context properties enforce. The context of a new object is generally based on metadata attributes on the class.

Classes that can be bound to a context are called context-bound classes. Context-bound classes can be attributed with specialized custom attributes known as context attributes. Since context attributes are completely extensible, you can create and attach these attributes to your class. Objects that are bound to a context derive from System.ContextBoundObject.

16.7.3 .NET Remoting Metadata

The .NET Remoting technology uses metadata to dynamically create proxy objects. The proxy objects that the client creates have the same members as the original class. The implementation of the proxy object just forwards all requests through the .NET Remoting runtime to the original object. The serialization formatter uses metadata to convert method calls to the payload stream and back.

16.7.4 Configuration Files

Configuration files (.Config) specify the Remoting-specific information for a given object. Usually each application domain has its own configuration file. Configuration files help to achieve location transparency. Details specified in the configuration files can also be set programmatically. The main advantage of using configuration files is that configuration information is separate from the code. Consequently, future changes to the configuration file do not require a recompilation of the source file. Configuration files are used both by .NET Remoting client and server objects.

The following table displays some of the combinations of Formatters and Channels:

Formatter	Channel	Used in Duwamish 7.0
SOAP	HTTP	No
Binary	HTTP	Yes
Binary	TCP	No
Network Data Representation (NDR)	DCOM	No

16.8 Web Services

16.8.1 Introduction

A web service allows a site to expose programmatic functionality via the Internet. Web services can accept messages and optionally return replies to those messages.

More about Web Services

Today's sites already expose functionality that allows you to do things such as query a database, book an airline reservation, check the status of an order, etc, but there is no consistent model for allowing you to program against these sites. Web Services can be invoked via HTTP–POST, HTTP–GET and the Simple Object Access Protocol (SOAP). SOAP is a remote procedure call protocol and specification developed by a group of companies including Microsoft and

DevelopMentor. SOAP is based on broadly adopted Internet standards such as XML and typically runs over HTTP

For more information on SOAP please see the SOAP specification on MSDN. Visual Studio.NET will be the formal shipping vehicle for creating rich web services on the .NET platform. With the release of Visual Studio.NET you will be able to create web services using ASP.NET and any language that targets the common–language runtime (CLR) or ATL Server and unmanaged C++. ATL Server is a collection of ATL (Active Template Library) classes that aid the C++ developer in writing native, unmanaged ISAPI extensions and filters.

16.8.2 Modules and Handlers

Instead of the .aspx file extension of an ASP.NET Web page, web services are saved in files with the extension of .asmx. Whenever the ASP.NET runtime receives a request for a file with an .asmx extension, the runtime dispatches the call to the web service handler. This mapping is established in the <httphandlers>section of the config.web files for the machine and individual applications. Config.web files are human–readable, XML files that are used to configure almost every aspect of your web applications.

Handlers are instances of classes that implement the System.Web.IHTTPHandler interface. The IHTTPHandler interface defines two methods, IsReusable and ProcessRequest. The IsReusable method allows an instance of IHTTPHandler to indicate whether it can be recycled and used for another request. The ProcessRequest method takes an HttpContext object as a parameter and is where the developer of a HTTP handler begins to do his work. A particular handler ultimately services inbound requests that are received by the ASP.NET runtime. After a handler is developed, it is configured in the config.web file of the application. A typical config.web file for a machine will have lines similar to the ones below:

```
<httphandlers>
< add verb="*" path="*.asmx" type="System.Web.Services.Protocols.WebServiceHandlerFactory,
System.Web.Services" validate="false" />
 </httphandlers>
```

<httphandlers>section states that for all requests (HTTP verbs such as GET, POST, PUT), if the file being requested has the extension of .asmx, create an instance of the WebServiceHandlerFactory, which lives in the System.Web.Services.dll assembly. If the administrator wanted this handler to only accept the GET verb, he would change the verb property to verb="Get".

Handlers accept requests and produce a response. When the HTTP runtime sees a request for a file with the extension of .aspx the handler that is registered to handle .aspx files is called. In the case of the default ASP.NET installation this handler will be System.Web.UI.PageHandlerFactory. This is the same way in which .asmx files are handled. For the default ASP.NET installation, web services are handled by System.Web.Services.Protocols.WebServiceHandlerFactory.

With this custom handler, ASP.NET is able to use reflection and dynamically create an HTML page describing the service's capabilities and methods. The generated HTML page also provides the user with a way in which to test the web methods within the service. Another advantage of ASP.NET is in publishing Web Service Definition Language (WSDL) contracts.

WSDL is an XML-based grammar for describing the capabilities of web services. WSDL allows a web service to be queried by potential consumers of your service - you can think of it as an XML-based type library made available via the web. For output to be generated, one must only make a HTTP request to the web service file passing in sdl in the querystring

SDL is an XML–based grammar for describing the capabilities of web services. SDL allows a web service to be queried by potential consumers of your service – you can think of it as an XML–based type–library made available via the web. For output to be generated, one must only make a HTTP request to the web service file passing in sdl in the querystring .

Another nice aspect of the web service handler is that it creates a simple test web page for your services. This test page allows you to confirm that everything is working with your web service without having to write your own test client.

Now let us take a look at an example.
First Web Service
The code below is an actual working web service:

```
<%@ WebService Language="C#" class="GreetingService" %>
using System;
using System.Web.Services;
public class GreetingService
{
[WebMethod]
public string SayHello(string Person)
{
```

16.8.3 Returning Complex Types

Our greeting service is only returning a string. Strings as well as most numbers are considered simple types. Web services are not limited to these simple types for return values or inbound parameters. Our next web service will demonstrate how to return a complex type. For this we will create a web service that returns the date and time from the server. We could just return the date and time within a string type and force the caller to parse the string, but this wouldn't be ideal. Instead, we are going to create a LocalTime struct that will be returned to the caller. For those people that may be unfamiliar with structs, they are synonymous with VB's user–defined types (UDT). A walkthrough of the code shows us defining a LocalTime struct that will contain all of the date and time information to be returned to the client. Our LocalTime struct holds seven values; Day, Month, Year, Hour, Minute, Seconds, Milliseconds, and Timezone. The struct's definition is below:

```
<%@ WebService Language="C#" class="TimeService" %>
using System;
using System.Web.Services;
public struct LocalTime {
public int Day;
public int Month;
public int Year;
public int Hour;
public int Minute;
public int Seconds;
public int Milliseconds;
public string Timezone; }
public class TimeService {
[WebMethod]
public LocalTime GetTime()
{ LocalTime lt = new LocalTime();
DateTime dt = DateTime.Now;
lt.Day = dt.Day;
lt.Month = dt.Month;
lt.Year = dt.Year;
lt.Hour = dt.Hour;
lt.Minute = dt.Minute;
lt.Seconds = dt.Second;
lt.Milliseconds = dt.Millisecond;
lt.Timezone = TimeZone.CurrentTimeZone.StandardName;
return lt; } }
```

16.8.4 Using the service.

When Microsoft's Visual Studio.NET ships it will take care of discovering and importing web services for programmatic use, with effort on par with adding a reference in Visual Basic. In the meantime though, the .NET team has created a console application that takes care of requesting the SDL contract of remote web services and generating a proxy to use the service. In order to use a remote web service a few things need to happen. First you need to know where the web service resides(ex http://www.servername.com/wstest.asmx). Next you need to create a local proxy for the remote service. The proxy allows the developer to work with the remote service as though it were local to the machine. When instantiated, the proxy accepts method calls from your code as though it were the remote service object. Calls are packaged up into SOAP methods and shipped via HTTP to the remote web service. If everything goes correctly, the remote service receives the request, unwraps the envelope, does the work that you asked it to do, then returns the results in a result envelope. Once the proxy receives this returned envelope, it is unwrapped and delivered to your code as a native method call.

16.8.5 The Web Services Utility

The wsdl is a console application that is supplied in Microsoft's .NET SDK. The utility takes care of requesting a SDL contract from a remote web service via HTTP and generating a proxy class for you. Although the Web Services Utility uses C# as its default proxy generation language, any language (including VB and JScript) that implements the ICodeGenerator interface will work. For us to create a proxy class for accessing my web service we use the command below:

```
wsdl "http://localhost/services/TimeService.asmx" /out:TimeServiceProxy.cs
```

The `/c:` parameter informs the utility that I want it to create a proxy for me. The `/pa:` parameter is the path to the SDL contract; this can be a local path, UNC path or URI. when we run this command a .cs file will be generated. TimeServiceProxy.cs An instance of this object is what takes care of accepting method calls, packaging up calls for SOAP, invoking via HTTP and returning the results if any to the caller. Now that you have a proxy you need to compile it using the appropriate compiler depending on the language you chose. The following command assumes that the C# compiler (csc.exe) is in the system's path and that you are working in the directory where your web service's .asmx file resides.

