CS T54 PLATFORM TECHNOLOGY
UNIT 1

1.Write about the .NET framework?

.NET FRAMEWORK

.NET Framework is a programming infrastructure created by Microsoft for building, deploying, and running applications and services that use .NET technologies, such as desktop applications and Web services. The .NET Framework is a new computing platform that simplifies application development in the highly distributed environment of the Internet Services.

.NET FRAMEWORK SERVICES

.NET Framework provides the following services namely,
- ✓ Tools for developing software applications
- ✓ Run-time environments for software application to execute
- ✓ Server infrastructure
- ✓ Value added intelligent software which helps developers to do less coding and work efficiently.

.NET FRAMEWORK OBJECTIVES

The .Net Framework will enable developers to develop applications for various devices and platforms like windows application, web applications windows services and web services.
The .NET Framework is designed to fulfill the following objectives:
- A consistent object-oriented programming environment, where object code can be stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- A code-execution environment that minimizes software deployment and versioning conflicts.
- A code-execution environment that guarantees safe execution of code, including code created by an unknown or semi-trusted third party.
- A code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- Developers can experience consistency across widely varying types of applications, such as Windows-based applications and Web-based applications.
- Build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code.

.NET FRAMEWORK ARCHITECTURE

The .NET Framework has two components: the .NET Framework class library and the common language runtime.

The .NET Framework class library facilitates types (CTS) that are common to all .NET languages.The common language runtime consists of (class loader) that load the IL code of a program into the runtime, which compiles the IL code into native code, and executes and manage the code to enforce security and type safety, and provide thread support.
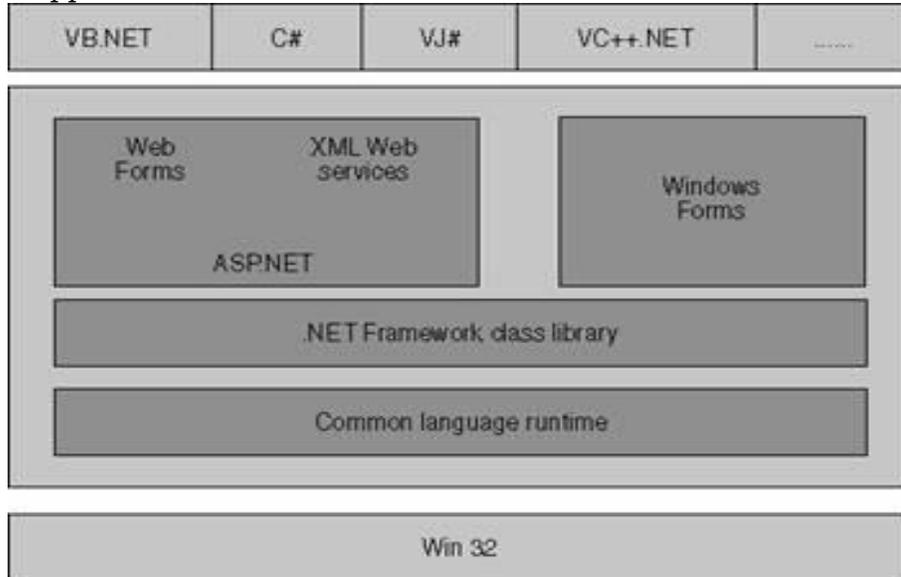

Fig 1.1 .NET framework

.NET Framework Architecture has languages at the top such as VB .NET C#, VJ#, VC++ .NET; developers can develop (using any of above languages) applications such as Windows Forms, Web Form, Windows Services and XML Web Services. Bottom two layers consist of .NET Framework class library and Common Language Runtime.

Common Language Runtime

- The .NET Framework provides a run-time environment called the common language runtime, which runs the code and provides services that make the development process easier.
- The common language runtime makes it easy to design components and applications whose objects interact across languages.

.net framework class library

The .NET framework class library, as the name suggests, is a library of classes, interfaces and value types. The applications, components and controls for applications are built on this framework and it provides the developer the access to the system functionality.

2.Explain briefly about the CLR

COMMON LANGUAGE RUNTIME (CLR)

- The .NET Framework provides a run-time environment called the common language runtime, which runs the code and provides services that make the development process easier.
- The common language runtime makes it easy to design components and applications whose objects interact across languages.
- Objects written in different languages can communicate with each other, and their behaviors can be tightly integrated. For example, you can define a class and then use a different language to derive a class from the original class or call a method on the original class.
- We can also pass an instance of a class to a method of a class written in a different language.

BENEFITS
The runtime provides the following benefits:
- Performance improvements.
- The ability to easily use components developed in other languages.
- Extensible types provided by a class library.
- Language features such as inheritance, interfaces, and overloading for object-oriented programming.
- Garbage collection.
- Use of delegates instead of function pointers for increased type safety and security.
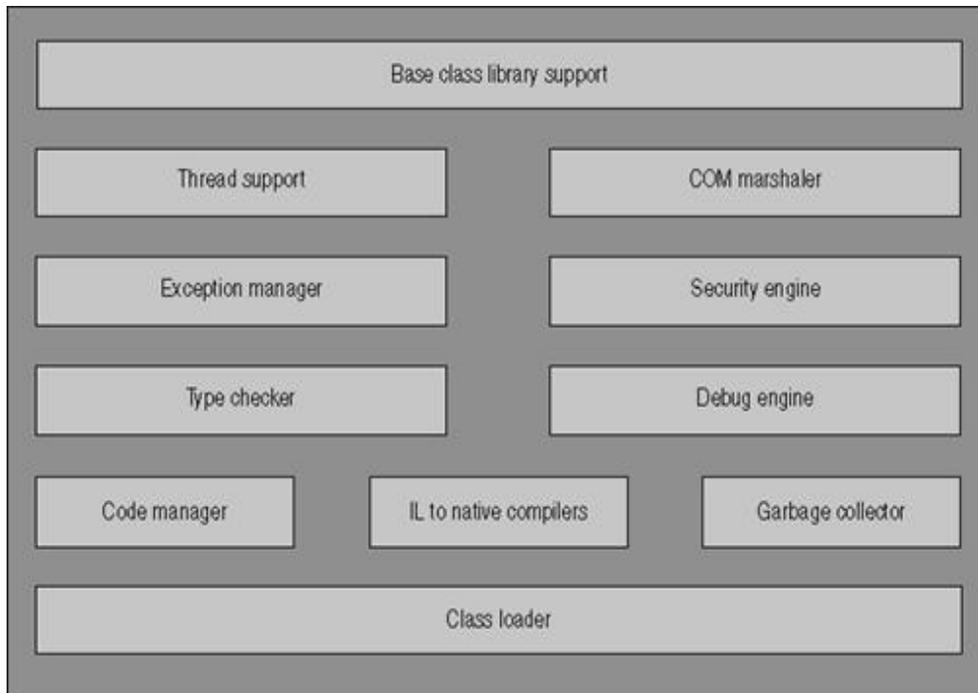
Fig 1.2 Common Language Runtime (CLR)

- *Class loader*: Loads classes into CLR.
- *MSIL to native code compile:* Converts MSIL code into native code.
- *Code manager*: Manages the code during execution.
- *Memory allocation and Garbage collector*: Performs automatic memory management.
- *Security engine*: Enforces security restrictions as code level security folder level and machine level security using tools provided by Microsoft .NET and using .NET Framework setting under control panel.
- *Type checker*: Enforces strict type checking.
- *Thread support*: Provides multithreading support to applications.
- *Exception manager*: Provides a mechanism to handle the run-time exceptions handling.
- *Debug engine*: Allows developer to debug different types of applications.
- *COM marshaler*: Allows .NET applications to exchange data with COM applications.
- *Base class library support*: which provides the classes (types) that the applications need at run time.

The language compiler compiles the source code into the MSIL code which consists of CPU- independent code and instructions which is platform independent. MSIL consists of the followings:

- Instructions performs arithmetic and logical operations.

- Access memory directly.
- Control the flow of execution.
- Handles exceptions.

MSIL code can be compiling into CPU specific instructions before executing, for which the CLR requires information about the code which is nothing but metadata. Metadata describes the code and defines the types that the code contains as well referenced to other types which the code uses at run time.

An assembly consists of portable executable file. At the time of executing PE file the class loader loads the MSIL code and the metadata form the portable executable file into the run time memory.CLR forms the heart of the .Net framework.

3. Explain briefly about the Framework Class Library (FCL)
(OR)
Write in detail about the namespaces in .NET framework?

.NET FRAMEWORK CLASS LIBRARY

- The .NET framework class library, as the name suggests, is a library of classes, interfaces and value types. The applications, components and controls for applications are built on this framework and it provides the developer the access to the system functionality
- The .NET Framework class Library is organized into namespaces. The namespace is a container for functionality. Similar classes and constructs are grouped together in a namespace to define parent-child relationships. Namespaces can be nested into namespaces.
- All namespaces stem from the root namespace called System Namespace.
- It contains all data types including the Object data type.
- Though all namespaces are subordinated to the System namespace, User defined libraries can also coexist with the System namespace. They can have their own root namespace which can be language focused namespaces such as Microsoft.Csharp, Microsoft.VisualBasic.
- The most significant feature of the .NET framework is the class Library collection of reusable types can be integrated with CLR.
- The programmer can accomplish a range of common programming tasks, such as string management; data collection; data base connectivity and file access using the .NET framework class library.
- The developer can create console applications, Windows GUI applications, ASP.NET applications, XML Web services or Windows services.
- The .NET Framework class library is a library of classes, interfaces, and value types that are included in the Microsoft .NET Framework SDK. This library provides access to system functionality and is designed to be the

foundation on which .NET Framework applications, components, and controls are built.

NAMESPACES IN .NET FRAMEWORK

The .NET Framework class library provides the following namespaces:

❖ **System**

The root of the tree, this namespace contains all of the other namespaces in the .Net Framework class library. System also contains the core data types used by the CLR. These types include several varieties of integers, a string type, and many more.

❖ **System.Windows.Forms**

The types in this namespace make up Windows Forms, and they are used to build Windows GUIs. Rather than relying on language-specific mechanisms, such as the older Microsoft Foundation Classes (MFC) in C++, .Net Framework applications written in any programming language use this common set of types to build graphical interfaces for Windows.

❖ **System.Collections**

Contains interfaces and classes that define various collections of objects, such as lists, queues, bit arrays, hashtables and dictionaries.

❖ **System.ComponentModel**

Provides classes that are used to implement the run-time and design-time behavior of components and controls. This namespace includes the base classes and interfaces for implementing attributes and type converters, binding to data sources, and licensing components.

❖ **System.Configuration**

Provides classes and interfaces that allow you to programmatically access .NET Framework configuration settings and handle errors in configuration files (.config files).

❖ **System.Configuration.Assemblies**

Contains classes that are used to configure an assembly.

❖ **System.Configuration.Install**

Provides classes that allow you to write custom installers for your own components. The Installer class is the base class for all custom installers in the .NET Framework.
USAGE

The class library reference documentation can be filtered by language, so that you can view syntax, descriptions, and examples for one language (either Visual Basic, C#, the Managed Extensions for C++, or JScript) or all four languages at once. To filter by language, click the filtering icon at the top of any reference page and select a language or choose Show All.

EXCEPTIONS

All instance methods in the class library throw an instance of NullReferenceException when an attempt is made to call the method and the underlying object holds a null reference.

4. Write short notes on windows forms?

NET WINDOWS FORMS

WINDOWS FORMS
- Windows Forms is a smart client technology for the .NET Framework, a set of managed libraries that simplify common application tasks such as reading and writing to the file system.
- When you use a development environment like Visual Studio, you can create Windows Forms smart-client applications that display information, request input from users, and communicate with remote computers over a network.
- In Windows Forms, a *form* is a visual surface on which you display information to the user.
- You ordinarily build Windows Forms applications by adding controls to forms and developing responses to user actions, such as mouse clicks or key presses. A *control* is a discrete user interface (UI) element that displays data or accepts data input.
- When a user does something to your form or one of its controls, the action generates an event. Your application reacts to these events by using code, and processes the events when they occur.
- Windows Forms contains a variety of controls that you can add to forms: controls that display text boxes, buttons, drop-down boxes, radio buttons, and even Web pages.

BUILDING APPLICATION WITH WINDOWS FORMS

- In window forms, every form is an instance of the Form class, while the message loop that accepts and distributes events is provided by a class called Application.
- Using these and other classes in System.Windows.Forms, a developer can create a single-document interface (SDI) application, able to display only one document at a time, or a multiple-document interface (MDI) application, able to display more than one document simultaneously.
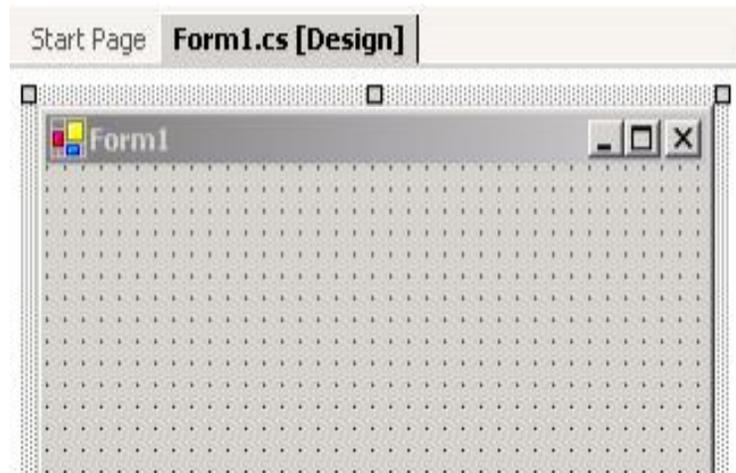
WINDOWS FORMS PROPERTIES

Each instance of the form class has a large set of properties that control how that form looks on the screen. Among them are Text, which indicates what caption should be displayed in the title bar; Size, which controls the form's initial on-screen size; DesktopLocation, which determines where on the screen the form appears; and many more. Developers set these properties to customize a form's appearance and behavior.
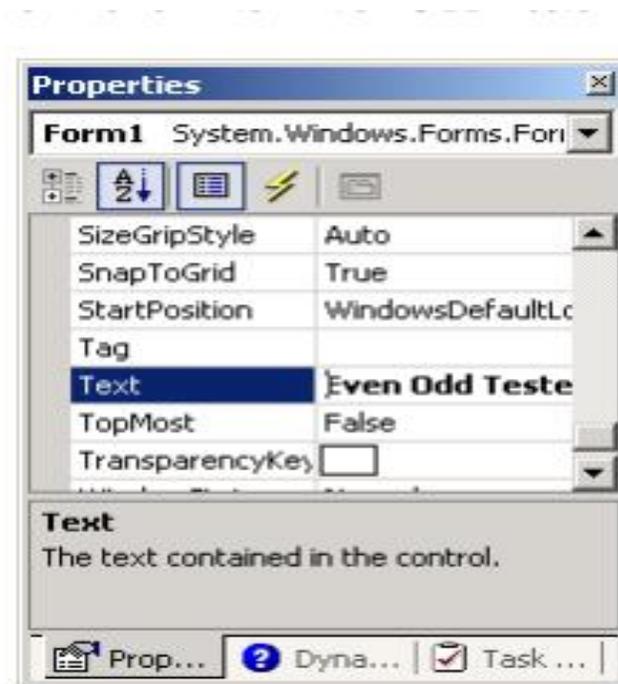
WINDOWS FORMS CONTROLS

- Forms commonly contain other classes called Windows Forms Controls.