On my system the TimeService.asmx file is located at C:\inetpub\wwwroot\Services. Since we are working with C#, the command is (this should be on one line):

```
csc /out:bin\TimeServiceProxy.dll /t:library /r:system.web.services.dll
/r:system.xml.serialization.dll TimeServiceProxy.cs
```

This command creates a DLL (library) named TimeServiceProxy.dll in the C:\inetpub\wwwroot\Services\bin directory importing the resources System.Web.Services.dll and System.xml.serialization.dll. Once we have the DLL in the bin directory of our ASP.NET application we access the methods of the remote web service as though they were running locally. But this remote web service is not limited to being used only by ASP.NET. Any .NET class can now access our time web service

16.8.6 TimeTest Application

To demonstrate that our time web service is usable by any .NET application, I have created a simple console application in C# that prints out the time from the remote service. This application is compiled into an executable (.exe) as opposed to a library (.dll): The TimeTestApp.exe first creates a new instance of our TimeService class that lives in the bin/TimeServiceProxy.dll assembly. Then a call is made to the GetTime method of the TimeService class (ts). The returned value is stored in a local variable named lt. The lt variable is of the type LocalTime. In case it isn't obvious, I want to point out that the LocalTime object that we are now using was originally defined in our remote .asmx file. The WebServiceUtil was able to create a local definition of the LocalTime struct based on the SDL contract that was generated and returned from the Web Service handler. Next in our code, we call GetTime and then begin to simply construct a couple of local strings that contain our formatted time and date. Then we write out the results using Console.WriteLine:

```
using System;
class MyClass
{
static void Main()
{
TimeService ts = new TimeService();
LocalTime lt = ts.GetTime();
string stime = lt.Hour + ":" + lt.Minute + ":" + lt.Seconds + "." +
lt.Milliseconds + " " + lt.Timezone;
string sdate = lt.Month + "/" + lt.Day + "/" + lt.Year;
Console.WriteLine("The remote date is: " + sdate);
Console.WriteLine("The remote time is: " + stime);
}
}
```

To compile the TimeTest application use the following command:

```
csc /r:system.web.services.dll /r:TimeServiceProxy.dll TimeTestApp.cs
```

This command will create an executable named TimeTestApp.exe in the local directory. We could have been explicit in telling the C# compiler that we wanted an executable, but the compiler creates executables (/target: exe) by default. Another item that should be noted is that although ASP.NET applications look in the bin directory within an ASP.NET application directory to resolve references, non–ASP.NET applications first look in the current directory and then in the system path to resolve assembly references.

16.8.7 .NET Coding Guidelines

Coding techniques incorporate many facets of software development. Although they usually have no impact on the functionality of the application, they contribute to an improved comprehension of source code. All forms of source code are considered here, including programming, scripting, markup and query languages.

The coding techniques defined here are not proposed to form an inflexible set of coding standards. Rather, they are meant to serve as a guide for developing a coding standard for a specific software project.

The coding techniques are divided into three sections:

Names

Comments

Format

The naming scheme is one of the most influential aids to understanding the logical flow of an application. A name should tell "what" rather than "how." By avoiding names that expose the underlying implementation, which can change, you preserve a layer of abstraction that simplifies the complexity. For example, you could use `GetNextStudent()` instead of `GetNextArrayElement()`.

A tenet of naming is that difficulty in selecting a proper name may indicate that you need to further analyze or define the purpose of an item. Make names long enough to be meaningful but short enough to avoid verbosity. Programmatically, a unique name serves only to differentiate one item from another. Expressive names function as an aid to a human reader; therefore, it makes sense to provide a name that a human reader can comprehend. However, be certain that the chosen names are in compliance with the applicable language's rules and standards.

The following points are recommended naming techniques.

Routines

Avoid elusive names that are open to subjective interpretation, such as AnalyzeThis() for a routine, or xxK8 for a variable. Such names contribute to ambiguity more than abstraction. In object-oriented languages, it is redundant to include class names in the name of class properties, such as Book.BookTitle. Instead, use Book.Title. Use the verb-noun method for naming routines that perform some operation on a given object, such as CalculateInvoiceTotal().

In languages that permit function overloading, all overloads should perform a similar function. For those languages that do not permit function overloading, establish a naming standard that relates similar functions.

Variables

Append computation qualifiers (Avg, Sum, Min, Max, Index) to the end of a variable name where appropriate. Use complementary pairs in variable names, such as min/max, begin/end, and open/close.

Since most names are constructed by concatenating several words, use mixed-case formatting to simplify reading them. In addition, to help distinguish between variables and routines, use Pascal casing (CalculateInvoiceTotal) for routine names where the first letter of each word is capitalized. For variable names, use camel casing (documentFormatType) where the first letter of each word except the first is capitalized.

Boolean variable names should contain Is which implies Yes/No or True/False values, such as fileIsFound. Avoid using terms such as Flag when naming status variables, which differ from Boolean variables in that they may have more than two possible values. Instead of documentFlag, use a more descriptive name such as documentFormatType.

Even for a short-lived variable that may appear in only a few lines of code, still use a meaningful name. Use single-letter variable names, such as i, or j, for short-loop indexes only. Do not use literal numbers or literal strings, such as For i = 1 To 7. Instead, use named constants, such as For i = 1 To NUM_DAYS_IN_WEEK for ease of maintenance and understanding.

Tables

When naming tables, express the name in the singular form. For example, use Employee instead of Employees. When naming columns of tables do not repeat the table name; for example, avoid a field called EmployeeLastName in a table called Employee. Do not incorporate the data type in the name of a column. This will reduce the amount of work should it become necessary to change the data type later.

Microsoft SQL Server

Do not prefix stored procedures with sp, which is a prefix reserved for identifying system stored procedures.

Do not prefix user-defined functions with fn_, which is a prefix reserved for identifying built-in functions.

Do not prefix extended stored procedures with xp_, which is a prefix reserved for identifying system extended stored procedures.

16.8.8 Miscellaneous

Minimize the use of abbreviations, but use those that you have created consistently. An abbreviation should have only one meaning and likewise, each abbreviated word should have only one abbreviation. For example, if you use min to abbreviate minimum, do so everywhere and do not use min to also abbreviate minute.

When naming functions, include a description of the value being returned, such as GetCurrentWindowName().

File and folder names, like procedure names, should accurately describe their purpose.

Avoid reusing names for different elements, such as a routine called ProcessSales() and a variable called iProcessSales.

Avoid homonyms, such as write and right, when naming elements to prevent confusion during code reviews.

When naming elements, avoid commonly misspelled words. Also, be aware of differences that exist between regional spellings, such as color/colour and check/cheque.

Avoid typographical marks to identify data types, such as $ for strings or % for integers.

Software documentation exists in two forms, external and internal. External documentation, such as specifications, help files, and design documents, is maintained outside of the source code. Internal documentation is comprised of comments that developers write within the source code at development time.

Despite the availability of external documentation, source code listings should be able to stand on their own because hard-copy documentation can be misplaced. External documentation should consist of specifications, design documents, change requests, bug history, and the coding standard used.

One challenge of internal software documentation is ensuring that the comments are maintained and updated in parallel with the source code. Although properly commenting source code serves no purpose at run time, it is invaluable to a developer who must maintain a particularly intricate or cumbersome piece of software.

The following points are recommended commenting techniques.

If developing in C#, use the XML Documentation feature. For more information, see: XML Documentation.

When modifying code, always keep the commenting around it up to date.

At the beginning of every routine, it is helpful to provide standard, boilerplate comments, indicating the routine's purpose, assumptions, and limitations. A boilerplate comment should be a brief introduction that explains why it exists and what it can do.

Avoid adding comments at the end of a line of code; end-line comments make code more difficult to read. However, end-line comments are appropriate when annotating variable declarations, in which case, align all end-line comments at a common tab stop.

Avoid clutter comments, such as an entire line of asterisks. Instead, use white space to separate comments from code.

Avoid surrounding a block comment with a typographical frame. It may look attractive, but it is difficult to maintain.

Prior to deployment, remove all temporary or extraneous comments to avoid confusion during future maintenance work.

If you need comments to explain a complex section of code, examine the code to determine if you should rewrite it. If at all possible, do not document bad code — rewrite it. Although performance should not typically be sacrificed to make the code simpler for human consumption, a balance must be maintained between performance and maintainability.

Use complete sentences when writing comments. Comments should clarify the code, not add ambiguity.

Comment as you code because you will not likely have time to do it later. Also, should you get a chance to revisit code you have written, that which is obvious today probably will not be obvious six weeks from now.

Avoid superfluous or inappropriate comments, such as humorous sidebar remarks.

Use comments to explain the intent of the code. They should not serve as inline translations of the code.

Comment anything that is not readily obvious in the code.

To prevent recurring problems, always use comments on bug fixes and work-around code, especially in a team environment.

Use comments on code that consists of loops and logic branches. These are key areas that will assist source code readers.

Throughout the application, construct comments using a uniform style with consistent punctuation and structure.

Separate comments from comment delimiters with white space. Doing so will make comments obvious and easy to locate when viewed without color clues.

16.8.9 Format

Formatting makes the logical organization of the code obvious. Taking the time to ensure that the source code is formatted in a consistent, logical manner is helpful to you and to other developers who must decipher the source code.

The following points are recommended formatting techniques.

Establish a standard size for an indent, such as four spaces, and use it consistently. Align sections of code using the prescribed indentation.

Use a monotype font when publishing hard-copy versions of the source code.

Align open and close braces vertically where brace pairs align, such as:

```
for (i = 0; i < 100; i++)
{
  ...
}
```

You can also use a slanting style, where open braces appear at the end of the line and close braces appear at the beginning of the line, such as:

```
for (i = 0; i < 100; i++){
  ...
}
```

Whichever style you choose, use that style throughout the source code.

Indent code along the lines of logical construction. Without indenting, code becomes difficult to follow, such as:

```
If ... Then
If ... Then
...
Else
End If
Else
...
End If
```

Indenting the code yields easier-to-read code, such as:

```
If ... Then
  If ... Then
  ...
  Else
  ...
  End If
Else
...
End If
```

Establish a maximum line length for comments and code to avoid having to scroll the source code editor and to allow for clean hard-copy presentation.

Use spaces before and after most operators when doing so does not alter the intent of the code. For example, an exception is the pointer notation used in C++.

Use white space to provide organizational clues to source code. Doing so creates "paragraphs" of code, which aid the reader in comprehending the logical segmenting of the software.

When a line is broken across several lines, make it obvious that it is incomplete without the following line by placing the concatenation operator at the end of each line instead of at the beginning.

Where appropriate, avoid placing more than one statement per line. An exception is a loop in C, C++, C#, or JScript, such as for (i = 0; i < 100; i++). When writing HTML, establish a standard format for tags and attributes, such as all uppercase for tags and all lowercase for attributes. As an alternative, adhere to the XHTML specification to ensure all HTML documents are valid. Although there are file size trade-offs to consider when creating Web pages, use quoted attribute values and closing tags to ease maintainability. When writing SQL statements, use all uppercase for keywords and mixed case for database elements, such as tables, columns, and views.

Divide source code logically between physical files.

Put each major SQL clause on a separate line so statements are easier to read and edit, for example:

```
SELECT FirstName, LastName
FROM Customers
WHERE State = 'WA'
```

Break large, complex sections of code into smaller, comprehensible modules.

16.9 Introduction to ASP .NET Web applications using Visual Studio

Visual Studio .NET allows you to create applications that leverage the power of the World Wide Web. This includes everything from a traditional Web site that serves HTML pages, to fully featured business applications that run on an intranet or the Internet, to sophisticated business-to-business applications providing Web-based components that can exchange data using XML.

This topic provides an overview of the types of Web applications you can create in Visual Studio, how they relate to one another and to other Visual Studio technologies, and suggestions about where to learn more about the types of Web applications you want to create.

Note For details about the relative advantages of Web applications and Windows applications, see *Windows Forms and Web Forms Recommendations.*

16.9.1 Visual Studio ASP.NET Web Applications

A Visual Studio Web application is built around ASP.NET. ASP.NET is a platform — including design-time objects and controls and a run-time execution context — for developing and running applications on a Web server.

ASP.NET in turn is part of the .NET Framework, so that it provides access to all of the features of that framework. For example, you can create ASP.NET Web applications using any .NET programming language (Visual Basic, C#, Managed Extensions for C++, and many others) and .NET debugging facilities. You access data using ADO.NET. Similarly, you can access operating system services using .NET Framework classes, and so on.

ASP.NET Web applications run on a Web server configured with Microsoft Internet Information Services (IIS). However, you do not need to work directly with IIS. You can program IIS facilities using ASP.NET classes, and Visual Studio handles file management tasks such as creating IIS applications when needed and providing ways for you to deploy your Web applications to IIS.

16.9.2 Where Does Visual Studio Fit In?

As with any .NET application, if you have the .NET Framework, you can create ASP.NET applications using text editors, a command-line compiler, and other simple tools. You can copy your files manually to IIS to deploy the application.

Alternatively, you can use Visual Studio. When you use Visual Studio to create Web applications, you are creating essentially the same application that you could create by hand. That is, Visual Studio does not create a different kind of Web application; the end result is still an ASP.NET Web application.

The advantage of using Visual Studio is that it provides tools that make application development much faster, easier, and more reliable. These tools include:

Visual designers for Web pages with drag-and-drop controls and code (HTML) views with syntax checking. Code-aware editors that include statement completion, syntax checking, and other IntelliSense features.

16.9.3 Integrated compilation and debugging.

Project management facilities for creating and managing application files, including deployment to local or remote servers. If you have used Visual Studio before, these kind of features will seem familiar, because they are similar to features that you get for creating applications in previous versions of Visual Basic and Visual C++. With Visual Studio .NET you can use these kind of features to create ASP.NET Web applications.

16.9.4 Elements of ASP.NET Web Applications

Creating ASP.NET Web applications involves working with many of the same elements you use in any desktop or client-server application. These include:

Project management features When creating an ASP.NET Web application, you need to keep track of the files you need, which ones need to be compiled, and which need to be deployed.

User interface Your application typically presents information to users; in an ASP.NET Web application, the user interface is presented in Web Forms pages, which send output to a browser. Optionally, you can create output tailored for mobile devices or other Web appliances.

Components Many applications include reusable elements containing code to perform specific tasks. In Web applications, you can create these components as XML Web services, which makes them callable across the Web from a Web application, another XML Web service, or a Windows Form.

Data : Most applications require some form of data access. In ASP.NET Web applications, you can use ADO.NET, the data services that are part of the .NET Framework.

Security, performance, and other infrastructure features As in any application, you must implement security to prevent unauthorized use, test and debug the application, tune its performance, and perform other tasks not directly related to the application's primary function.

The following diagram provides an overview of how the pieces of ASP.NET Web applications fit together and fit into the broader context of the .NET Framework.

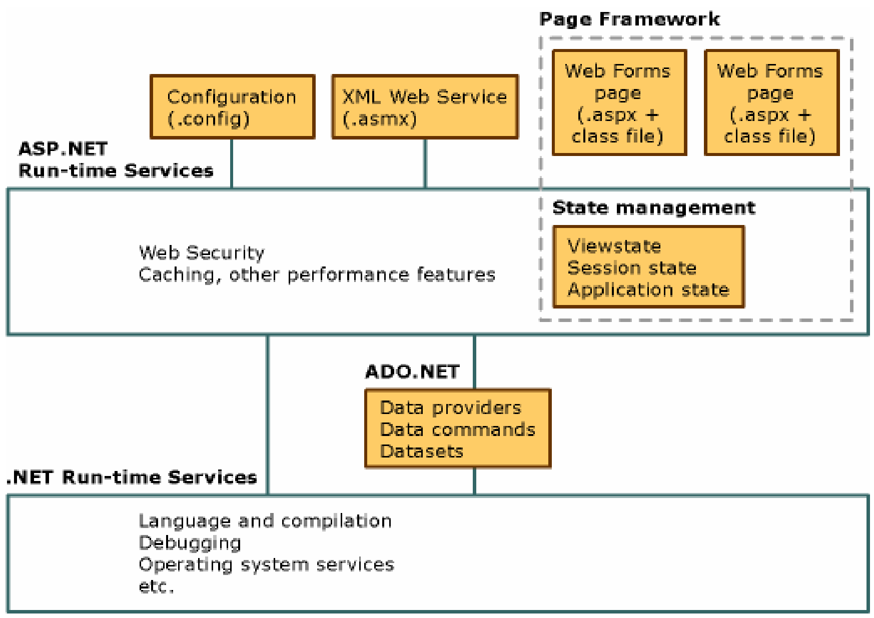


Fig 9 overview of ASP.NET Web applications

The sections below outline the different elements of an ASP.NET Web application and provide links to more information about how to work with each element in Visual Studio.

16.9.5 Overview: ASP.NET

The information in this topic provides only an introduction to ASP.NET.

Project Management: ASP.NET Web Application Projects and Deployment

To work with an ASP.NET Web application using Visual Basic or Visual C#, you use the ASP.NET Web Application project template. As with other Visual Studio projects, a Web application project is a central repository for all the information required to design, run, and manage the application.

When you create a Web application project, Visual Studio creates the necessary files and folders on the server, sets the appropriate security settings on them, and creates the IIS application. Each Web application can maintain a Web.config file that follows the format and conventions of .NET configuration files. In ASP.NET, the configuration file allows you to establish project-specific settings for security, compilation options, tracing, error handling, and more.

For details about how to create and manage ASP.NET Web application projects in Visual Studio, see the following topics.

Working with Web Projects Provides links to topics that describe Web application projects in Visual Studio.

Web Forms Pages and Projects in Visual Studio Provides details about the types of files created in a Web Application project.

Deployment of a Web Setup Project provides an introduction to deploying Web applications.

ASP.NET Configuration Describes the configuration file system and provides links to topics about the contents of the Web.config file.

User Interface: Web Forms and the ASP.NET Page Framework

HTML pagescan be created s the user interface for the application. HTML pages are generally used for static content. In contrast, Web Forms pages give you a programmable interface that works much like a Visual Basic form, except that the user interface is rendered in a Web browser or other Web device.

Note You can use a Windows Form as the user interface in an application that calls XML Web services. For an example, see Walkthrough: Creating a Distributed Application.

Web Forms pages are built on the ASP.NET Page framework. Each Web Forms page is an object that derives from the ASP.NET Page class, which acts as a container for controls. The page actually consists of two files: an .aspx file that contains the UI elements (static HTML text and control elements) and a class file that contains the code that runs the page. When users request a Web Forms page, the Page framework runs the Web Forms page object and all the individual controls on it. The output of the **Page** class and of the controls is HTML. Because the Web is inherently stateless — by default, components in a Web application are alive only long enough to process a single request — Web applications face challenges in preserving values as the user works with the application. To help, the Page framework includes facilities for managing state. These include page-based "view state" (a method for preserving values in controls) and access to non-page-based state facilities such as Session state (user-specific) and Application state (global to the application). The Page framework supports a set of controls you can use to program user interaction with your Web Forms pages. User actions in the form are captured and processed by the Page framework in a way that lets you treat them as standard events. You can choose from a large selection of controls available in Visual Studio. In addition, you can create your own custom controls.

Note You can also create output for mobile devices. To do so, you use the same ASP.NET page framework, but you create Mobile Web Forms instead of Web Forms pages and use controls specifically designed for mobile devices.

For details about Web Forms pages and how to work with them in Visual Studio, see the following topics:

Web Forms Pages Lists topics that provide an overview of Web Forms, including background on ASP.NET and on the Web Forms code model.

Creating and Managing Web Forms Pages Provides information on how to create, add, and manage Web Forms pages.

Programming Web Forms Provides information on the specific aspects of programming in a Web Forms environment.

16. 10 Web-based Components: XML Web Services

An XML Web service is a component that can be called over a TCP/IP network by other applications. It performs a specific function — anything from calculations and credit card validation to complex order processing— and returns values to the calling application.

What makes XML Web services unique is that they can be called across the Web. XML Web services are invoked using HTTP or SOAP requests and exchange data with other components using XML. As such, they can form an integral part of ASP.NET Web applications, providing services not only to your applications, but to any application that has Web access, making them ideal for business-to-business transactions.