- Each of these controls typically displays some kind of output, accepts some input from the user, or both.
- The System.Windows.Forms namespace provides a large set of controls, many of which will be familiar to anyone who's built or even used a GUI.
- The control classes available in this namespace include Button, TextBox, CheckBox, RadioButton, ListBox, ComboBox, and many more.
- Also provided are more complex controls such as OpenFileDialog, which encapsulates the operations that let a user open a file; SaveFileDialog, which encapsulates the operations that let a user save a file; PrintDialog, which encapsulates the operations that let a user print a document; and several others.

Control	Control
A Label	ComboBox
A LinkLabel	ListView
ab Button	TreeView
abl TextBox	TabControl
MainMenu	DateTimePicker
CheckBox	MonthCalendar
RadioButton	HScrollBar
GroupBox	VScrollBar
PictureBox	Timer
Panel	Splitter
DataGrid	DomainUpDown
ListBox	NumericUpDown
CheckedListBox	ProgressBar

WINDOWS FORMS CONTROLS PROPERTIES

- Like a form, each control has properties that can be set to customize its appearance and behavior. Many of these properties are inherited from System.Window.Forms.Control, the base class for every control.
- The Button control, for example, has a location property that determines where the button will appear relative to its container and a size property that determines how big the on-screen button will be, both of which are directly inherited from the parent Control class.

EXAMPLE: CONSIDER BUTTON CONTROL

EVENTS

- Forms and controls also support events.
- Some examples of common events include
    - ✓ Click, indicating that a mouse click has occurred;
    - ✓ GotFocus, indicating that the form or control has been selected by the user
    - ✓ KeyPress, indicating that a key has been pressed.
- A developer can create code to handle events received by a form or control. Called an event handler, this code determines what happens when the event occurs.

5. Uses Of Web Forms & Web Services

USES OF WEB FORMS & WEB SERVICES

- An important part of Visual Studio .NET is the ability to create distributed applications based around the Web. Visual Studio .NET allows you to create the application user interface using Web Forms pages and to create components using XML Web services.
- Web Forms is the ASP.NET technology that allows you to create a user interface for Web-based applications, whether customers are accessing your application from a traditional Web browser or a mobile device. Using Web Forms pages, you can create a browser-neutral UI that does its processing on the Web server, freeing you of the need to create browser-specific (or device-specific) versions of your user interface.
- XML Web services are components that run on the server and typically include business logic. Like traditional components, they encapsulate specific functionality and can be called from different programs. However, they are available via Web protocols, making them compatible with programs running in different languages, on different computers, and even on different operating systems.

Interoperability has Highest Priority

When all major platforms could access the Web using Web browsers, different platforms could interact. For these platforms to work together, Web-applications were developed. Web-applications are simple applications that run on the web. These are built around the Web browser standards and can be used by any browser on any platform.

Web Services take Web-applications to the Next Level

By using Web services, your application can publish its function or message to the rest of the world.Web services use XML to code and to decode data, and SOAP to transport it (using open protocols).

Reusable application-components

There are things applications need very often.Web services can offer application-components like: currency conversion, weather reports, or even language translation as services.

Connect existing software

Web services can help to solve the interoperability problem by giving different applications a way to link their data. With Web services you can exchange data between different applications and different platforms.

6. Explain about Common Type System (CTS) and Common Language Specification (CLS)

CTS and CLS are parts of .NET CLR and are responsible for type safety with in the code. Both allow cross language communication and type safety.

COMMON TYPE SYSTEM

CTS stands for Common Type System. It defines the rules which Common Language Runtime follows when declaring, using, and managing types. The common type system performs the following functions:

- ✓ It enables cross-language integration, type safety, and high-performance code execution.
- ✓ It provides an object-oriented model for implementation of many programming languages.
- ✓ It defines rules that every language must follow which runs under .NET framework. It ensures that objects written in different .NET Languages like C#, VB.NET, F# etc. can interact with each other.
- ✓

For example, C# has int Data Type and VB.Net has Integer Data Type. Hence a variable declared as int in C# or Integer in vb.net, finally after compilation, use the same structure Int32 from CTS.

VALUE TYPE AND REFERENCE TYPE

A Substantial subset of the types defined by the CTS is shown in the figure below.
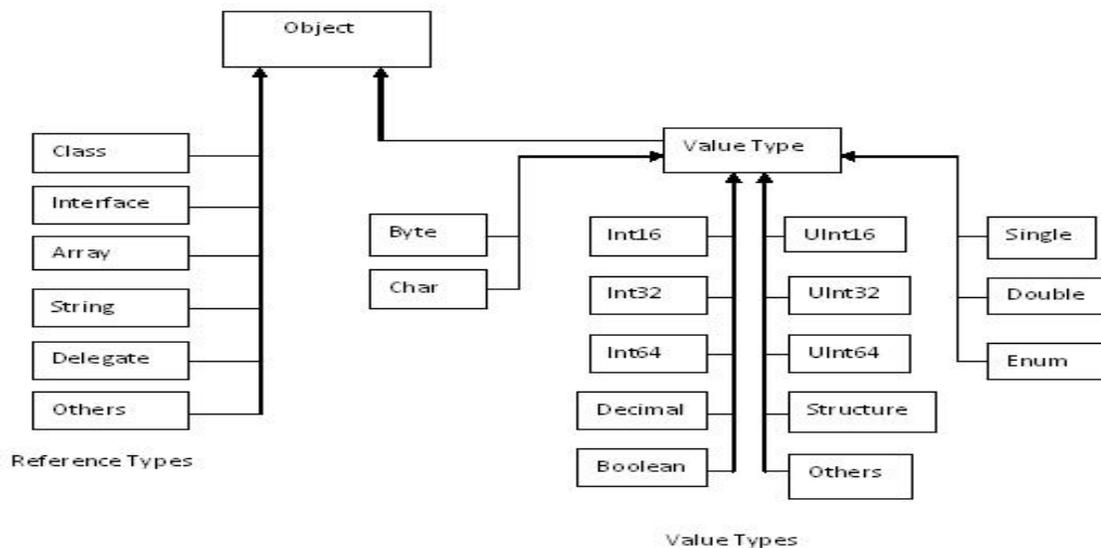

Fig 1.4 Value type and Reference type

1) The first thing to note is that every type inherits either directly or indirectly from a type called Object.

2)The second thing to note is that every type defined by the CTS is either a reference type or a value type.

3)As the name suggest, an instance of a reference type always contains the value itself. Reference types inherit directly from Object, while all value types inherit directly from a type called ValueType, which in turn inherits from Object.

Value Types

All value types inherit from ValueType. Like Object, ValueType provides an Equals method. Value types cannot act as a parent type for inheritance, however, so it's not possible to say, define a new type that inherits from Int32.

Many of the value types are defined by the CTS. Those types are as follows:

- **Byte:** An 8-bit unsigned integer
- **Char:** A 16 – bit Unicode character.
- **Int16, Int32, and Int64:** 16-, 32-, and 64- bit unsigned Integers.
- **Single and Double:** Single-precision (32-bit) and double-precision (64-bit) floating-point numbers.
- **Decimal:** 96-bit decimal numbers.
- **Enum:** A way to name a group of values of some integer type. Enumerated types inherit from System.Enum and are used to define types whose values have meaningful names rather than just numbers.
- **Boolean:** True or False

Reference Types

Compared with most value types, the reference types defined by the CTS are relatively complicated. Before describing some of the more important reference types, it's useful to look first at a few elements, officially known as type members that are common to several types. Those elements are as follows:

- **Methods**: Executable code that carries out some kind of operation. Methods can be overloaded, which means that a single type can define two or more methods with the same name. To distinguish among them, each of these identical named methods must differ somehow in its parameter list. Another way to say this is to state that each method must have a unique signature. If a method encounters an error, it can throw an exception, which provides some indication of what has gone wrong, provides some indication of what has gone wrong.
- **Fields:** A value of some type.
- **Events:** A mechanism for communicating with other types. Each event includes methods for subscribing and unsubscribing and for sending the event to subscribers.
- **Properties:** In effect, a value together with specified methods to read and/or write that value.
- **Nested Types:** A type is defined inside another type. A common example of this is defining a class that is nested inside another class.

- **Class:** A CTS class can have methods, events, and properties; it can maintain its state in one or more fields; and it can contain nested types. A class's visibility can be public, which means it's available to any other type, or assembly, which means it's available only to other classes in the same assembly. Classes have one or more constructors, which are initialization methods that execute when a new instance of this class is created. A class can directly inherit from at most one other class and can act as the direct parent for at most one inheriting child class.
- **Interface:** AN Interface can include methods, properties, and events. Unlike classes, interfaces do support multiple inheritance, so an interface can inherit from one or more other interfaces simultaneously. An interface does not actually implement anything, however. Instead, it provides a way to group type definition together; leaving the implementation to whatever type supports the interface.
- **Array:** An array is a group of values of the same type. Arrays can have one or more dimensions, and their upper bounds and lower bounds can be set more or less arbitrarily. All arrays inherit from a common System.Array type.
- **Delegate:** A delegate is effectively a pointer to a method. All delegates inherit from a common System.Delegate type, and they are commonly used for event handling and callbacks. Each delegate has a set of associated members called an invocation list. When the delegate is invoked, each member on this list gets called; with each one passed the parameters that the delegate received.

Value Type Vs Reference Type

A basic difference between value types and reference types is that a standalone instance of a value type is allocated on the stack, while an instance of a reference type has only a reference to its actual value allocated on the stack. The value itself is allocated on the heap.
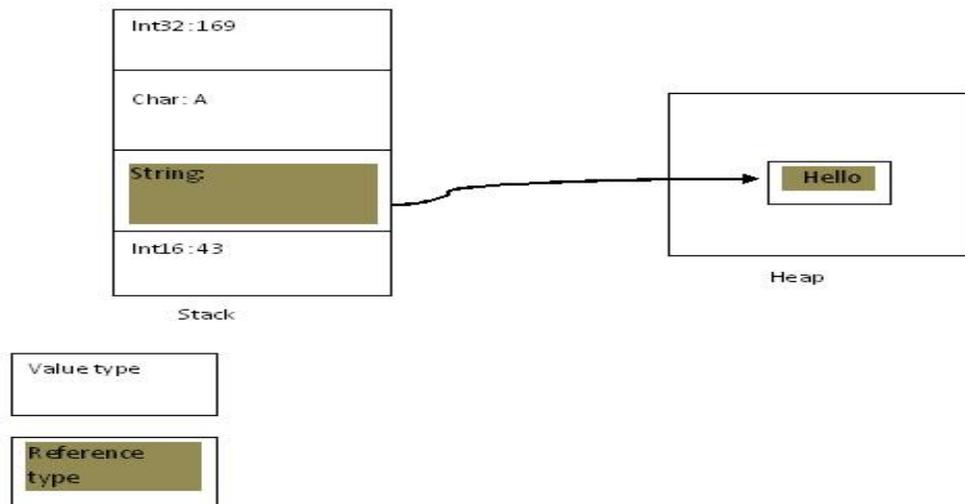

Fig 1.5 Stack and Heap

The figure 4 shows the abstract picture of how this looks. In the case shown here, three instances of value types – Int16, Char, and Int32 – have been created on the managed stack, while one instance of the reference type instance has an entry on the stack – it's a reference to the memory on the heap – but the instance's contents are stored on the heap. Understanding the distinction between value types and reference types is essential in understanding the CTS type system and, ultimately, the types used by CLR-based languages.

Converting Value Types to Reference Types: Boxing32

There are cases when an instance of a value type needs to be treated as an instance of a reference type. For example, suppose we would like to pass an instance of a value type as a parameter to some method, but that parameter is defined to be a reference to a value rather than the value itself. For situations like this, a value type instance can be converted into reference type instance through a process called Boxing.

When a value type instance is boxed, storage is allocated on the heap, and the instance's value is copied into that space. A Reference to this storage is placed on the stack, as shown in the figure.
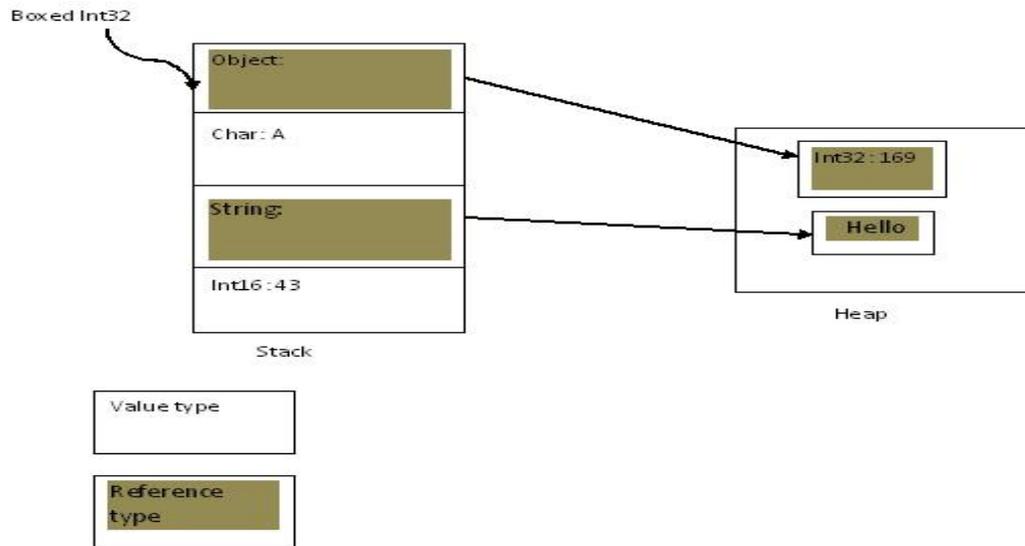

Fig 1.6 Boxing

A boxed value type instance can also be converted back to its original form, a process called unboxing.

COMMON LANGUAGE SPECIFICATION

- CLS stands for Common Language Specification and it is a subset of CTS. It defines a set of rules and restrictions that every language must follow which runs under .NET framework.

- The languages which follows these set of rules are said to be CLS Compliant. In simple words, CLS enables cross-language integration.
- **For example**, one rule is that you cannot use multiple inheritance within .NET Framework. As you know C++ supports multiple inheritance but; when you will try to use that C++ code within C#, it is not possible because C# doesn't supports multiple inheritance.
- One another rule is that you cannot have members with same name with case difference only i.e. you cannot have add() and Add() methods. This easily works in C# because it is case-sensitive but when you will try to use that C# code in VB.NET, it is not possible because VB.NET is not case-sensitive.

7. Give a brief note on MSIL?

MICROSOFT INTERMEDIATE LANGUAGE (MSIL)

When source code written in a CLR-based language is compiled, two things are produced:
- Instructions expressed in Microsoft Intermediate Language (MSIL), and
- Metadata, information about those instructions and the data they manipulate.

The figure illustrates this process. The code being compiled contains three CTS types, all of them classes. When this code is compiled using whatever compiler is appropriate for the language it's written in, the result is an equivalent set of MSIL code for each class along with metadata describing those classes.

Both the MSIL and the metadata are stored in a standard Windows portable executable (PE) file. This can be either a DLL or an EXE file.
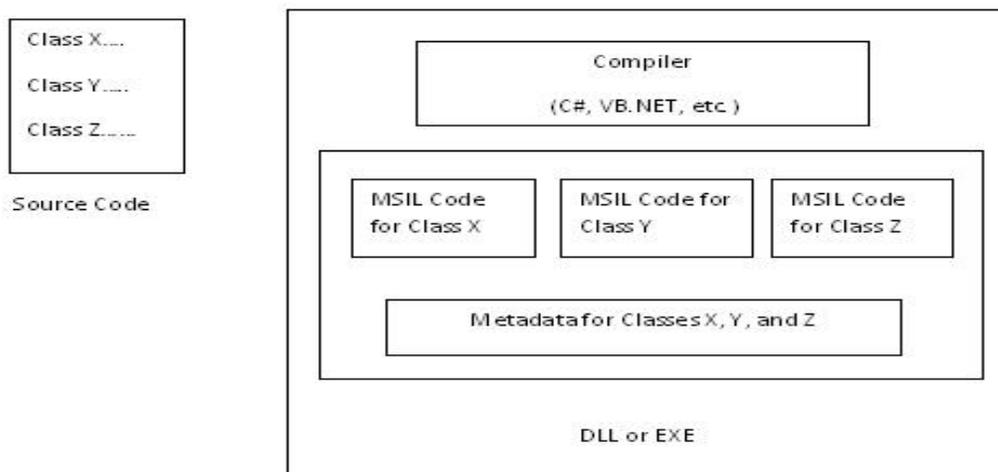

Fig 1. 7: Microsoft Intermediate language (MSIL)

FEATURES

MSIL is quite similar to a processor's native instruction set. However, no hardware that actually executes these instructions is available. Instead, MSIL code is always translated into native code for whatever processor this code is running on before it's executed. Here are a few example MSIL instructions and what they are used for:

- Add: Adds the top two values on the stack and pushes the result back onto the stack.
- Box: Converts a value type to a reference type; that is, it boxes the value.
- Br: Transfers control to a specified location in memory
- Call: calls a specified method.
- Idfld: Loads a specified field of an object onto the stack.
- Idobj: Copies the value of a specified value type onto the stack.
- Newobj: Creates a new object or a new instance of a value type.
- Stfd: Stores a value from the stack into a specified field of an object.
- Stobj: stores a value on the stack into a specified value type.
- Unbox: Converts a boxed value type back to its ordinary form.

METADATA

Compiling managed code always produces MSIL. Compiling managed code also always produces metadata describing that code. Metadata is information about the types defined in the managed code it's associated with, and it's stored in the same file as the MSIL generated from those types.

Compiling managed code generates MSIL and metadata compiling managed code when managed code is compiled, two things are produced: instruction expressed in MSIL (Microsoft Intermediate Language) and metadata.
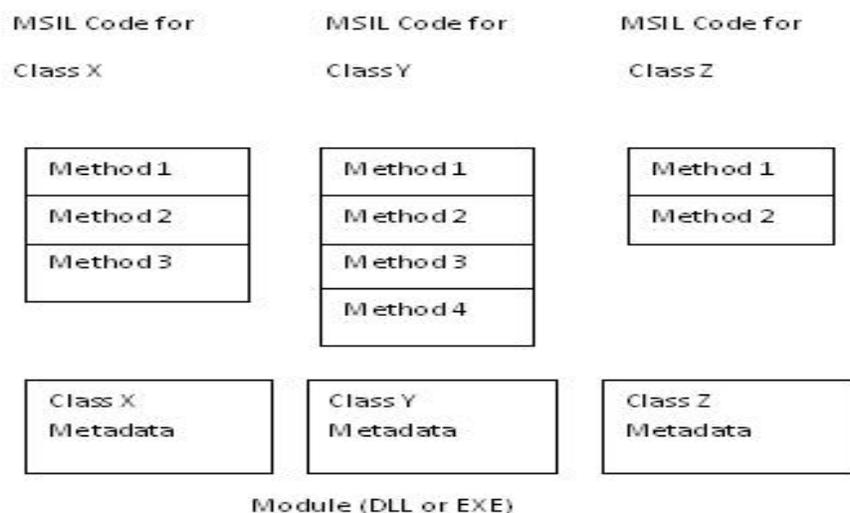

Fig 1.8 A module contains metadata for each type in the file.

The figure 8 shows an abstract view of a module produced by a CLR-based compiler. The file contains the MSIL code generated from the types in the original program, which once again are the types in the original program, which once again are the three classes X, Y and Z. Along with the code for the methods in each class, the file contains metadata describing these classes and any other types defined in this file. This information is loaded into memory when the file itself is loaded, making the metadata accessible when the file itself is loaded, making the metadata accessible at runtime. Metadata can also be read directly from the file contains it, making information available even when code isn't loaded into memory. The process of reading metadata is known as reflection.

INFORMATION IN METADATA

Metadata describes the types contained in a module. Among the information it stores for a type are the following things.
- The type's name

- The type's visibility, which can be public or assembly

- What type this type inherits from, if any

- Any interfaces the type implements

- Any methods the type implements

- Any properties the type exposes

- Any events the type provides.

More detailed information is also available. For example, the description of each method includes the method's parameter and their types, along with the type of the method's return value.

ATTRIBUTES

Metadata also includes attributes. Attributes are values that are stored in the metadata and can be read and used to control various aspects of how this code executes. Attributes can be added to types, such as classes, and to fields, methods, and properties of those types.

Developers can also create custom attributes used to control behavior in an application-specific way. To create a custom attribute, a developer using a CLR-based programming languages such as C# or VB can define a class that inherits from System. Attribute. An instance of the resulting class will automatically have its value stored in metadata when it is compiled.

8. Explain about the components of CLR
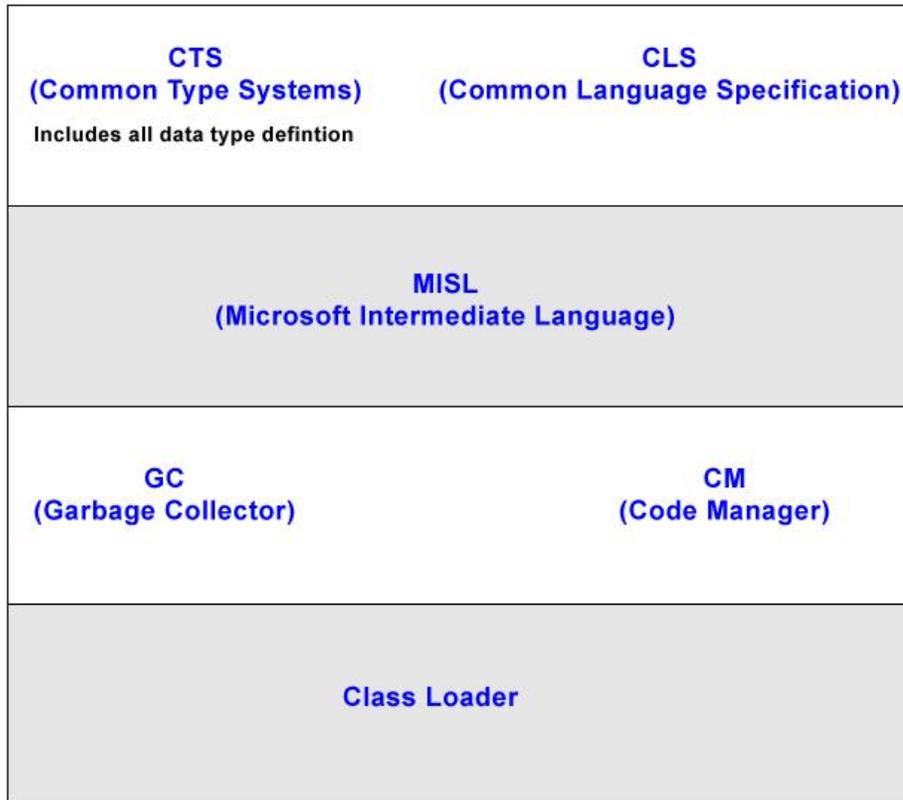
COMPONENTS OF CLR

Fig 1.9 Components of CLR

CTS(Common Type System)
CTS stands for Common Type System. It defines the rules which Common Language Runtime follows when declaring, using, and managing types.

CLS (Common Language Specification)

CLS stands for Common Language Specification and it is a subset of CTS. It defines a set of rules and restrictions that every language must follow which runs under .NET framework. The languages which follow these set of rules are said to be CLS Compliant. In simple words, CLS enables cross-language integration.

Code Manager
    Code manager invokes class loader for execution.
        .NET supports two kind of coding
       1) Managed Code
      2) Unmanaged Code

- **Managed Code**

The resource, which is with in your application domain is, managed code. The resources that are within domain are faster.The code, which is developed in .NET framework, is known as managed code. This code is directly executed by CLR with help of managed code execution. Any language that is written in .NET Framework is managed code.

Managed code uses CLR which in turns looks after your applications by managing memory, handling security, allowing cross - language debugging, and so on.

- **Unmanaged Code**

The code, which is developed outside .NET, Framework is known as unmanaged code. Applications that do not run under the control of the CLR are said to be unmanaged, and certain languages such as C++ can be used to write such applications, which, for example, access low - level functions of the operating system. Background compatibility with code of VB, ASP and COM are examples of unmanaged code.

Unmanaged code can be unmanaged source code and unmanaged compile code. Unmanaged code is executed with help of wrapper classes.

Wrapper classes are of two types: CCW (COM callable wrapper) and RCW (Runtime Callable Wrapper).Wrapper is used to cover difference with the help of CCW and RCW. COM callable wrapper unmanaged code.
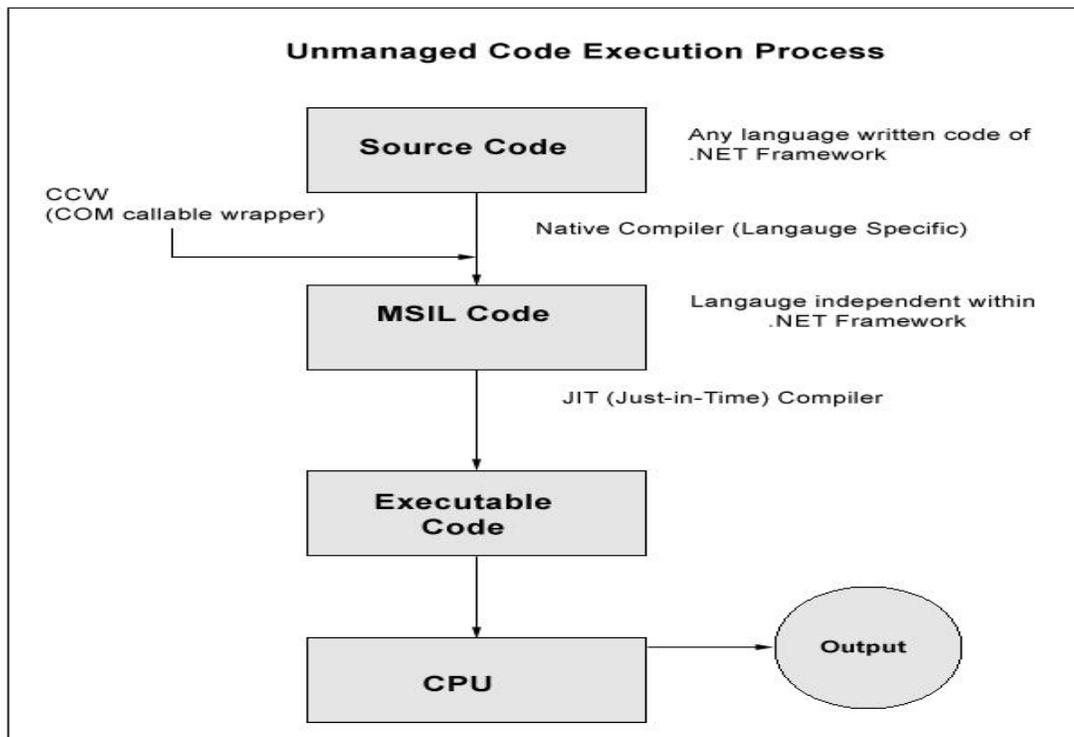
Fig1.10 Unmanaged Code Execution Process

Native Code

The code to be executed must be converted into a language that the target operating system understands, known as native code. This conversion is called compiling code, an act that is performed by a compiler.