For information about XML Web Services, how they work with ASP.NET applications, and how you can create them in Visual Studio, see Programming the Web with XML Web Services.

16.10.1 Web Application Data Access

Most ASP.NET Web applications involve at least some level of access to data. ASP.NET does not directly include data access facilities. Instead, Web applications use ADO.NET data services.

ADO.NET provides a complete framework for accessing and managing data from a variety of sources, including databases and XML files or streams. ADO.NET includes *providers* — classes that allow you to connect to data sources, execute commands, and read results. You can optionally keep data in a *dataset*, which is a disconnected, in-memory cache.

Data access in Web applications, whether in a Web Forms page or an XML Web service, introduces special challenges:

Statelessness The components of Web applications usually do not preserve state, which makes it impractical to maintain live connections to a data source (or other resources).

Scalability Because Web applications can have user loads that vary substantially over short periods of time, data access has to be designed with scalability in mind.

Web Application Infrastructure: Security, Performance, and More

In addition to giving you the means to create user interface elements and callable components, ASP.NET provides a context for those elements to run in. For example, ASP.NET communicates with IIS to handle requests for Web Forms pages and XML Web services, parse the files, call related components, and so on.

Much of this work occurs beneath the surface, at a level where you do not typically need to program when you are creating desktop applications.

Note Visual Studio provides limited support for working with ASP.NET infrastructure features. For example, you can edit the Web application configuration file (Web.config) using the Visual Studio Code Editor. But the .NET Framework is both pluggable and extensible, providing you with low-level access if necessary.

There are other aspects of Web applications that you must often contend with that are part of the application's infrastructure. These include:

Security You frequently have to authenticate and authorize users of your Web application. There are special problems associated with security in a Web application, because users are getting access to server-based resources and because you have very little control over the client side of the application (the browser or mobile device). ASP.NET includes security features that you can configure and program against in your Web application.

Performance and optimization You can tune your application's performance by caching pages and data. ASP.NET maintains an output cache that stores pages that have been requested before; by specifying caching settings, you can control how long pages are cached and under what circumstances they are refreshed.

Tracing Because Web applications run on the server — often a remote server — they do not have output other than the application output (a Web Forms page, for example). ASP.NET therefore gives you the opportunity to include trace information directly in a Web Forms page. For details about adding tracing to ASP.NET Web applications, with links to topics that address specific trace procedures, see ASP.NET Trace.

16.10.2 Using Web Services for Remoting over the Internet

This program describes a design and implementation (C#) of the Remoting over Internet using the Web Service as a gateway into the Remoting infrastructure. The Web Service Gateway (Custom Remoting Channel) allows to enhance the remoting channel over Internet and its chaining with another heterogeneous channel. Consuming a remote object over Internet is full transparently and it does not require any special implementation from the remoting via intranet. The Web Service Gateway enables to create a logical model of the connectivity between the different platforms and languages. Before than we will go to its implementation details, let us start it with usage and configuration issue. For some demonstration purpose we will use a MSMQ Custom Remoting Channel (*MSMQChannelLib.dll*), Assuming that you have a knowledge of the .NET Remoting and Web Service.

Consuming a remote object over Internet using the Web Service Gateway is very straightforward and it actually requires only to install the following assemblies:

WebServiceChannelLib, this is a Custom Remoting Channel on the client side to forward a remoting message to the Web Service Gateway over Internet (outgoing message).

WebServiceListener, this is a Web Service (gateway) to listen an incoming message from the client side and forward it to the local remoting infrastructure (incoming message).

Note that the above assemblies have to be installed (into the GAC) both on the server and client sides when a remote callback is used. The next step is to configure a server and client host sides. Their configuration are depended from the actually application. Let me assume, we want to call a remote object driven by MSMQ custom channel over internet. Their config files might look like the following snippets:

server.exe.config

```
 <configuration>
 <system.runtime.remoting>
  <application >
   <service>
    <wellknown mode="Singleton" type="MyRemoteObject.RemoteObject, MyRemoteObject"
           objectUri="endpoint" />
   </service>
   <channels>
   <channel type="RKiss.MSMQChannelLib.MSMQReceiver, MSMQChannelLib"
         listener=".\ReqChannel"/>
   <channel type="System.Runtime.Remoting.Channels.Tcp.TcpChannel, System.Runtime.Remoting"
         port="8090" />
   </channels>
  </application>
 </system.runtime.remoting>
</configuration>
```

The above server config file will register two channels to listen an incoming message for the remote well known singleton object.

client.exe.config

This is an example of the client config file to register our Custom Remoting Channel.

```
<configuration>
 <system.runtime.remoting>
```

```
  <application>
   <client >
    <wellknown type="MyRemoteObject.RemoteObject, RemoteObject"
            url="ws://localhost/WebServiceListener/Listener.asmx;
             tcp://localhost:8090/endpoint/RemoteObject" />
   </client>
   <channels>
    <channel type="RKiss.WebServiceChannelLib.Sender, WebServiceChannelLib" mode="soap"/>
   </channels>
  </application>
 </system.runtime.remoting>
</configuration>
```

The ws is a Custom Remoting Client channel to dispatch an IMessage over internet using a binary respectively soap mode formatter. Note that the mode is a CustomChannelProperty and its default value is binary.

web.config

This is a Web Service config file. The following snippet is its part. The Web Service gateway is also a local remoting client, therefore a client (sender) channel is requested to be registered. The following snippet shows a configuration of the two channels - Tcp and MSMQ.

```
<system.runtime.remoting>
 <application >
  <channels>
   <channel type="System.Runtime.Remoting.Channels.Tcp.TcpChannel,
System.Runtime.Remoting"/>
   <channel type="RKiss.MSMQChannelLib.MSMQSender, MSMQChannelLib"
          respond=".\RspChannel" admin=".\AdminChannel" timeout="30" priority="10"/>
  </channels>
 </application>
</system.runtime.remoting>
```

16.10.3 Activating a remote object

The well known remote object (WKO) is activated by its consumer using the GetObject method mechanism. The proxy is created based on the remote object metadata assembly installed in the GAC (see an argument objectType*)*. The remoting channel is selected by the objectUrl argument. The *url* address in this solution has two parts :

connectivity to the Web Service gateway over internet

connectivity to the Remote object over intranet within the Web Service gateway

Between the primary and secondary addresses is a semicolon delimiter as it is shown the below:

```
string objectUrl = @"ws://localhost/WebServiceListener/Listener.asmx;
msmq://./reqchannel/endpoint";
```

Using this objectUrl design pattern allows an easy selection of the web service gateways on the Internet. Note that *the ws* custom remoting channel will trim this primary address and forward only its secondary part. In this solution, the objectUrl represents a physical path of the logical connectivity between the consumer and remote object regardless of how many channels are needed. In this example, the Web Service gateway resides on the *localhost* and it should be replaced by the real machine name.

Finally, the following code snippet shows an activation of the remote object:

```
// activate a remote object
Type objectType = typeof(MyRemoteObject.RemoteObject);
string objectUrl = @"ws://localhost/WebServiceListener/Listener.asmx;
msmq://./reqchannel/endpoint";
RemoteObject ro = (RemoteObject)Activator.GetObject(objectType, objectUrl);
```

Note that a metadata (assembly) of the remote object must be installed into the GAC in the places such as client, Web Service gateway and server host.

That's all for the client/remoteObject plumbing issue over the Internet.

The following pictures shows this connectivity:

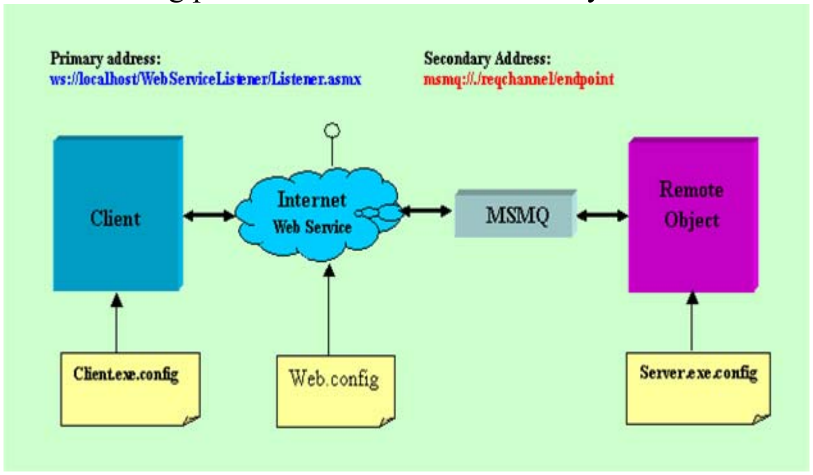
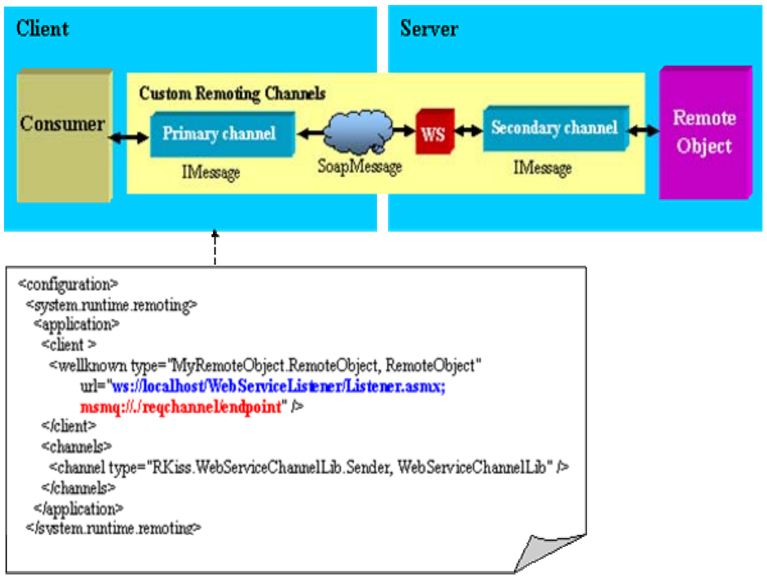


Fig 10



```
<configuration>
  <system.runtime.remoting>
    <application>
      <client >
        <wellknown type="MyRemoteObject.RemoteObject, RemoteObject"
            url="ws://localhost/WebServiceListener/Listener.asmx;
            msmq://./reqchannel/endpoint" />
      </client>
      <channels>
        <channel type="RKiss.WebServiceChannelLib.Sender, WebServiceChannelLib" />
      </channels>
    </application>
  </system.runtime.remoting>
```

Fig11

Now, to understand how the message flows between the heterogeneous channels over the Internet, have a look at the following paragraphs:

16.10. 4 Concept and Design

Concept of the Remoting over Internet is based on dispatching a remoting message over Internet using the **Web Service features as a transport layer**. The following picture shows this solution:

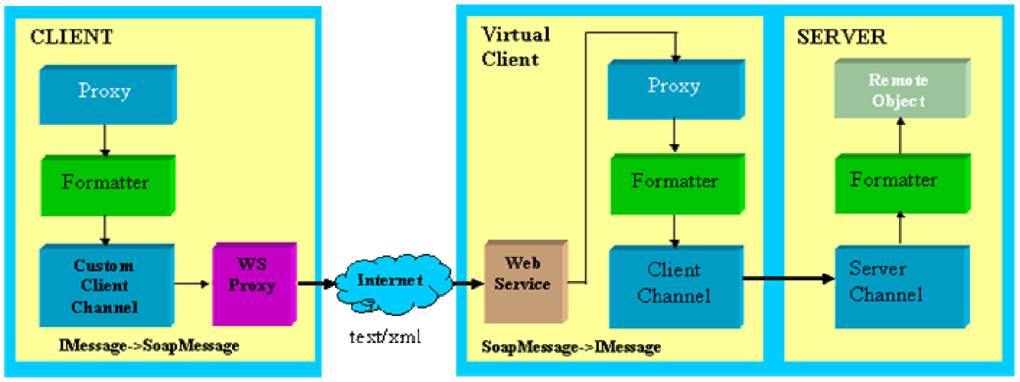


Fig 12

Client activates a remote WKO to obtain its transparent proxy. During this process the custom remoting channel (*ws*) is initiated and inserted into the client channel sink chain. Invoking a method on this proxy, the IMessage is created which it represents a runtime image of the method call at the client side. This IMessage is passed into the channel sink. In our solution to the custom client (sender) channel *ws*. This channel sink has a responsibility to convert IMessage to the SoapMessage in the text/xml format pattern and send it over Internet to the Web Service gateway. The following picture shows this:

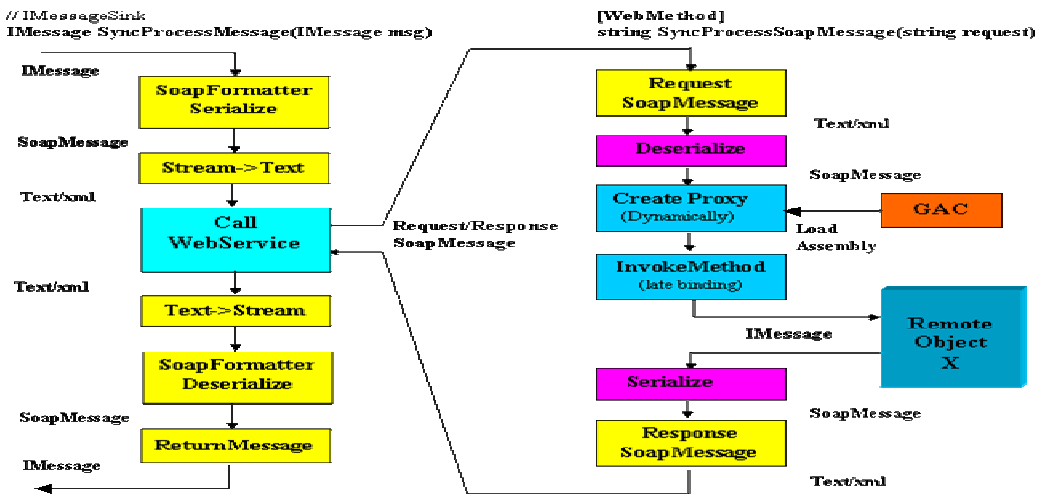


Fig 13

The Web Service gateway has two simply WebMethods, one for the SoapMessage format and the other one for a binary format encoded by base64 into the text string. The first one method enables to use a call from an unknown client, as opposite in the binary formatting message for .NET client.

Lets continue with the IMessage/SoapMessage flows on the Web Service gateway side as it is shown on the above picture. The text/xml formatted SoapMessage sent over Internet might look like the following snippet:

Text/XML formatted SoapMessage Request.

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:a3="http://schemas.microsoft.com/clr/nsassem/MyRemoteObject/MyRemoteObject,
Version=1.0.772.24659, Culture=neutral, PublicKeyToken=ec0dd5142ae7a19b"
xmlns:a1="http://schemas.microsoft.com/clr/ns/System.Runtime.Remoting.Messaging">
<SOAP-ENV:Body>
<System.Runtime.Remoting.Messaging.MethodCall id="ref-1">
<__Uri id="ref-2" xsi:type="SOAP-ENC:string">msmq://./reqchannel/endpoint</__Uri>

<__MethodName id="ref-3" xsi:type="SOAP-ENC:string">get_Id</__MethodName>

<__TypeName id="ref-4" xsi:type="SOAP-ENC:string">MyRemoteObject.RemoteObject,
MyRemoteObject, Version=1.0.772.24659, Culture=neutral,
PublicKeyToken=ec0dd5142ae7a19b</__TypeName>

<__Args href="#ref-5"/>
<__CallContext href="#ref-6"/>
</System.Runtime.Remoting.Messaging.MethodCall>
<SOAP-ENC:Array id="ref-5" SOAP-ENC:arrayType="xsd:ur-type[0]">
</SOAP-ENC:Array>
<a1:LogicalCallContext id="ref-6">
<User href="#ref-8"/>
</a1:LogicalCallContext>
<a3:User id="ref-8">
<FirstName id="ref-9" xsi:type="SOAP-ENC:string">Roman</FirstName>
<LastName id="ref-10" xsi:type="SOAP-ENC:string">Kiss</LastName>
</a3:User>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This string *request* has to be de-serialized back to the SoapMessage object, which is a clone object of the sender's origin. After that, we have an enough information to perform a Method call on the remote object. Conversion of the SoapMessage to IMessage needs to use some trick, therefore there is no class in .NET namespaces to do it. The trick is based on creating a RealProxy wrapper using the remote object type and its endpoint *url* address and overriding an Invoke method of the base RealProxy class. Using the Reflection (late binding) to invoke the remote method, the RealProxy wrapper will catch the IMessage before its processing in the channel sink. Now, the *Invoke* method can perform updating the IMessage by original images such as LocicalCallContext and *url* address. After that, the IMessage is the same like on the client side over Internet. Now it is easy to forward this IMessage to the Message Sink calling its SyncProcessMessage method. The rest is done by a remoting paradigm.

The SyncProcessMessage returns an IMessage which it represents a ReturnMessage from the remote method. Now the process is going to reverse into the text/xml format of the SoapMessage response . I will skip it this process for its simplicity and I will continue on the client custom channel (*ws*) where a response message has been returned. Before that, have a look the text/xml formatted SoapMessage response how it has been sent back to the custom channel over Internet:
Text/XML formatted SoapMessage Response.

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:a3="http://schemas.microsoft.com/clr/nsassem/MyRemoteObject/MyRemoteObject,
Version=1.0.772.24659, Culture=neutral, PublicKeyToken=ec0dd5142ae7a19b"
xmlns:a1="http://schemas.microsoft.com/clr/ns/System.Runtime.Remoting.Messaging">
<SOAP-ENV:Body>
<a1:MethodResponse id="ref-1">
<__Uri xsi:type="xsd:ur-type" xsi:null="1"/>
<__MethodName xsi:type="xsd:ur-type" xsi:null="1"/>
<__MethodSignature xsi:type="xsd:ur-type" xsi:null="1"/>
<__TypeName xsi:type="xsd:ur-type" xsi:null="1"/>
<__Return xsi:type="xsd:int">0</__Return>
<__OutArgs href="#ref-2"/>
<__CallContext href="#ref-3"/>
</a1:MethodResponse>
<SOAP-ENC:Array id="ref-2" SOAP-ENC:arrayType="xsd:ur-type[0]">
</SOAP-ENC:Array>
<a1:LogicalCallContext id="ref-3">
<User href="#ref-5"/>
</a1:LogicalCallContext>
<a3:User id="ref-5">
<FirstName id="ref-6" xsi:type="SOAP-ENC:string">Roman</FirstName>
<LastName id="ref-7" xsi:type="SOAP-ENC:string">Kiss</LastName>
</a3:User>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The result of the remoting call has to be de-serialized into the SoapMessage and then an IMessage can be generated by the function ReturnMessage. This IMessage is returned back to the remoting client infrastructure. In this point, the process of the remoting over Internet is done and the rest is a regular remoting mechanism.

As I mentioned earlier, the Web Service gateway has two methods, one has been described the above using the SoapFormatter and the other one is using a BinaryFormatter.