MSIL (Microsoft Intermediate Language)

It is language independent code. When you compile code that uses the .NET Framework library, we don't immediately create operating system - specific native

code.Instead, we compile our code into Microsoft Intermediate Language (MSIL) code. The MSIL code is not specific to any operating system or to any language.

JIT (Just-in-Time)

Just - in - Time (JIT) compiler, which compiles MSIL into native code that is specific to the OS and machine architecture being targeted. Only at this point can the OS execute the application. The just - in - time part of the name reflects the fact that MSIL code is only compiled as, and when, it is needed.

Several JIT compilers exist, each targeting a different architecture, and the appropriate one will be used to create the native code required.

JIT are of three types:
- Pre JIT: It converts all the code in executable code and it is slow
- Econo JIT: It will convert the called executable code only. But it will convert code every time when a code is called again.
- Normal JIT: It will only convert the called code and will store in cache so that it will not require converting code again. Normal JIT is fast.

Assemblies

When you compile an application, the MSIL code created is stored in an assembly. Assemblies include both executable application files that you can run directly from Windows without the need for any other programs (these have a .exe file extension), and libraries (which have a .dll extension) for use by other applications.In addition to containing MSIL, assemblies also include meta information (that is, information about the information contained in the assembly, also known as metadata) and optional resources (additional data used by the MSIL, such as sound files and pictures).

Garbage Collection (GC)

One of the most important features of managed code is the concept of garbage collection. This is the .NET method of making sure that the memory used by an application is freed up completely when the application is no longer in use.

9. **Distinguish Between the .NET Compilers**

DISTINGUISH BETWEEN THE .NET COMPILERS

- The high level programming languages that need to be compiled require a runtime, so that the architecture on which the language runs is provided with details on how to execute its code.
- All the programming languages use its corresponding runtime to run the application.
- For example, to run an application developed using Visual Basic, the computer on which the application will be run must be installed with the Visual Basic runtime. The Visual Basic runtime can run only the applications developed with Visual Basic and not the ones developed with any other programming language like Java.
- In the .NET Framework, all the Microsoft .NET languages use a common language runtime, which solves the problem of installing separate runtime for each of the programming languages.

- Microsoft .NET Common Language Runtime installed on a computer can run any language that is Microsoft .NET compatible.
- The main advantage of the .NET Framework is the interoperability between different languages. As all the Microsoft .NET languages share the same common runtime language, they all work well together. For example, you can use an object written in C# from Visual Basic.NET. The same applies for all the other Microsoft .NET languages.
- When you compile a Microsoft.NET language, the complier generates code written in the Microsoft Intermediate Language (MSIL). MSIL is a set of instructions that can quickly be translated into native code.
- A Microsoft.NET application can be run only after the MSIL code is translated into native machine code.
- In .NET Framework, the intermediate language is complied "just in time" (JIT) into native code when the application or component is run instead of compiling the application at development time.

SML.NET

SML.NET is a compiler for the functional programming language Standard ML that targets the .NET Common Language Runtime and which supports language interoperability features for easy access to .NET libraries.
- Major improvements to the Visual Studio Integration Package

Much more accurate and reliable Intellisense; hovering over a keyword reports the type of its smallest enclosing expression; hovering over a pattern reports the types of its bindings.
- Major improvements to Debugger Support

Bindings in the Locals window now enter and exit scope appropriately. The values of sub-expressions, not just identifiers, are reported as locals in Locals window. Most constructed values now have symbolic tags derived from the constructor name. Values of heap-allocated SML datatypes now support ToString() and ToString(int depth) methods that can be invoked in the VS Immediate window to inspect values at runtime. Improved stepping behaviour. SML.NET stack frames now typically have meaningful, not mangled, source names.
- Preliminary support for Whidbey

The distribution has been updated to support the current 2.0 release of the Microsoft .NET Framework and Microsoft Visual Studio .NET 2005. SML.NET remains compatible with the initial 1.0 and 1.1 releases. NB: Although SML.NET fully supports SML polymorphism, it does not yet produce or consume .NET generics (we hope to in future): this release just allows you to continue working with your existing SML.NET code on the Whidbey platform.
- Improved code generation and performance

SML.NET now makes much better use of locals and the stack; pattern matching is compiled as a switch when appropriate.
- Various bug fixes

Including: the annoying (but benign) overflow error when persisting compilation units has been fixed; SML.NET now exploits some previously missed opportunities for tail-recursion.

Features

- ✓ Support for all of Standard ML
- ✓ Support for the Basis library
- ✓ Seamless interoperability with other languages

SML.NET extends the SML language to support safe, convenient use of the .NET Framework libraries and code written in other languages for the CLR, such as C# or VB. SML.NET can both consume and produce .NET classes, interfaces, delegates etc.

- ✓ Command-line compilation
- ✓ SML.NET supports traditional compilation from the command-line.
- ✓ Interactive compilation environment
- ✓ Automatic dependency analysis

In either mode of compilation, the compiler requires only the names of root modules and a place to look for source code. It then does dependency analysis to determine which files are required and which need recompilation.

- ✓ Produces verifiable CLR IL

The output of the compiler is verifiable MSIL (Microsoft Intermediate Language) for the CLR.

- ✓ Whole program optimization

SML.NET performs optimizations on a whole program (or library) at once. It usually produces small executables with fairly good performance.

- ✓ Integration with Visual Studio .NET

A binary distribution includes an experimental package for Microsoft Visual Studio .NET 2002, 2003 and & 2005 that allows you to edit, build and debug SML.NET projects from within the development environment.

10. How are the managed code organized and executed?

ORGANISING AND EXECUTING MANAGED CODE

ORGANIZING MANAGED CODE: ASSEMBLIES

- In .Net Framework applications, files that make up a logical unit of Functionality are grouped into an assembly.
- An assembly is one or more files that comprise a logical unit. Assemblies are fundamental to developing, deploying, and running .Net Framework applications.

Metadata for Assemblies: Manifests
- An assembly's manifest, by contrast, contains information about all of the modules and other files in an assembly.
- A manifest is metadata about an assembly.

- The manifest is contained in one of the assembly's files, and it includes information about the assembly and the files that comprise it.

- The figure shows, an assembly can be built from a single file or a group of files. With a single-file assembly, the manifest is stored in the file itself.
- With a multiple assembly, the manifest is stored in one of the files in the assembly

  Among the things an assembly's manifest includes are the following:
  - **The name of the assembly:** All assemblies have a text name and can optionally have a strong name.
  - **The assembly's version number**: This number has the form <major version>.<minor version>.<build version>.<revision>. The versioning is per assembly, not per the module.
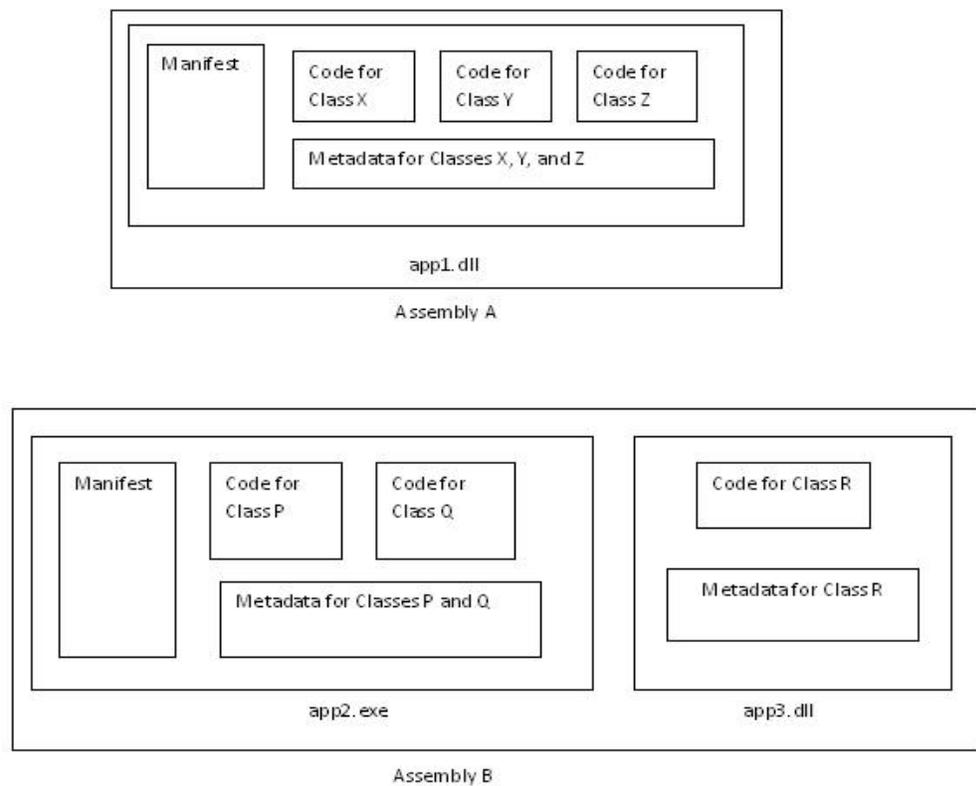

Fig 1.11: Organizing Managed Code: Assemblies

- **The assembly's culture**: indicating the culture or language an assembly supports.
- A list of all files contained in this assembly, together with a hash value that's been computed from those files.

What other assemblies this one depends on and the version number of each of those dependent assemblies.

Most assemblies consist of just a single DLL. Whether it contains one file or multiple files, however, an assembly is logically an indivisible unit.

Categorizing Assemblies

There are various ways to categorize assemblies. One distinction is between static and dynamic assemblies.

Static Assemblies

Static assemblies are produced by a tool such as Visual Studio, and their contents are stored on disc.

Dynamic Assembly

The code (and metadata) for a dynamic assembly is created directly in memory and can be executed immediately upon creation. Once it has been created, a dynamic assembly can be saved to disk, then loaded and executed again. Probably the most common examples of dynamic assemblies are those created by ASP.NET when it processes .aspx pages.

Categorizing based on assembly name

- Another way to categorize assemblies is by how they are named. Completely naming any assembly requires specifying three things: the assembly's name; its version number; and if one is provided, the culture it supports.
- All assemblies have simple text names, such as "Account Access", but it also includes the usual three parts of an assembly name.
- A strong name includes the usual three parts of an assembly name, but it also includes a digital signature computed on the assembly and the public key that corresponds to the private key used to create that signature.

EXECUTING MANAGED CODE

Assemblies provide a way to package modules containing MSIL and metadata into units for deployment. Assemblies are loaded into memory only when they are needed.

Loading Assemblies

- When an application built using the .NET Framework is executed, the assemblies that make up that application must be found and loaded into memory.
- Assemblies aren't loaded until they're needed, so if an application never calls any methods in a particular assembly, that assembly won't be loaded.
- If the assemblies are not loaded automatically, the CLR determines what version of a particular assembly it's looking for.
- By default, it will look only for the exact version specified for this assembly in the manifest of the assembly from which the call originated.
- Once it has determined exactly which version it needs, the CLR checks whether the desired assembly is already loaded. If it is, the search is over; this loaded version will be used.

- If the desired assembly is not already loaded, the CLR will begin searching in various places to find it.
- The first place the CLR looks is usually the global assembly cache (GAC), a special directory intended to hold assemblies that are used by more than one application.
- If the assembly it's hunting for isn't in the global assembly cache, the cache continues its search by checking for a codebase element in one of the configuration files for this application.
- If there is no codebase element, however, the CLR will begin its last-ditch search for the desired assembly, a process called probing, in what's known as the application base. This can be either the root directory in which the application is installed or a URL, perhaps on some other machine.

Compiling MSIL

A compiler that produces managed code always generates MSIL. Yet MSIL can't be executed by any real processor. Before it can be run, MSIL code must be compiled yet again into native code that targets the processor on which it will execute. Two options exist for doing this: MSIL code can be compiled one method at a time during execution, or it can be compiled into native code all at once before an assembly is executed.

JIT Compilation

The most common way to compile MSIL into native code is to let the CLR load an assembly and then compile each method the first time that method is invoked. Because each method is compiled only when it's first called, the process is called just-in-time (JIT) compilation.
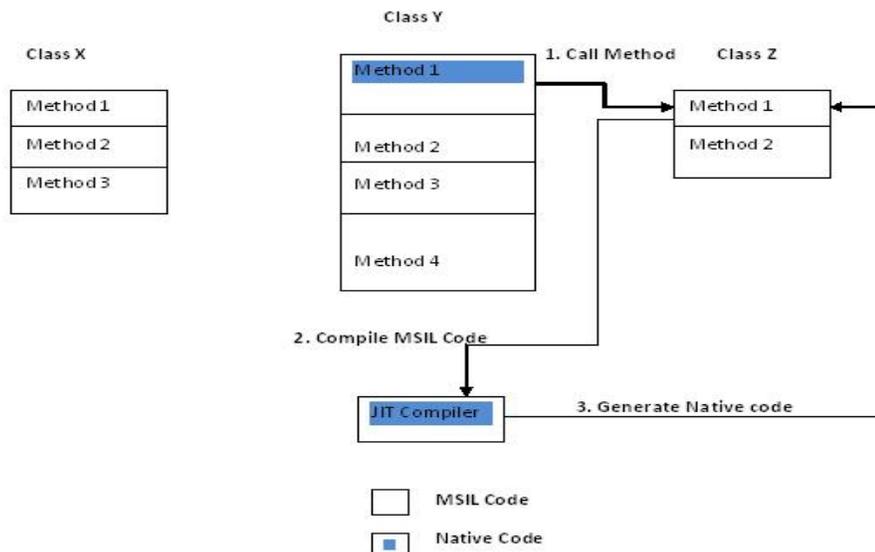
The first time class Z's method 1 is called, the JIT compiler is invoked to translate the method's MSIL into native code

This is simple example that shows just three classes, once again called X, Y and Z, each containing some number of methods. In the Figure only method 1 of

class Y has been compiled. All other code in all other methods of the three classes is still in MSIL, the form in which it was loaded. When class Y's method 1 calls class Z's method 1, the CLR notices that this newly called method is not in an executable form. The CLR invokes the JIT compiler, which class Z's method 1 and redirects calls made to that method to this compiled native code. The method can now execute.

When Class Y's method 4 is called, the JIT compiler is once again used to translate the method's MSIL into native code.
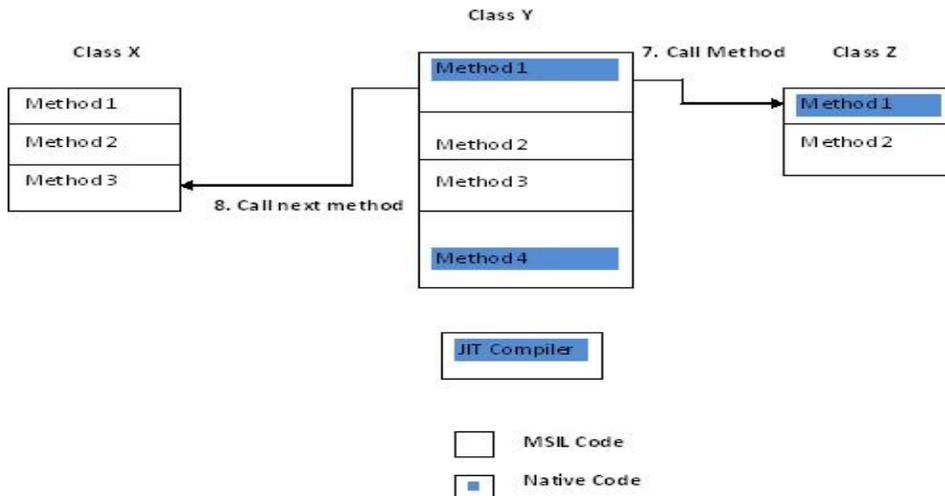

Fig 1.14 When class Z's method 1 is called again, no compilation is necessary

Figure shows what happens when class Y's method 1 again calls method 1 in class Z. This method has already been JIT compiled, so there's no need to do any more work. The native code has been saved in memory, so it just executes. The JIT compiler isn't involved. The process continues in this same way, with each method compiled the first time it is invoked.

11. Write about serialization

SERIALIZATION

- **Serialization** is a process of converting an object into a stream of data so that it can be is easily transmittable over the network or can be continued in a persistent storage location.
- This storage location can be a physical file, database or ASP.NET Cache.
- The advantage of serialization is the ability to transmit data across the network in a cross-platform-compatible format, as well as saving it in a persistent or non-persistent storage medium in a non-proprietary format.
- Serialization is used by Remoting, Web Services SOAP for transmitting data between a server and a client. De-serialization is the reverse; it is the process of reconstructing the same object later.

- The Remoting technology of .NET makes use of serialization to pass objects by value from one application domain to another.
- Serialization in .NET is provided by the System.Runtime.Serialization namespace. This namespace contains an interface called IFormatter which in turn contains the methods Serialize and De-serialize that can be used to save and load data to and from a stream.
- In order to implement serialization in .NET, we basically require a stream and a formatter. While the stream acts as a container for the serialized object(s), the formatter is used to serialize these objects onto the stream.
- The basic advantage of serialization is the ability of an object to be serialized into a persistent or a non-persistent storage media and then reconstructing the same object if required at a later point of time by de-serializing the object.
- Remoting and web services depend heavily on Serialization and De-serialization.

WORKING WITH FORMATTERS

- A formatter is used to determine the serialization format for objects.
- They expose an interface called the IFormatterinterface. IFormatter's significant methods are Serialize and De-serialize which perform the actual serialization and de-serialization.
- There are two formatter classes provided within .NET, the BinaryFormatter and theSoapFormatter. Both these classes extend the IFormatter interface.
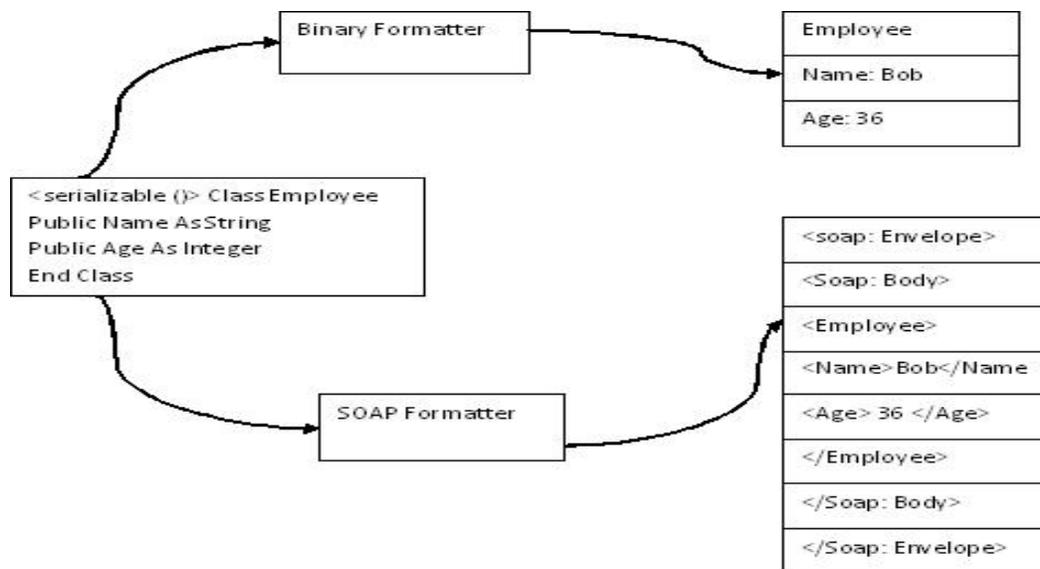
The figure illustrates the serialization process. As the figure shows, an instance of a class can be run through a formatter that extracts the state of this object in a particular form. The binary formatter emits that state information in a simple and

compact form, while the SOAP formatter generates the same information wrapped in XML and formatted as a SOAP message. While the outputs shown in the figure are simplified – the binary formatter actually stores integers in binary form, for instance – they illustrate the key difference between the two serialization options built into the .NET Framework class library.

ADVANTAGES AND DISADVANTAGES OF SERIALIZATION

Advantages
- Modification of XML documents without using the DOM.
- Passing an object from one application to another.
- Passing an object from one domain to another.
- Passing an object through a firewall as an XML string

Disadvantages
The primary disadvantage of serialization can be attributed to the resource overhead (both the CPU and the IO devices) that is involved in serializing and de-serializing the data and the latency issues that are involved for transmitting the data over the network. Further, serialization is quite slow. Moreover, XML serialization is insecure, consumes a lot of space on the disk and it works on public members and public classes and not on the privateor internal classes. Therefore, it compels the developer to allow the class to be accessed by the outside world.

12. Explain briefly about the input and output?

INPUT AND OUTPUT NAMESPACE

The System.IO namespace contains types that allow reading and writing to files and data streams, and types that provide basic file and directory support.
Classes

The table shows the classes with their descriptions.

Table 1.2 Classes (write any 10)

Class	Description
BinaryReader	Reads primitive data types as binary values in a specific encoding.
BinaryWriter	Writes primitive types in binary to a stream and supports writing strings in a specific encoding.
BufferedStream	Adds a buffering layer to read and write operations on another stream. This class cannot be inherited.

Directory	Exposes static methods for creating, moving, and enumerating through directories and subdirectories.
DirectoryInfo	Exposes instance methods for creating, moving, and enumerating through directories and subdirectories.
DirectoryNotFoundException	The exception that is thrown when part of a file or directory cannot be found.
EndOfStreamException	The exception that is thrown when reading is attempted past the end of a stream.
ErrorEventArgs	Provides data for the Error event.
File	Provides static methods for the creation, copying, deletion, moving, and opening of files, and aids in the creation of FileStream objects.
FileInfo	Provides instance methods for the creation, copying, deletion, moving, and opening of files, and aids in the creation of FileStream objects.
FileLoadException	The exception that is thrown when a managed assembly is found but cannot be loaded.
FileNotFoundException	The exception that is thrown when an attempt to access a file that does not exist on disk fails.
FileStream	Exposes a Stream around a file, supporting both synchronous and asynchronous read and write operations.
FileSystemEventArgs	Provides data for the directory events: Changed, Created, Deleted.
FileSystemInfo	Provides the base class for both FileInfo and DirectoryInfo objects.
FileSystemWatcher	Listens to the file system change notifications and raises events when a directory, or file in a directory, changes.
InternalBufferOverflowException	The exception thrown when the internal buffer overflows.
IODescriptionAttribute	Sets the description visual designers can display when referencing an event,

	extender, or property.
IOException	The exception that is thrown when an I/O error occurs.
MemoryStream	Creates a stream whose backing store is memory.
Path	Performs operations on String instances that contain file or directory path information. These operations are performed in a cross-platform manner.
PathTooLongException	The exception that is thrown when a pathname or filename is longer than the system-defined maximum length .
RenamedEventArgs	Provides data for the Renamed event.
Stream	Provides a generic view of a sequence of bytes.
StreamReader	Implements a TextReader that reads characters from a byte stream in a particular encoding.
StreamWriter	Implements a TextWriter for writing characters to a stream in a particular encoding.
StringReader	Implements a TextReader that reads from a string.
StringWriter	Implements a TextWriter for writing information to a string. The information is stored in an underlying StringBuilder.
TextReader	Represents a reader that can read a sequential series of characters.
TextWriter	Represents a writer that can write a sequential series of characters. This class is abstract.

Structures(write all)

Structure	Description
WaitForChangedResult	Contains information on the change that occurred.

Delegates(write all)

Delegate	Description
ErrorEventHandler	Represents the method that will handle the Error event of a FileSystemWatcher.
FileSystemEventHandler	Represents the method that will handle the Changed, Created, or Deleted event of a FileSystemWatcher class.
RenamedEventHandler	Represents the method that will handle the Renamed event of a FileSystemWatcher class.

Enumerations(write all)

Enumeration	Description
FileAccess	Defines constants for read, write, or read/write access to a file.
FileAttributes	Provides attributes for files and directories.
FileMode	Specifies how the operating system should open a file.

13. How does .NET incorporate with the XML?

WORKING WITH XML

Microsoft .NET provides a good variety of ways to work with .xml files. For instance, the Dataset object includes methods that allow users to load the contents of .xml files to a database table or to save the table to an .xml file. Microsoft .NET libraries also have specific classes used to parse .xml files.

Normally, XML parsers use either Document Object Model (DOM) or Simple API for XML (SAX). DOM parsers read the whole document and create the document scheme in memory. They provide random access to elements of .xml files and allow developers to read and write information from or to .xml files.

SAX provides read-only access to .xml files. However, it uses less memory and works faster than DOM.

SAX is an event-driven push parser. When it reads an XML document it generates specific events. To handle these events, an application must implement a specific event-handler interface that contains methods which handle appropriate events:
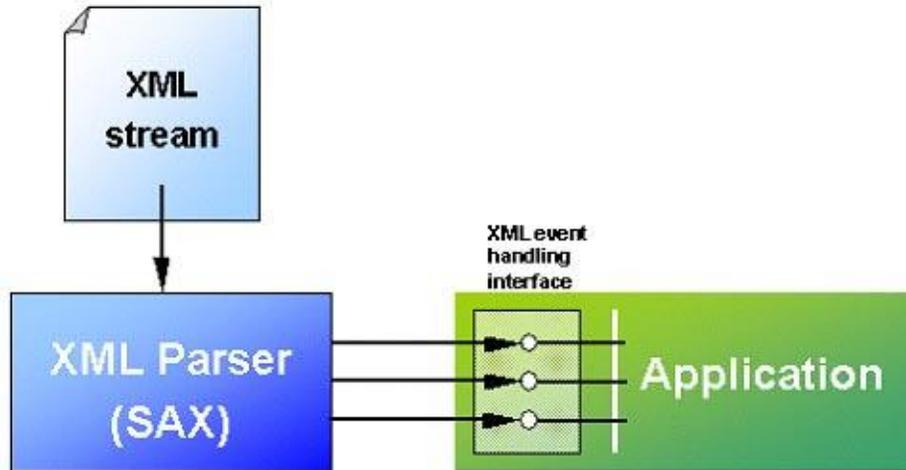
Fig 1.19 Working of XML Parser

An alternative to the push model is the pull one. A pull parser does not "push" events to the application for processing. Instead, the application pulls data from the parser by calling special methods of the parser object:
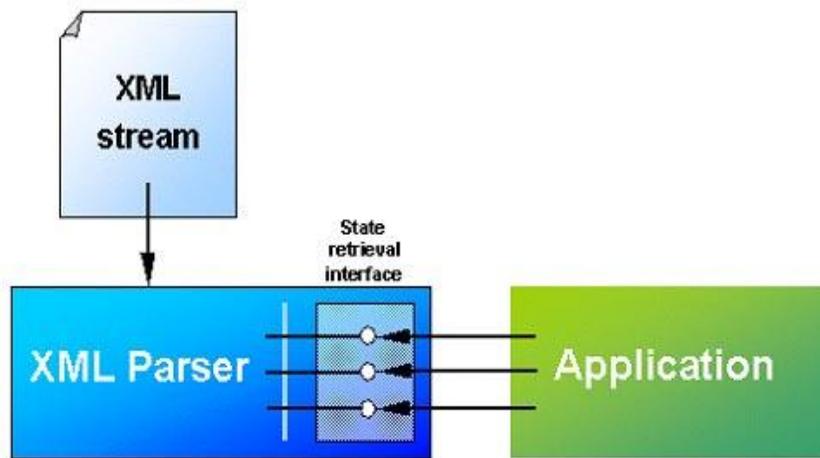
Fig 1.20 Working of XML Parser with inteface

In many cases the pull model is more convenient than the push one, which is why most parsers implement the pull model.We will work with an .xml file, dataset.xml, which is generated from the following table (MS SQL Server):

Table 1.3 Fields

Field	Type
ID	INTEGER (the identity column)
Text	TEXT
VarChar	VARCHAR(100)

Data	DECIMAL

XmlReader:

      Like SAX parsers, XMLReader provides forward-only read-only access to .xml files. However, unlike SAX parsers, it uses the pull model. Note that XMLReader is an abstract class (MustOverride in VB). To parse .xml files we will use its descendant, XMLTextReader.