16.10.5 Using the Binary Formatted Message.

The .NET clients can use a remoting over Internet in more efficient (faster) way using the BinaryFormatter. The design implementation is more straightforward than using the SoapFormatter. The IMessage is serialize/deserialize by the BinaryFormatter to/from memory stream. Than this stream image is encoded/decoded using the Base64 conversion class to/from text string. Note that this text string is not readable. The IMessage is plugged-in into the server remoting infrastructure using the RemotingServices.Connect and RemotingServices.ExecuteMessage functions from the Remoting namespace. Thanks for these functions, they really saved my time.
Limitation note

The limitation of the above solution is done by the Web Service functionality. You cannot handle a distributed transaction over Internet. In this case, the Web Service gateway represents a non-transactional client and remote object is a root of the transaction.

16.10.6 Implementation

The implementation is divided into two assemblies - Custom Client Channel and Web Service gateway. Their implementation is straightforward without using any third party library support. I am going to concentrate only for these parts related to the IMessage processing. More details about the design and implementation of the Custom Remoting Channel can be found it in [1].
WebServiceChannelLib

This is a Custom Client Channel assembly to process an outgoing remoting message over Internet. The Client Message Sink has an implementation both message processing such as SyncProcessMessage and AsyncProcessMessage. Based on the *m_mode* value, the IMessage can be formatted by the SoapFormatter or BinaryFormatter.

The SyncProcessMessage function initiates the Web Service client proxy generated by the VS. There is a small modification in its constructor. This proxy is a generic for any location of the Web service, that's why the url address is pass through its constructor instead of its hard coding. Note that any *Exception* will be converted into the IMessage format and send back to the client.
Sender:

```
// IMessageSink (MethodCall)
public virtual IMessage SyncProcessMessage(IMessage msgReq)
{
  IMessage msgRsp = null;

  try
  {
    msgReq.Properties["__Uri"] = m_ObjectUri;
    Service webservice = new Service(m_outpath);

    if(m_mode == "SOAP")
    {
      // serialize IMessage into the stream (SoapMessage)
      MemoryStream reqstream = new MemoryStream();
      SoapFormatter sf = new SoapFormatter();
      RemotingSurrogateSelector rss = new RemotingSurrogateSelector();
      rss.SetRootObject(msgReq);
      sf.SurrogateSelector = rss;
      sf.AssemblyFormat = FormatterAssemblyStyle.Full;
      sf.TypeFormat = FormatterTypeStyle.TypesAlways;
      sf.TopObject = new SoapMessage();
      sf.Serialize(reqstream, msgReq);
      ISoapMessage sm = sf.TopObject;
      reqstream.Position = 0;
      StreamReader sr = new StreamReader(reqstream);
      string request = sr.ReadToEnd();
      reqstream.Close();
      sr.Close();

      // call web service
      string respond = webservice.SyncProcessSoapMessage(request);

      // return messages
      StreamWriter rspsw = new StreamWriter(new MemoryStream());
      rspsw.Write(respond);
      rspsw.Flush();
      rspsw.BaseStream.Position = 0;
      ISoapMessage rspsoapmsg = (ISoapMessage)sf.Deserialize(rspsw.BaseStream);
      rspsw.Close();

      if(rspsoapmsg.ParamValues[0] is Exception)
      {
        throw rspsoapmsg.ParamValues[0] as Exception;
      }
      else
      {
        object returnVal = rspsoapmsg.ParamValues[4];
        object[] OutArgs = rspsoapmsg.ParamValues[5] as object[];
        LogicalCallContext lcc = rspsoapmsg.ParamValues[6] as LogicalCallContext;
        ReturnMessage rm = new ReturnMessage(
                returnVal,          //Object return
                OutArgs,            //Object[] outArgs
                OutArgs.Length,     //int outArgsCount
                lcc,                //LogicalCallContext callCtx
                msgReq as IMethodCallMessage    //IMethodCallMessage mcm
                );
        msgRsp = rm as IMessage;
      }
    }
```

```
    else
    {
      msgReq.Properties["__Uri2"] = m_ObjectUri; // workaround!
      // serialize and encode IMessage
      BinaryFormatter bf = new BinaryFormatter();
      MemoryStream reqstream = new MemoryStream();
      bf.Serialize(reqstream, msgReq);
      reqstream.Position = 0;
      string request = Convert.ToBase64String(reqstream.ToArray());
      reqstream.Close();
      // call Web Service
      string respond = webservice.SyncProcessMessage(request);

      // decode and deserialize IMessage
      byte[] rspbyteArray = Convert.FromBase64String(respond);
      MemoryStream rspstream = new MemoryStream();
      rspstream.Write(rspbyteArray, 0, rspbyteArray.Length);
      rspstream.Position = 0;
      msgRsp = (IMessage)bf.Deserialize(rspstream);
      rspstream.Close();
    }
  }
  catch(Exception ex)
  {
    Trace.WriteLine(string.Format("Client:SyncProcessMessage error = {0}", ex.Message));
    msgRsp = new ReturnMessage(ex, (IMethodCallMessage)msgReq);
  }

  return msgRsp;
}

public virtual IMessageCtrl AsyncProcessMessage(IMessage msgReq, IMessageSink replySink)
{
  IMessageCtrl imc = null;

  if(replySink == null) // OneWayAttribute
  {
    Trace.WriteLine("Client-[OneWay]Async:CALL");
    SyncProcessMessage(msgReq);
  }
  else
  {
    Trace.WriteLine("Client-Async:CALL");
    // spawn thread (delegate work)
    delegateAsyncWorker daw = new delegateAsyncWorker(handlerAsyncWorker);
```

```
      daw.BeginInvoke(msgReq, replySink, null, null);
   }

return imc;
}
```

The Web Service Client Proxy changes:

```
[System.Web.Services.WebServiceBindingAttribute(Name="ServiceSoap",
Namespace="http://tempuri.org/")]
public class Service : System.Web.Services.Protocols.SoapHttpClientProtocol {

   [System.Diagnostics.DebuggerStepThroughAttribute()]
      public Service(string uri) {
      this.Url = uri;
   }
   ...
}
```

16.10.7 WebServiceListener

This is a Web Service gateway to listen a MethodCall formatted into the string *request*. There are two different WebMethods for this process: SyncProcessMessage and SyncProcessSoapMessage. The functions have a simple logic divided into tree steps:

Decoding and de-serializing of the request message
Invoking the Remote Method
Encoding and serialization of the response message

```
[WebMethod]
public string SyncProcessMessage(string request)
{
   // Request: decoding and deserializing
   byte[] reqbyteArray = Convert.FromBase64String(request);
   MemoryStream reqstream = new MemoryStream();
   reqstream.Write(reqbyteArray, 0, reqbyteArray.Length);
   reqstream.Position = 0;
   BinaryFormatter bf = new BinaryFormatter();
   IMessage reqmsg = (IMessage)bf.Deserialize(reqstream);
   reqmsg.Properties["__Uri"] = reqmsg.Properties["__Uri2"]; // work around!!
   reqstream.Close();

   // Action: invoke the Remote Method
   string[] stype = reqmsg.Properties["__TypeName"].ToString().Split(new Char[]{','}); // split
typename
   Assembly asm = Assembly.Load(stype[1].TrimStart(new char[]{' '})); // load type of the remote
object
   Type objectType = asm.GetType(stype[0]);                    // type
   string objectUrl = reqmsg.Properties["__Uri"].ToString();        // endpoint
   object ro = RemotingServices.Connect(objectType, objectUrl);      // create proxy
   TraceIMessage(reqmsg);
```

```
  IMessage rspmsg = RemotingServices.ExecuteMessage((MarshalByRefObject)ro,
                                    (IMethodCallMessage)reqmsg);
  TraceIMessage(rspmsg);

  // Response: encoding and serializing
  MemoryStream rspstream = new MemoryStream();
  bf.Serialize(rspstream, rspmsg);
  rspstream.Position = 0;
  string response = Convert.ToBase64String(rspstream.ToArray());
  rspstream.Close();

  return response;
}

[WebMethod]
public string SyncProcessSoapMessage(string request)
{
  IMessage retMsg = null;
  string response;

  try
  {
    Trace.WriteLine(request);

    // Request: deserialize string into the SoapMessage
    SoapFormatter sf = new SoapFormatter();
    sf.TopObject = new SoapMessage();
    StreamWriter rspsw = new StreamWriter(new MemoryStream());
    rspsw.Write(request);
    rspsw.Flush();
    rspsw.BaseStream.Position = 0;
    ISoapMessage soapmsg = (ISoapMessage)sf.Deserialize(rspsw.BaseStream);
    rspsw.Close();

    // Action: invoke the Remote Method
    object[] values = soapmsg.ParamValues;
    string[] stype = values[2].ToString().Split(new Char[]{','});
    Assembly asm = Assembly.Load(stype[1].TrimStart(new char[]{' '}));
    Type objectType = asm.GetType(stype[0]);
    string objectUrl = values[0].ToString();
    RealProxyWrapper rpw = new RealProxyWrapper(objectType, objectUrl,
                                soapmsg.ParamValues[4]);
    object ro = rpw.GetTransparentProxy();
    MethodInfo mi = objectType.GetMethod(values[1].ToString());
    object retval = mi.Invoke(ro, values[3] as object[]);
    retMsg = rpw.ReturnMessage;
  }
  catch(Exception ex)
  {
    retMsg = new ReturnMessage((ex.InnerException == null) ?
                               ex : ex.InnerException, null);
  }
  finally
  {
    // Response: serialize IMessage into string
    Stream rspstream = new MemoryStream();
    SoapFormatter sf = new SoapFormatter();
    RemotingSurrogateSelector rss = new RemotingSurrogateSelector();
    rss.SetRootObject(retMsg);
    sf.SurrogateSelector = rss;
    sf.AssemblyFormat = FormatterAssemblyStyle.Full;
    sf.TypeFormat = FormatterTypeStyle.TypesAlways;
    sf.TopObject = new SoapMessage();
    sf.Serialize(rspstream, retMsg);
    rspstream.Position = 0;
    StreamReader sr = new StreamReader(rspstream);
    response = sr.ReadToEnd();
    rspstream.Close();
    sr.Close();
  }

  Trace.WriteLine(response);
  return response;
}
```