XMLDocument

      The XMLDocument object works with .xml files using DOM (Document Object Model): It implements W3C Document Object Model (DOM) Level 1 Core and Core DOM Level 2. XMLDocument contains specific methods used to read and write XML documents and navigate the DOM tree.

DataSet

      One of the ways to process data stored in an .xml file is to load this file to a dataset and then process the data in the dataset fields. When the processing is over, the dataset records can be exported to an .xml file.

Profiling

      To measure the performance of XMLReader, XmlDocument and Dataset objects we've created several tables with a different number of records.

14. Write short notes on remoting?

REMOTING

- Remoting provides a framework that allows objects to interact with one another across application domains.
- The framework provides a number of services, including activation and lifetime support, as well as communication channels responsible for transporting messages to and from remote applications.
- Formatters are used for encoding and decoding the messages before they are transported by the channel.
- Applications can use binary encoding where performance is critical, or XML encoding where interoperability with other remoting frameworks is essential.
- All XML encoding uses the SOAP protocol in transporting messages from one application domain to the other.

Types of .NET Remotable Objects

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

Single Call

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

Singleton Objects

Singleton objects are those objects that service multiple clients, and hence share data by storing state information between client invocations. They are useful in cases in which data needs to be shared explicitly between clients, and also in which the overhead of creating and maintaining objects is substantial.

Client-Activated Objects (CAO)

- Client-activated objects (CAO) are server-side objects that are activated upon request from the client.
- When the client submits a request for a server object using a "new" operator, an activation request message is sent to the remote application.
- The server then creates an instance of the requested class, and returns an ObjRef back to the client application that invoked it.
- A proxy is then created on the client side using the ObjRef. The client's method calls will be executed on the proxy.

REMOTE OBJECTS

- Any object outside the application domain of the caller should be considered remote, even if the objects are executing on the same machine.
- Inside the application domain, all objects are passed by reference while primitive data types are passed by value.
- Any object can be changed into a remote object by deriving it from MarshalByRefObject.
- When a client activates a remote object, it receives a proxy to the remote object.
- All operations on this proxy are appropriately indirected to enable the remoting infrastructure to intercept and forward the calls appropriately.

PROXY OBJECTS

- Proxy objects are created when a client activates a remote object.
- The proxy object acts as a representative of the remote object and ensures that all calls made on the proxy are forwarded to the correct remote object instance.
- A remote object is registered in an application domain on a remote machine.
- The object is marshaled to produce an ObjRef.
- The ObjRef contains all the information required to locate and access the remote object from anywhere on the network.
- This information includes the strong name of the class, the class's hierarchy (its parents), the names of all the interfaces the class implements, the object URI, and details of all available channels that have been registered.
- The remoting framework uses the object URI to retrieve the ObjRef instance created for the remote object when it receives a request for that object.

- A client activates a remote object by calling new or one of the Activator functions like CreateInstance.

15. Write about the enterprise services?

ENTERPRISE SERVICES

- **Enterprise services** are SAP's term for **services** that have the proper scope to play a productive role in automating business processes in**enterprise** computing. The scope of an **enterprise services** is important because if too much functionality is included in a service, it becomes complicated to reuse.
- COM (Component Object Model) provides one way to write component-based applications.
- It is well known that the plumbing work required to write COM components is significant and repetitive. COM+ is not so much about a new version of COM; rather, COM+ provides a services infrastructure for components.
- Components are built and then installed in COM+ applications in order to build scalable server applications that achieve high throughput with ease of deployment.
- Services components in .NET are able to use COM+ services such as declarative and distributed transactions, role based security, object pooling messaging. To access these services your class needs to derive from the class System.EnterpriseServices.ServicedComponent, adorn your class and assemblies with relevant attributes, and configure your application by registering your serviced components with the COM+ catalog.
- The overall landscape of accessing and using COM+ services within .NET goes by the name .NET Enterprise Services.
- Many of these services can be provided without the need to derive from a ServicedComponent though the use of Spring's Aspect-Oriented Programming functionality.
- By using Spring's ServicedComponentExporter, EnterpriseServicesExporter and ServicedComponentFactory you can easily create and consume serviced components without having your class inherit from ServicedComponent and automate the manual deployment process that involves strongly signing your assembly and using the regsvcs utility.

Server Side
- Spring.Enterprise.ServicedComponentExporter is responsible for exporting a single component and making sure that it derives from ServicedComponent class.
- It also allows you to specify class-level and method-level attributes for the component in order to define things such as transactional behavior, queuing, etc.

- Spring.Enterprise.EnterpriseServicesExporter corresponds to a COM+ application, and it allows you to specify list of components that should be included in the application, as well as the application name and other assembly-level attributes.

16. Write about interoperability in .NET?
(OR)
Write about Interoperability of COM components with .NET

INTEROPERABILITY IN .NET

- The .NET CLR enables interoperability by hiding the complexity associated with calls between managed and unmanaged code.
- The runtime automatically generates code to translate calls between the two environments.
- .NET manages the following aspects of interoperability between managed and unmanaged code:

Object binding
Both early and late bound interfaces are supported.
Data marshaling and translation
Data type conversion is handled between managed and unmanaged data types.
Object lifetime management
Object references are managed to ensure that objects are either released or marked for garbage collection.
Object identity
COM object identity rules are enforced.
Exception and error handling
The runtime translates COM HRESULT values to .NET exceptions and vice versa.

The important goals of .NET during its development was to promote interoperability with existing technologies. .NET interoperability comes in three types:
- ✓ Interoperability of .NET code with COM components (called as COM interop)

- ✓ Interoperability of COM components with .NET (called .NET interop)

- ✓ Interoperability of .NET code with Win32 DLLs (called P/Invoke)

- .NET runtime allows us to use legacy COM code from .NET components. We can call it backward compatibility.
- In the same way, .NET runtime also provides us forward compatibility, means accessing .NET components from COM components.

Differences Between .NET Framework and COM Framework :
- The .NET framework object model and its workings are different from Component Object Model (COM) and its workings. For example, clients of .NET

components don't have to worry about the lifetime of the object. Common Language Runtime (CLR) manages things for them. In contrast, clients of COM objects must take care of the lifetime of the object.

- Similarly, .NET objects live in the memory space that is managed by CLR. CLR can move objects around in the memory for performance reasons and update the references of objects accordingly, but COM object clients have the actual address of the object and depend on the object to stay on the same memory location.

- Similarly, .NET runtime provides many new features and constructs to managed components. For example, .NET components can have parameterized constructors, functions of the components can have accessibility attributes (like public, protected, internal, and others) associated with them, and components can also have static methods. Apart from these features, there are many others. These include ones that are not accessible to COM clients because standard implementation of COM does not recognize these features. Therefore .NET runtime must put something in between the two, .NET server and COM client, to act as mediator.Interoperability of .NET code with COM components (called as COM interop)

A .NET component in C# named CManagedServer. This component is in the ManagedServer assembly. Client side, I have created a Visual Basic (VB) 6.0-based client that uses the services of our managed server.
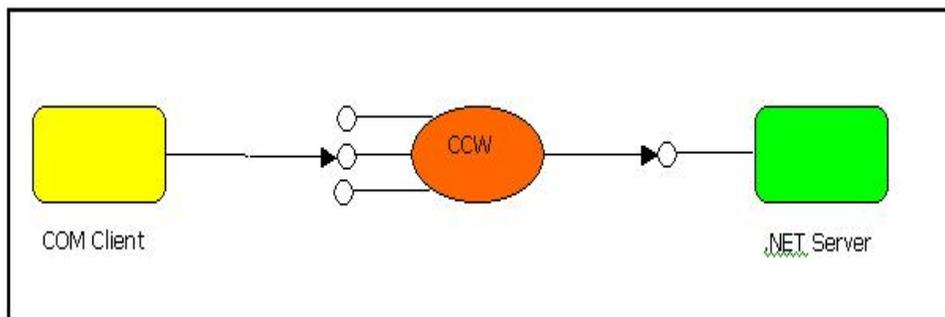
Fig Interoperability of COM components with .NET

- Components written in .NET are managed whereas components written using COM are unmanaged.
- Therefore, the first challenge to overcome is, to bridge these two models.
- Each model has its own way of memory allocation, object lifetime management and parameter passing convention that to bridge these two models require the usage of an intermediary that handles all these differences.
- Having an intermediary is important because we do not want applications writing this code for each application that wants to interop with COM components.

- The .NET developers were aware of this and thus gave an intermediary called as the Runtime Callable Wrapper (RCW).
- The RCW takes care of all the intricacies of communicating between the two platforms. The following figure shows the role of the RCW.

Fig 1 RCW

Runtime Callable Wrapper(RCW)

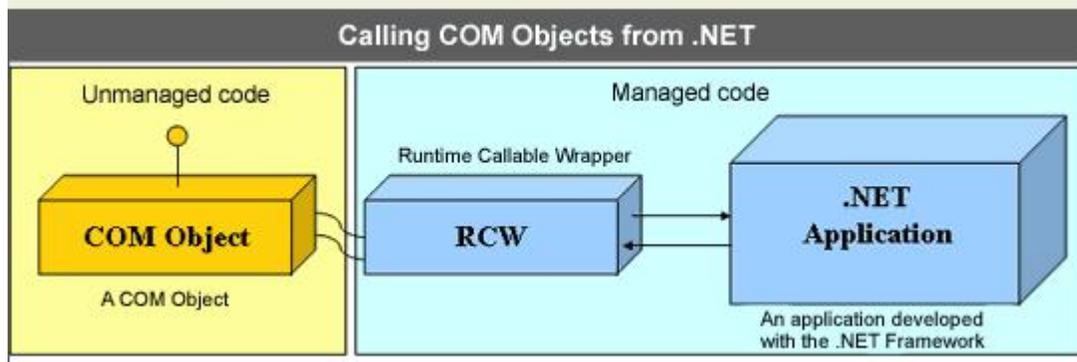- The main job of the RCW is to hide all the differences between the two worlds and as such is an object that is created from the managed heap.
- Only one instance of the RCW is ever created, irrespective of how many .NET clients access it.
- Once a .NET client object wants to instantiate a COM client, the RCW intervenes and creates the object on your behalf and then manages the lifetime of the COM object. COM objects are reference counted.
- This means that each client accessing the COM client will increase its reference count by 1 and each release of the reference decreases its reference count by 1.
- When the reference count becomes 0, the COM client is released.
- This counting happens by calling the Add and Release methods of the IUnknown interface, an interface that all COM objects may implement.
- All these intricacies are taken over by the RCW. It is important to remember that .NET is a garbage collected environment.
- The RCW will be collected when the last client holding a reference to the COM object releases the reference.
- At this point, the reference count on the COM object becomes 0 and will thus be collected by the operating system while the RCW will be garbage collected by the .NET runtime.

17. Write about the GUI?

GUI

A graphical user interface (GUI) is a human-computer interface (i.e., a way for humans to interact with computers) that uses windows, icons and menus and which can be manipulated by a mouse (and often to a limited extent by a keyboard as well).

Visual Studio .NET on a Windows platform gives you a multitude of classes to easily create typical Windows GUI applications

Although it is possible to create windows applications purely in a textual environment, we will instead make use of the Visual Studio .NET interface which lets us graphically drag and drop to determine the layout of our program. The .NET environment will then generate the necessary code for us to perform the layout.
To create a Windows Forms Application, start Visual Studio .NET and create a new Visual C# Project. Make sure you select a Windows Application as the template.
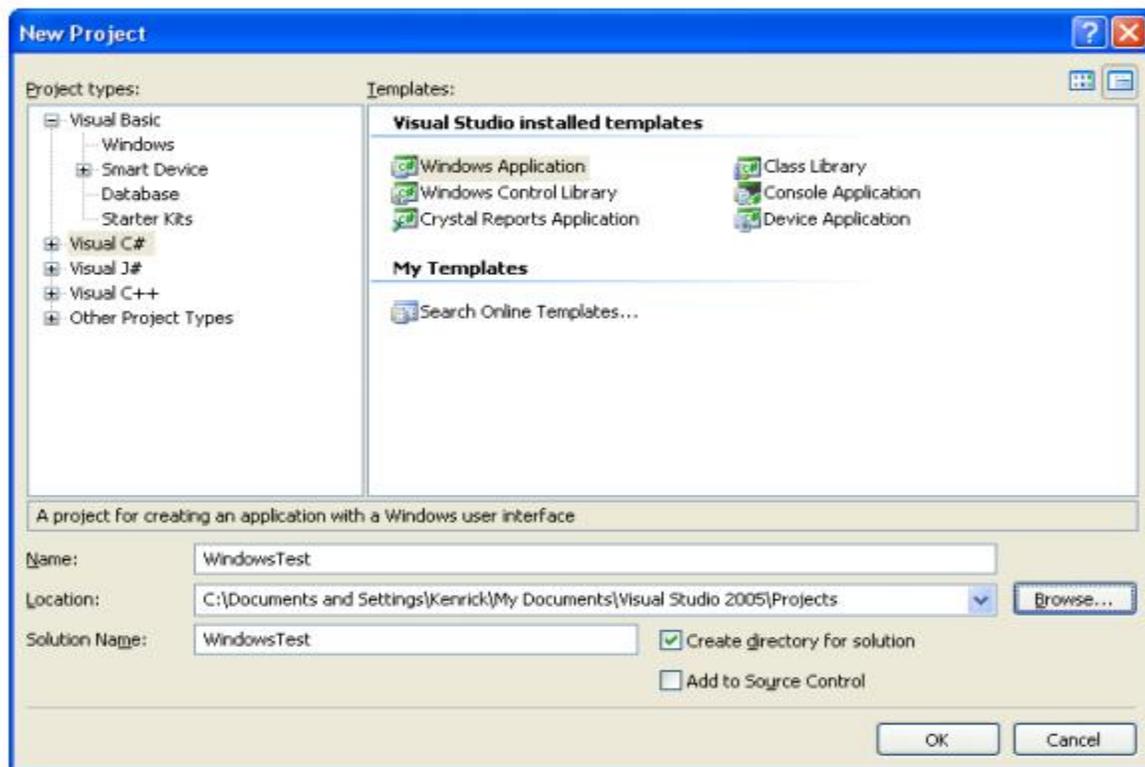
You will be presented with a blank form. This document assumes that you have the Toolbox pane visible on the left of the screen and it is set to Windows Forms. This pane contains common controls which make up the elements on the form. The layout

is shown below:
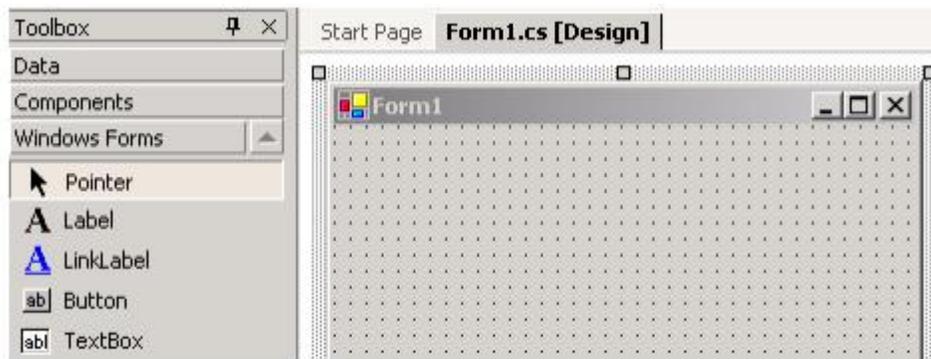
A control is a component with a visual representation. The Control class in the System.Windows.Forms namespace is the base class for the graphical objects on the screen. All controls are derived from the base control. Examples of controls include:

In this document we will only discuss a few controls: Buttons, Labels, TextBox, ProgressBar, and PictureBox. A Button is just like it sounds, a button we can press and when it is pressed some code is invoked. A Label is a way to output some text on the form. A TextBox can be used to output data and provide a way for the user to enter text to the program (although a label is better for data that is purely output only). A ProgressBar displays a graphical bar of progress for some slow event. Finally, a PictureBox is used to load and display an image

To add controls to the form, select the control you want from the Toolbox on the left. Then click and drag in the form window in the location you want the control to go. If you select the Pointer tool, you can select existing items on the form and move them elsewhere. You can also click and resize the control on the form. Try dragging a button onto the form:
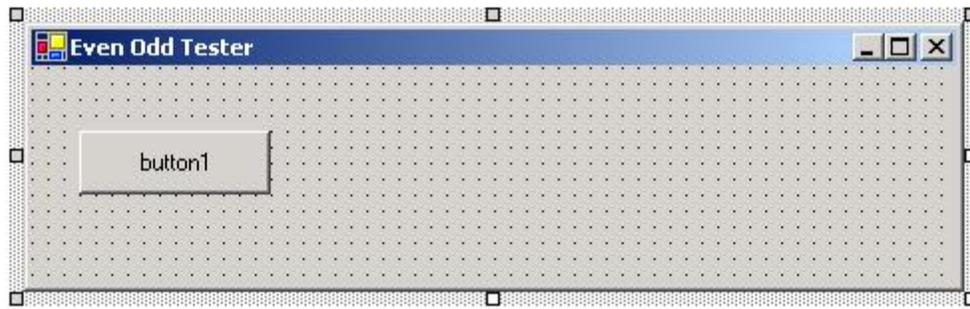
Click the button on the form and change its Name property to "buttonEvenOdd". This is setting the name of the variable used to reference the button object from its default of "Button1". Also change the button's text property to "Even or Odd". You should see the change to the text field take place on the form.
In the same fashion as the button, add a label to the form and name it "labelResult" and set its Text field to "Enter data".
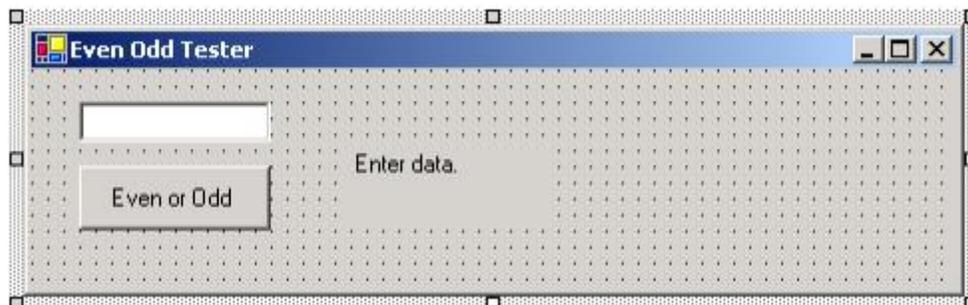
You can look at the code at any time by right-clicking on the "Form1.cs" name
and select "View Code" to see the C# code: Look figure below.

```
Form1.cs   Form1.cs [Design]
WindowsTest.Form1

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsTest
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Thus GUI has been discussed.

CS T54 PLATFORM TECHNOLOGY

UNIT-2

CONCEPT OF OOPS

At the center of C# is *object-oriented programming* (OOP). The object-oriented methodology is inseparable from C#, and all C# programs are to at least some extent object oriented. C#'s basic unit of encapsulation is the *class.* A class defines the form of an object. It specifies both the data and the code that will operate on that data. C# uses a class specification to construct *objects.* Objects are instances of a class. Thus, a class is essentially a set of plans that specify how to build an object. Collectively, the code and data that constitute a class are called its *members.* Method is C#'s term for a subroutine. Thus, the methods of a class contain code that acts on the fields defined by that class.

To support the principles of object-oriented programming, all OOP languages, including C#, have three traits in common: encapsulation, polymorphism, and inheritance.

2.1.1 Object oriented programming

- **Encapsulation**
  Encapsulation is a programming mechanism that binds together code and the data. In an objectoriented language, code and data can be bound together in such a way that an *object* is created.

- **Polymorphism**
  Concept of polymorphism is often expressed by the phrase "one interface, multiple methods." This means that it is possible to design a generic interface to a group of related activities.

- **Inheritance**
  Inheritance is the process by which one object can acquire the properties of another object.

1. Write about the Types and Variables

DATA TYPES

C# contains two general categories of built-in data types: *value types* and *reference types*. For a value type, a variable holds an actual value, such 3.1416 or 212. For a reference type, a variable holds a reference to the value. The most commonly used reference type is the class. The value types are described here.

The Boolean Type

Boolean types are declared using the keyword, *bool*. They have two values: *true* or *false*. In other languages, such as C and C++, boolean conditions can be satisfied where 0 means false and anything else means true. However, in C# the only values that satisfy a boolean condition is *true* and *false*, which are official keywords. Listing 2-1 shows one of many ways that boolean types can be used in a program.

Displaying Boolean Values: Boolean.cs

```
using System;

class Booleans
{   public static void Main()
    {      bool content = true;

      bool noContent = false;

      Console.WriteLine("It is {0} that C# Station provides C# programming language content.", content);

      Console.WriteLine("The statement above is not {0}.", noContent);

   }

}
```

In the above example the boolean values are written to the console as a part of a sentence. The only legal values for the bool type are either true or false, as shown by the assignment of true to content and false to noContent. When run, this program produces the following output:

It is True that C# Station provides C# programming language content.

The statement above is not False.

Integral Types

In C#, an integral is a category of types. For anyone confused because the word Integral sounds like a mathematical term, from the perspective of C# programming, these are actually defined as Integral types in the C# programming language specification. They are whole numbers, either signed or unsigned, and the char type. The char type is a Unicode character, as defined by the Unicode Standard. For more information, visit The Unicode Home Page. table 2-1 shows the integral types, their size, and range.

Table 2.1 Integral types

Type	Size (in bits)	Range
sbyte	8	-128 to 127
byte	8	0 to 255
short	16	-32768 to 32767
ushort	16	0 to 65535
int	32	-2147483648 to 2147483647
uint	32	0 to 4294967295
long	64	-9223372036854775808 to 9223372036854775807
ulong	64	0 to 18446744073709551615
char	16	0 to 65535

Integral types are well suited for those operations involving whole number calculations. The char type is the exception, representing a single Unicode character. As you can see from the table above, you have a wide range of options to choose from, depending on your requirements.

Floating Point and Decimal Types

A C# floating point type is either a float or double. They are used any time you need to represent a real number, as defined by IEEE 754. For more information on IEEE 754, visit the IEEE Web Site. Decimal types should be used when representing financial or money values. table 2-2 shows the floating point and decimal types, their size, precision, and range.

Table 2.2 floating point types

Type	Size (in bits)	precision	Range
float	32	7 digits	1.5×10^{-45} to 3.4×10^{38}
double	64	15-16 digits	5.0×10^{-324} to 1.7×10^{308}
decimal	128	28-29 decimal places	1.0×10^{-28} to 7.9×10^{28}

Floating point types are used when you need to perform operations requiring fractional representations. However, for financial calculations, the decimal type is the best choice because you can avoid rounding errors.

```
// Find the radius of a circle given its area.

using System;

class FindRadius {

static void Main() {

Double r;

Double area;

area = 10.0;

r = Math.Sqrt(area / 3.1416);

Console.WriteLine("Radius is " + r);

}}
```

The output from the program is shown here:

Radius is 1.78412203012729

The string Type

A string is a sequence of text characters. You typically create a string with a string literal, enclosed in quotes: "This is an example of a string." You've seen strings being used in Lesson 1, where we used the Console.WriteLine method to send output to the console. Some characters aren't printable, but you still need to use them in strings. Therefore, C# has a special syntax where characters can be escaped to represent non-printable characters. For example, it is common to use newlines in text, which is represented by the '\n' char. The backslash, '\',

represents the escape. When preceded by the escape character, the 'n' is no longer interpreted as an alphabetical character, but now represents a newline.

We have to escape that too by typing two backslashes, as in '\\'. table 2-3 shows a list of common escape sequences.

Table 2.3 String types

Escape Sequence	Meaning
\'	Single Quote
\"	Double Quote
\\	Backslash
\0	Null, not the same as the C# *null* value
\a	Bell
\b	Backspace
\f	form Feed
\n	Newline
\r	Carriage Return
\t	Horizontal Tab
\v	Vertical Tab

Another useful feature of C# strings is the verbatim literal, which is a string with a @ symbol prefix, as in @"Some string". Verbatim literals make escape sequences translate as normal characters to enhance readability. To appreciate the value of verbatim literals, consider a path statement such as "c:\\topdir\\subdir\\subdir\\myapp.exe". As you can see, the backslashes are escaped, causing the string to be less readable. You can improve the string with a verbatim literal, like this: @"c:\topdir\subdir\subdir\myapp.exe".

VARIABLES

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C# has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.The basic value types provided in C# can be categorized as:

Type	Example
Integral types	sbyte, byte, short, ushort, int, uint, long, ulong and char
Floating point types	float and double
Decimal types	decimal
Boolean types	true or false values, as assigned
Nullable types	Nullable data types

C# also allows defining other value types of variable like **enum** and reference types of variables like **class**, which we will cover in subsequent chapters. For this chapter, let us study only basic variable types.

Variable Definition in C#

Syntax for variable definition in C# is:
<data_type> <variable_list>;
Here, data_type must be a valid C# data type including char, int, float, double, or any user-defined data type, etc., and variable_list may consist of one or more identifier names separated by commas.
Some valid variable definitions are shown here:
int i, j, k;
char c, ch;
float f, salary;
double d;
You can initialize a variable at the time of definition as:
int i = 100;

Variable Initialization in C#

Variables are initialized (assigned a value) with an equal sign followed by a constant expression. The general form of initialization is:
variable_name = value;
Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as:
<data_type> <variable_name> = value;
Some examples are:
int d = 3, f = 5;   /* initializing d and f. */
byte z = 22;   /* initializes z. */
double pi = 3.14159; /* declares an approximation of pi. */
char x = 'x';   /* the variable x has the value 'x'. */

It is a good programming practice to initialize variables properly, otherwise sometimes program would produce unexpected result.

Try the following example which makes use of various types of variables:

```csharp
namespace VariableDefinition
{
    class Program
    {
        static void Main(string[] args)
        {
            short a;
            int b ;
            double c;

            /* actual initialization */
            a = 10;
            b = 20;
            c = a + b;
            Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

a = 10, b = 20, c = 30

Accepting Values from User

The **Console** class in the **System** namespace provides a function **ReadLine()** for accepting input from the user and store it into a variable.

For example,

```csharp
int num;
num = Convert.ToInt32(Console.ReadLine());
```

The function **Convert.ToInt32()** converts the data entered by the user to int data type, because **Console.ReadLine()** accepts the data in string format.