The implementation of the steps are depended from the type of formatter such as SoapFormatter or BinaryFormatter. The first and last steps are straightforward using the Remoting namespace classes. The second one (action) for the SoapFormatter message needed to create the following class to obtain IMessage of the MethodCall:

```
public class RealProxyWrapper : RealProxy
{
  string _url;
  string _objectURI;
  IMessageSink _messageSink;
  IMessage _msgRsp;
  LogicalCallContext _lcc;

  public IMessage ReturnMessage { get { return _msgRsp; }}
  public RealProxyWrapper(Type type, string url, object lcc) : base(type)
  {
    _url = url;
    _lcc = lcc as LogicalCallContext;

    foreach(IChannel channel in ChannelServices.RegisteredChannels)
    {
      if(channel is IChannelSender)
      {
        IChannelSender channelSender = (IChannelSender)channel;
        _messageSink = channelSender.CreateMessageSink(_url, null, out _objectURI);
        if(_messageSink != null)
          break;
      }
    }

    if(_messageSink == null)
    {
      throw new Exception("A supported channel could not be found for url:"+ _url);
    }
  }
  public override IMessage Invoke(IMessage msg)
  {
    msg.Properties["__Uri"] = _url; // endpoint
    msg.Properties["__CallContext"] = _lcc; // caller's callcontext
    _msgRsp = _messageSink.SyncProcessMessage(msg);

    return _msgRsp;
  }
}// RealProxyWrapper
```

16.10.8 Test Functionality

We built the following package to test functionality of the WebServiceListener and WebServiceChannelLib assemblies.
ConsoleClient, the test console program to invoke the call over Internet - client machine
ConsoleServer, the host process of the MyRemoteObject - server machine
MyRemoteObject, the remote object - server machine
WebServiceChannelLib, the custom client channel
WebServiceListener, the Web Service listener - server machine
To recompile a package and its deploying in your environment follow these notes:
The folder WebServiceListener has to be moved to the virtual directory (inetpub\wwwroot).
The MyRemoteObject assembly has to be install into the GAC on the server machine
The WebServiceChannelLib assembly has to be install into the GAC on the client machine
(option) The MSMQChannelLib assembly [1] has to be install into the GAC on the server machine
The solution can be tested also using the same machine (Win2k/Adv Server)

Use the Echo WebMethod on the test page of the WebServiceListener to be sure that this service will be work The test case is very simple. First step is to launch the *ConsoleServer* program.

Secondly open the *ConsoleClient* program and follow its prompt text. If everything is going right you will see a response from the remote object over Internet

In this program has been shown one simple way how to implement a solution for remoting over Internet. We have used the power of .NET Technologies such as SOAP, Remoting, Reflection and Web Service. The advantage of this solution is a full transparency between the consumer and remote object. This logical connectivity can be mapped into the physical path using the config files, which they can be administratively changed.

16.11 Networking

The .NET framework provides two namespaces, System.NET and System.NET.Sockets for network programming. The classes and methods of these namespaces help us to write programs, which can communicate across the network. The communication can be either connection oriented or connectionless. They can also be either stream oriented or data-gram based. The most widely used protocol TCP is used for stream-based communication and UDP is used for data-grams based applications. The System.NET.Sockets.Socket is an important class from the System.NET.Sockets namespace. A Socket instance has a local and a remote end-point associated with it. The local end-point contains the connection information for the current socket instance.

There are some other helper classes like IPEndPoint, IPADdress, SocketException etc, which we can use for Network programming. The .NET framework supports both synchronous and asynchronous communication between the client and server. There are different methods supporting for these two types of communication. A synchronous method is operating in blocking mode, in which the method waits until the operation is complete before it returns. But an asynchronous method is operating in non-blocking mode, where it returns immediately, possibly before the operation has completed.

16.11.1 DNS Class

The System.NET namespace provides this class, which can be used to creates and send queries to obtain information about the host server from the **Internet Domain Name Service (DNS)**. Remember that in order to access DNS, the machine executing the query must be connected to a network. If the query is executed on a machine, that does not have access to a domain name server, a System.NET.SocketException is thrown. All the members of this class are static in nature. The important methods of this class are given below.

public static IPHostEntry GetHostByAddress(string address)

Where address should be in a dotted-quad format like "202.87.40.193". This method returns an IPHostEntry instance containing the host information. If DNS server is not available, the method returns a SocketException.

public static string GetHostName()

This method returns the DNS host name of the local machine.

In my machine Dns.GetHostName() returns vrajesh which is the DNS name of my machine.

public static IPHostEntry Resolve(string hostname)

This method resolves a DNS host name or IP address to a IPHostEntry instance. The host name should be in a dotted-quad format like 127.0.01 or www.msn.com.

16.11.2 IPHostEntryclass

This is a container class for Internet host address information. This class makes no thread safety guarantees. The following are the important members of this class.

AddressList property

Gives an IPAddress array containing IP addresses that resolve to the host name.

Aliases property

Gives a string array containing DNS name that resolves to the IP addresses in AddressList property.

The following program shows the application of the above two classes.

```
using System;
using System.NET;
using System.NET.Sockets;

class MyClient
{
        public static void Main()
        {
        IPHostEntry IPHost = Dns.Resolve("www.hotmail.com");
        Console.WriteLine(IPHost.HostName);
        string []aliases = IPHost.Aliases;
        Console.WriteLine(aliases.Length);

IPAddress[] addr = IPHost.AddressList;
        Console.WriteLine(addr.Length);
        for(int i= 0; i < addr.Length ; i++)
        {
        Console.WriteLine(addr[i]);
        }
        }


}
```

16.11.3 IPEndPoint Class

This class is a concrete derived class of the abstract class EndPoint. The IPEndPoint class represents a network end point as an IP address and a port number. There is couple of useful constructors in this class:

```
IPEndPoint(long addresses, int port)
IPEndPoint (IPAddress addr, int port)
IPHostEntry IPHost = Dns.Resolve("www.C#corner.com");
Console.WriteLine(IPHost.HostName);
string []aliases = IPHost.Aliases;
IPAddress[] addr = IPHost.AddressList;
```

```
Console.WriteLine(addr[0]);
EndPoint ep = new IPEndPoint(addr[0],80);
```

We already discussed about the basics of network programming with C#. We will discuss about the System.NET.Socket class and how we can use Socket class to write network programming in the coming articles.

Socket Programming: Synchronous Clients

The steps for creating a simple synchronous client are as follows.

1. Create a Socket instance.
2. Connect the above socket instance to an end-point.
3. Send or Receive information.
4. Shutdown the socket
5. Close the socket

The Socket class provides a constructor for creating a Socket instance.

public Socket (AddressFamily af, ProtocolType pt, SocketType st)

Where AddressFamily, ProtocolType and SocketTYpe are the enumeration types declared inside the Socket class.The AddressFamily member specifies the addressing scheme that a socket instance must use to resolve an address. For example AddressFamily.InterNetwork indicates that an IP version 4 addresses is expected when a socket connects to an end point. The SocketType parameter specifies the socket type of the current instance. For example SocketType.Stream indicates a connection-oriented stream and SocketType.Dgram indicates a connectionless stream.The ProtocolType parameter specifies the ptotocol to be used for the communication. For example ProtocolType.Tcp indicates that the protocol used is TCP and ProtocolType.Udp indicates that the protocol using is UDP.

public Connect (EndPoint ep)

The Connect() method is used by the local end-point to connect to the remote end-point. This method is used only in the client side. Once the connection has been established the Send() and Receive() methods can be used for sending and receiving the data across the network. The Connected property defined inside the class Socket can be used for checking the connection. We can use the Connected property of the Socket class to know whether the current Socket instance is connected or not. A property value of true indicates that the current Socket instance is connected.

```
IPHostEntry IPHost = Dns.Resolve("www.microsoft.com");
Console.WriteLine(IPHost.HostName);
string []aliases = IPHost.Aliases;
IPAddress[] addr = IPHost.AddressList;
Console.WriteLine(addr[0]);
EndPoint ep = new IPEndPoint(addr[0],80);

Socket sock = new      Socket(AddressFamily.InterNetwork,SocketType.Stream,ProtocolType.Tcp);
sock.Connect(ep);
if(sock.Connected)
Console.WriteLine("OK");
```

The Send() method of the socket class can be used to send data to a connected remote socket.

public int Send (byte[] buffer, int size, SocketFlags flags)

Where byte[] parameter storing the data to send to the socket, size parameter containing the number of bytes to send across the network. The SocketFlags parameter can be a bitwise combination of any one of the following values defined in the System.NET.Sockets.SocketFlags enumerator.

SocketFlags.None
SocketFlags.DontRoute
SocketFlags.OutOfBnd

The method Send() returns a System.Int32 containing the number of bytes send.Remember that there are other overloaded versions of Send() method as follows.

public int Send (byte[] buffer, SocketFlags flags)
public int Send (byte[] buffer)
public int Send (byte[] buffer,int offset, int size, SocketFlags flags)

The Receive() method can be used to receive data from a socket.

public int Receive(byte[] buffer, int size, SocketFlags flags)

Where byte[] parameter storing the data to send to the socket, size parameter containing the number of bytes to send across the network. The SocketFlags parameter can be a bitwise combination of any one of the following values defined in the System.NET.Sockets.SocketFlags enumerator explained above.

The overloaded versions of Receive() methods are shown below.

public int Receive (byte[] buffer, SocketFlags flags)
public int Receive (byte[] buffer)
public int Receive (byte[] buffer,int offset, int size, SocketFlags flags)

When the communication across the sockets is over, the connection between the sockets can be terminated by invoking the method ShutDown()

public void ShutDown(SocketShutdown how)

Where 'how' is one of the values defined in the SocketSHutdown enumeration. The value SoketShutdown.Send means that the socket on the other end of the connection is notified that the current instance would not send any more data. The value SoketShutdown.Receive means that the socket on the other end of the connection is notified that the current instance will not receive any more data and the value SoketShutdown.Both means that both the action are not possible.

Remember that the ShutDown() method must be called before the Close(0 method to ensure that all pending data is sent or received.

A socket can be closed by invoking the method Close().

public void Close()

This method closes the current instance and releases all managed and un-managed resources allocated by the current instance. This method internally calls the Dispose() method with an argument of 'true' value, which frees both managed and un-managed resources used by the current instance.

protected virtual void Dispose(bool)

The above method closes the current instance and releases the un-managed resources allocated by the current instance and exceptionally release the managed resources also. An argument value of 'true' releases both managed and un-managed resources and a value of 'false' releases only un-managed resources.

The source code for a simple synchronous client by using the sockets is show below. The following program can send an HTTP request to a web server and can read the response from the web server.

```
using System;
using System.NET;
using System.NET.Sockets;
using System.Text;
class MyClient
{
public static void Main()
{
IPHostEntry IPHost = Dns.Resolve("www.google.com");
Console.WriteLine(IPHost.HostName);
string []aliases = IPHost.Aliases;
IPAddress[] addr = IPHost.AddressList;
Console.WriteLine(addr[0]);
EndPoint ep = new IPEndPoint(addr[0],80);
Socket sock = new Socket(AddressFamily.InterNetwork,SocketType.Stream,ProtocolType.Tcp);
sock.Connect(ep);
if(sock.Connected)
        Console.WriteLine("OK");
Encoding ASCII = Encoding.ASCII;
string Get = "GET / HTTP/1.1\r\nHost: " + "www. google.com" +
        "\r\nConnection: Close\r\n\r\n";
Byte[] ByteGet = ASCII.GetBytes(Get);
Byte[] RecvBytes = new Byte[256];
sock.Send(ByteGet, ByteGet.Length, 0);
Int32 bytes = sock.Receive(RecvBytes, RecvBytes.Length, 0);
Console.WriteLine(bytes);
String strRetPage = null;
strRetPage = strRetPage + ASCII.GetString(RecvBytes, 0, bytes);
while (bytes > 0)
{
   bytes = sock.Receive(RecvBytes, RecvBytes.Length, 0);
   strRetPage = strRetPage + ASCII.GetString(RecvBytes, 0, bytes);
   Console.WriteLine(strRetPage );
}
sock.ShutDown(SocketShutdown.Both);
sock.Close();
}
}
```

16.12 WINCV Tool – Windows Forms Class Viewer

The Windows Forms Class Viewer allows you to quickly look up information about a class or series of classes, based on a search pattern. The class viewer displays information by reflecting on the type using the common language runtime reflection API.

`wincv` `[`*`options`*`]`

Option	Description
/h	Displays command syntax and options for the tool.
/hide: *type*	Hides the specified member type. You must specify protected, private, internal, or inherited as the *type* argument. If you do not specify this option, protected, private, and internal types are hidden by default. To specify multiple member types to hide, specify the /hide option multiple times on the command line separated by a space. For example, /hide: protected /hide: private.
/nostdlib[+|-]	Specifies whether to load the following default assemblies: mscorlib.dll, System.dll, System.Data.dll, System.Design.dll, System.DirectoryServices.dll, System.Drawing.dll, System.Drawing.Design.dll, System.Messaging.dll, System.Runtime.Serialization.Formatters.Soap.dll, System.ServiceProcess.dll, System.Web.dll, System.Web.Services.dll, System.Windows.Forms.dll, System.XML.dll, If you specify the plus symbol (+), Wincv.exe does not load the default assemblies. The default is /nostdlib-, which loads the default assemblies.
/r:*assemblyFile*	Specifies an assembly to load and browse.
/show:*type*	Displays the specified member type. You must specify protected, private, internal, or inherited as the *type* argument. If you do not specify this option, only inherited types are displayed by default. . To specify multiple member types to display, specify the /show option multiple times on the command line separated by a space. For example, /show:protected /show:private.
@*fileName*	Reads the specified response file for more options.
/?	Displays command syntax and options for the tool.

16.12.1 Date and Time

Represents an instant in time, typically expressed as a date and time of day.
For a list of all members of this type, see DateTime Members.

```
[C++]
[Serializable]
public __value struct DateTime : public IComparable, IFormattable,
   IConvertible
```

[JScript] In JScript, you can use the structures in the .NET Framework, but you cannot define your own.

16.12.2 Thread Safety

Any public static (Shared in Visual Basic) members of this type are safe for multithreaded operations. Any instance members are not guaranteed to be thread safe.

Remarks

The **DateTime** value type represents dates and times with values ranging from 12:00:00 midnight, January 1, 0001 C.E. (Common Era) to 12:59:59 P.M., December 31, 9999 C.E.

Time values are measured in 100-nanosecond units called ticks, and a particular date is the number of ticks since 12:00 midnight, January 1, 1 C.E. in the GregorianCalendar calendar. For example, a ticks value of 31241376000000000L represents the date, Friday, January 01, 0100 12:00:00 midnight. A **DateTime** value is always expressed in the context of an explicit or default calendar.

The **DateTime** and TimeSpan value types differ in that a **DateTime** represents an instant in time, whereas a **TimeSpan** represents a time interval. This means, for example, that you can subtract one instance of **DateTime** from another to obtain the time interval between them. Or you could add a positive **TimeSpan** to the current **DateTime** to calculate a future date.

Time values can be added to, or subtracted from, an instance of **DateTime**. Time values can be negative or positive, and expressed in units such as ticks, seconds, or instances of **TimeSpan**. Methods and properties in this value type take into account details such as leap years and the number of days in a month.

Descriptions of time values in this type are often expressed using the coordinated universal time (UTC) standard, which was previously known as Greenwich mean time (GMT).

Calculations and comparisons of **DateTime** instances are only meaningful when the instances are created in the same time zone. For that reason, it is assumed that the developer has some external mechanism, such as an explicit variable or policy, to know in which time zone a **DateTime** was created. Methods and properties in this class always use the local time zone when making calculations or comparisons.

A calculation on an instance of **DateTime**, such as Add or Subtract, does not modify the value of the instance. Instead, the calculation returns a new instance of **DateTime** whose value is the result of the calculation.

This type inherits from IComparable, IFormattable, and IConvertible. Use the Convert class for conversions instead of this type's explicit interface member implementation of **IConvertible**.

Requirements

Namespace**:** System

Platforms**:** Windows 98, Windows NT 4.0, Windows Millennium Edition, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows .NET Server family

Assembly**:** Mscorlib (in Mscorlib.dll)

16.13 Mathematical functions

Math Class [C#]

Namespace: System

Platforms: Windows 98, Windows NT 4.0, Windows Millennium Edition, Windows 2000, Windows XP Home Edition, Windows XP Professional, Windows .NET Server family

Assembly: Mscorlib (in Mscorlib.dll)

Provides constants and static methods for trigonometric, logarithmic, and other common mathematical functions.

System.Object, System.Math, public sealed class Math, Thread Safety

Any public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Any instance members are not guaranteed to be thread safe.

16.13.1 Math Members

Public Fields

E	Represents the natural logarithmic base, specified by the constant, **e**.
PI	Represents the ratio of the circumference of a circle to its diameter, specified by the constant, π.

16.13.2 Public Methods

Abs	Overloaded. Returns the absolute value of a specified number.
Acos	Returns the angle whose cosine is the specified number.

Asin	Returns the angle whose sine is the specified number.
Atan	Returns the angle whose tangent is the specified number.
Atan2	Returns the angle whose tangent is the quotient of two specified numbers.
Ceiling	Returns the smallest whole number greater than or equal to the specified number.
Cos	Returns the cosine of the specified angle.
Cosh	Returns the hyperbolic cosine of the specified angle.
Exp	Returns **e** raised to the specified power.
Floor	Returns the largest whole number less than or equal to the specified number.
IEEERemainder	Returns the remainder resulting from the division of a specified number by another specified number.
Log	Overloaded. Returns the logarithm of a specified number.
Log10	Returns the base 10 logarithm of a specified number.
Max	Overloaded. Returns the larger of two specified numbers.
Min	Overloaded. Returns the smaller of two numbers.
Pow	Returns a specified number raised to the specified power.
Round	Overloaded. Returns the number nearest the specified value.
Sign	Overloaded. Returns a value indicating the sign of a number.
Sin	Returns the sine of the specified angle.
Sinh	Returns the hyperbolic sine of the specified angle.
Sqrt	Returns the square root of a specified number.
Tan	Returns the tangent of the specified angle.
Tanh	Returns the hyperbolic tangent of the specified angle.

Review Questions.

1. What is a data set Explain?

2.Differentiate Typed dataset and Untyped dataset?

3.How will you populate a dataset?

4.Explain the functions of Assemblies in .Net Framework?

5.What are the uses of reflection?

6.Explain COM Interoperability?

7.How will you call a .net component from COM?

8.What is Marshalling. Expalin its use with respect to Custom Interfaces?

9.Explain .Net remoting?

10. Explain ASP.NET web application with help of a diagram?