Lvalues and Rvalues in C#:

There are two kinds of expressions in C#:

lvalue: An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment.

rvalue: An expression that is an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and can not appear on the left-hand side. Following is a valid statement:

int g = 20;

But following is not a valid statement and would generate compile-time error:

10 = 20;

2.Write about operators in c#
Operators in C#

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C# is rich in built-in operators and provides the following type of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

This tutorial will explain the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

Arithmetic Operators

Following table shows all the arithmetic operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20 then:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator increases integer value by one	A++ will give 11
--	Decrement operator decreases integer value by one	A-- will give 9

EXAMPLE

```
using System;
namespace OperatorsAppl
{
  class Program
  {
    static void Main(string[] args)
    {
      int a = 21;
      int b = 10;
      int c;
      c = a + b;
      Console.WriteLine("Line 1 - Value of c is {0}", c);
      c = a - b;
      Console.WriteLine("Line 2 - Value of c is {0}", c);
      c = a * b;
      Console.WriteLine("Line 3 - Value of c is {0}", c);
      c = a / b;
      Console.WriteLine("Line 4 - Value of c is {0}", c);
      c = a % b;
      Console.WriteLine("Line 5 - Value of c is {0}", c);
      c = a++;
      Console.WriteLine("Line 6 - Value of c is {0}", c);
      c = a--;
      Console.WriteLine("Line 7 - Value of c is {0}", c);
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result:

OUTPUT

Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of c is 2
Line 5 - Value of c is 1
Line 6 - Value of c is 21
Line 7 - Value of c is 22

Relational Operators

Following table shows all the relational operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20, then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

EXAMPLE
```
using System;

class Program
{
  static void Main(string[] args)
  {
    int a = 21;
    int b = 10;

    if (a == b)
    {
      Console.WriteLine("Line 1 - a is equal to b");
    }
```

```csharp
      else
      {
         Console.WriteLine("Line 1 - a is not equal to b");
      }
      if (a < b)
      {
         Console.WriteLine("Line 2 - a is less than b");
      }
      else
      {
         Console.WriteLine("Line 2 - a is not less than b");
      }
      if (a > b)
      {
         Console.WriteLine("Line 3 - a is greater than b");
      }
      else
      {
         Console.WriteLine("Line 3 - a is not greater than b");
      }
      /* Lets change value of a and b */
      a = 5;
      b = 20;
      if (a <= b)
      {
         Console.WriteLine("Line 4 - a is either less than or equal to  b");
      }
      if (b >= a)
      {
         Console.WriteLine("Line 5-b is either greater than or equal to b");
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:
OUTPUT
Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either less than or equal to b
Line 5 - b is either greater than or equal to b

Logical Operators

Following table shows all the logical operators supported by C#. Assume variable **A** holds Boolean value true and variable **B** holds Boolean value false, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

EXAMPLE

```
using System;
namespace OperatorsAppl
{
   class Program
   {
      static void Main(string[] args)
      {
         bool a = true;
         bool b = true;

         if (a && b)
         {
            Console.WriteLine("Line 1 - Condition is true");
         }
         if (a || b)
         {
            Console.WriteLine("Line 2 - Condition is true");
         }
         /* lets change the value of  a and b */
         a = false;
         b = true;
         if (a && b)
         {
            Console.WriteLine("Line 3 - Condition is true");
         }
         else
```

```
        {
            Console.WriteLine("Line 3 - Condition is not true");
        }
        if (!(a && b))
        {
            Console.WriteLine("Line 4 - Condition is true");
        }
        Console.ReadLine();
    }
  }
}
```
When the above code is compiled and executed, it produces the following result:

OUTPUT

Line 1 - Condition is true
Line 2 - Condition is true
Line 3 - Condition is not true
Line 4 - Condition is true

Bitwise Operators

Bitwise operator works on bits and perform bit by bit operation. The truth tables for &, |, and ^ are as follows:

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; now in binary format they will be as follows:

A = 0011 1100
B = 0000 1101

A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
~A = 1100 0011

The Bitwise operators supported by C# are listed in the following table. Assume variable A holds 60 and variable B holds 13 then:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12. which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61, which is 1100 0011 in 2's complement due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15, which is 0000 1111

EXAMPLE

```csharp
using System;
namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 60;            /* 60 = 0011 1100 */
            int b = 13;            /* 13 = 0000 1101 */
            int c = 0;

            c = a & b;         /* 12 = 0000 1100 */
            Console.WriteLine("Line 1 - Value of c is {0}", c );

            c = a | b;         /* 61 = 0011 1101 */
            Console.WriteLine("Line 2 - Value of c is {0}", c);

            c = a ^ b;         /* 49 = 0011 0001 */
            Console.WriteLine("Line 3 - Value of c is {0}", c);

            c = ~a;            /*-61 = 1100 0011 */
            Console.WriteLine("Line 4 - Value of c is {0}", c);

            c = a << 2;     /* 240 = 1111 0000 */
            Console.WriteLine("Line 5 - Value of c is {0}", c);

            c = a >> 2;     /* 15 = 0000 1111 */
            Console.WriteLine("Line 6 - Value of c is {0}", c);
            Console.ReadLine();
        }
    }
}
```
When the above code is compiled and executed, it produces the following result:

OUTPUT

Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15

Assignment Operators

There are following assignment operators supported by C#:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

EXAMPLE
using System;

namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)

```csharp
        {
            int a = 21;
            int c;

            c = a;
            Console.WriteLine("Line 1 - =  Value of c = {0}", c);

            c += a;
            Console.WriteLine("Line 2 - += Value of c = {0}", c);

            c -= a;
            Console.WriteLine("Line 3 - -=  Value of c = {0}", c);

            c *= a;
            Console.WriteLine("Line 4 - *=  Value of c = {0}", c);

            c /= a;
            Console.WriteLine("Line 5 - /=  Value of c = {0}", c);

            c = 200;
            c %= a;
            Console.WriteLine("Line 6 - %=  Value of c = {0}", c);

            c <<= 2;
            Console.WriteLine("Line 7 - <<=  Value of c = {0}", c);

            c >>= 2;
            Console.WriteLine("Line 8 - >>=  Value of c = {0}", c);

            c &= 2;
            Console.WriteLine("Line 9 - &=  Value of c = {0}", c);

            c ^= 2;
            Console.WriteLine("Line 10 - ^=  Value of c = {0}", c);

            c |= 2;
            Console.WriteLine("Line 11 - |=  Value of c = {0}", c);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

OUTPUT

Line 1 - =   Value of c = 21
Line 2 - +=   Value of c = 42
Line 3 - -=   Value of c = 21

Line 4 - *=   Value of c = 441
Line 5 - /=   Value of c = 21
Line 6 - %=   Value of c = 11
Line 7 - <<=   Value of c = 44
Line 8 - >>=   Value of c = 11
Line 9 - &=   Value of c = 2
Line 10 - ^=   Value of c = 0
Line 11 - |=   Value of c = 2

Misc Operators

There are few other important operators including **sizeof, typeof** and **? :** supported by C#.

Operator	Description	Example
sizeof()	Returns the size of a data type.	sizeof(int), will return 4.
typeof()	Returns the type of a class.	typeof(StreamReader);
&	Returns the address of an variable.	&a; will give actual address of the variable.
*	Pointer to a variable.	*a; will pointer to a variable.
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y
is	Determines whether an object is of a certain type.	If(Ford is Car) // checks if Ford is an object of the Car class.
as	Cast without raising an exception if the cast fails.	Object obj = new StringReader("Hello"); StringReader r = obj as StringReader;

EXAMPLE

```csharp
using System;

namespace OperatorsAppl
{

  class Program
  {
    static void Main(string[] args)
    {

      /* example of sizeof operator */
      Console.WriteLine("The size of int is {0}", sizeof(int));
      Console.WriteLine("The size of short is {0}", sizeof(short));
      Console.WriteLine("The size of double is {0}", sizeof(double));

      /* example of ternary operator */
      int a, b;
      a = 10;
      b = (a == 1) ? 20 : 30;
      Console.WriteLine("Value of b is {0}", b);

      b = (a == 10) ? 20 : 30;
      Console.WriteLine("Value of b is {0}", b);
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result:

OUTPUT

The size of int is 4
The size of short is 2
The size of double is 8
Value of b is 30
Value of b is 20

3. What are the Control Statements in C#?Explain with appropriate examples.

(OR)
Explain about the Control Statements with example

Control Statements –Selection

There are three categories of program control statements: *selection* statements, which are the **if** and the **switch** *iteration* statements, which consist of the **for**, **while**, **do-while**, and **foreach** loops; and *jump* statements, which include **break**, **continue**, **goto**, **return**, and **throw**

- ✓ If Statement
- ✓ Switch Statement
- ✓ Break Statement
- ✓ Goto Statement

The if Statement

An if statement allows you to take different paths of logic, depending on a given condition. When the condition evaluates to a boolean true, a block of code for that true condition will execute. You have the option of a single if statement, multiple else if statements, and an optional else statement.

```
using System;

class IfSelect
{    public static void Main()
  {      string myInput;

     int myInt;

    Console.Write("Please enter a number: ");

     myInput = Console.ReadLine();

     myInt = Int32.Parse(myInput);

   // Single Decision and Action with braces

  if (myInt > 0)
```

```csharp
        {   Console.WriteLine("Your number {0} is greater than zero.", myInt);        }

    // Single Decision and Action without brackets

        if (myInt < 0)

            Console.WriteLine("Your number {0} is less than zero.", myInt);

    // Either/Or Decision

        if (myInt != 0)

        { Console.WriteLine("Your number {0} is not equal to zero.", myInt);        }        else

        {        Console.WriteLine("Your number {0} is equal to zero.", myInt);

    } // Multiple Case Decision

        if (myInt < 0 || myInt == 0)

    {    Console.WriteLine("Your number {0} is less than or equal to zero.", myInt);

    }        else if (myInt > 0 && myInt <= 10)

        {            Console.WriteLine("Your number {0} is in the range from 1 to 10.", myInt);

    }        else if (myInt > 10 && myInt <= 20)

    {            Console.WriteLine("Your number {0} is in the range from 11 to 20.", myInt);

    }        else if (myInt > 20 && myInt <= 30)

        {            Console.WriteLine("Your number {0} is in the range from 21 to 30.", myInt);

    }        else

        {    Console.WriteLine("Your number {0} is greater than 30.", myInt);

        }

    }

}
```

The switch Statement

Another form of selection statement is the switch statement, which executes a set of logic depending on the value of a given parameter. The types of the values a switch statement operates on can be booleans, enums, integral types, and strings.

```
using System;

class SwitchSelect

{

  public static void Main()

  {

    string myInput;

    int myInt;

    begin:

    Console.Write("Please enter a number between 1 and 3: ");

    myInput = Console.ReadLine();

    myInt = Int32.Parse(myInput);
 // switch with integer type

    switch (myInt)

    {   case 1:

        Console.WriteLine("Your number is {0}.", myInt);

        break;

case 2:   Console.WriteLine("Your number is {0}.", myInt);

        break;

      case 3:
```

```csharp
            Console.WriteLine("Your number is {0}.", myInt);

            break;

        default:

            Console.WriteLine("Your number {0} is not between 1 and 3.", myInt);

            break;        }

decide:

Console.Write("Type \"continue\" to go on or \"quit\" to stop: ");

    myInput = Console.ReadLine();

// switch with string type

    switch (myInput)

    {     case "continue":

            goto begin;

        case "quit":

            Console.WriteLine("Bye.");

            break;

        default:

            Console.WriteLine("Your input {0} is incorrect.", myInput);

            goto decide;

    }    }      }
```

2.1.5 Control Statements - Loops

- While loop.

- Do loop.

- For loop.

- Foreach loop.

The while Loop

A while loop will check a condition and then continues to execute a block of code as long as the condition evaluates to a boolean value of true. Its syntax is as follows: while (<boolean expression>) { <statements> }. The statements can be any valid C# statements. The boolean expression is evaluated before any code in the following block has executed. When the boolean expression evaluates to true, the statements will execute. Once the statements have executed, control returns to the beginning of the while loop to check the boolean expression again.

When the boolean expression evaluates to false, the while loop statements are skipped and execution begins after the closing brace of that block of code. Before entering the loop, ensure that variables evaluated in the loop condition are set to an initial state. During execution, make sure you update variables associated with the boolean expression so that the loop will end when you want it to.

```
using System;

class WhileLoop

{

  public static void Main()

  {      int myInt = 0;

  while (myInt < 10)

      {   Console.Write("{0} ", myInt);

        myInt++;

      }   Console.WriteLine();

  } }
```

The do Loop

A do loop is similar to the while loop, except that it checks its condition at the end of the loop. This means that the do loop is guaranteed to execute at least one time. On the other hand, a while loop evaluates its boolean expression at the beginning and there is generally no guarantee that the statements inside the loop will be executed, unless you program the code to explicitly do so. One reason you may want to use a do loop instead of a while loop is to present a message.

```
using System;

class DoLoop

{    public static void Main()

   {      string myChoice;

 do

     {     // Print A Menu

       Console.WriteLine("My Address Book\n");

       Console.WriteLine("A - Add New Address");

       Console.WriteLine("D - Delete Address");

       Console.WriteLine("M - Modify Address");

       Console.WriteLine("V - View Addresses");

       Console.WriteLine("Q - Quit\n");

       Console.WriteLine("Choice (A,D,M,V,or Q): ");

       // Retrieve the user's choice

       myChoice = Console.ReadLine();

       // Make a decision based on the user's choice

       switch(myChoice)

       {  case "A":

          case "a":
```

```csharp
            Console.WriteLine("You wish to add an address.");

            break;

        case "D":

        case "d":

            Console.WriteLine("You wish to delete an address.");

            break;

        case "M":

        case "m":

            Console.WriteLine("You wish to modify an address.");

            break;

        case "V":

        case "v":

            Console.WriteLine("You wish to view the address list.");

            break;

        case "Q":case "q":

            Console.WriteLine("Bye.");

            break;

        default:

            Console.WriteLine("{0} is not a valid choice", myChoice);

            break;

    }

// Pause to allow the user to see the results
```

```
        Console.Write("press Enter key to continue...");

        Console.ReadLine();

        Console.WriteLine();

    } while (myChoice != "Q" && myChoice != "q"); // Keep going until the user wants to
quit

    }    }
```

The for Loop

A for loop works like a while loop, except that the syntax of the for loop includes initialization and condition modification. for loops are appropriate when you know exactly how many times you want to perform the statements within the loop. The contents within the for loop parentheses hold three sections separated by semicolons (<initializer list>; <boolean expression>; <iterator list>) { <statements> }. The initializer list is a comma separated list of expressions. These expressions are evaluated only once during the lifetime of the for loop. This is a one-time operation, before loop execution. This section is commonly used to initialize an integer to be used as a counter.

Once the initializer list has been evaluated, the for loop gives control to its second section, the boolean expression. There is only one boolean expression, but it can be as complicated as you like as long as the result evaluates to true or false. The boolean expression is commonly used to verify the status of a counter variable.

When the boolean expression evaluates to true, the statements within the curly braces of the for loop are executed. After executing for loop statements, control moves to the top of loop and executes the iterator list, which is normally used to increment or decrement a counter. The iterator list can contain a comma separated list of statements, but is generally only one statement.

```
using System;

class ForLoop

{

   public static void Main()

   {    for (int i=0; i < 20; i++)
```

```
{   if (i == 10)

        break;

    if (i % 2 == 0)

        continue;

Console.Write("{0} ", i);

    }   Console.WriteLine();

  }

}
```

The foreach Loop

A foreach loop is used to iterate through the items in a list. It operates on arrays or collections such as ArrayList, which can be found in the System.Collections namespace. The syntax of a foreach loop is foreach (<type> <iteration variable> in <list>) { <statements> }. The type is the type of item contained in the list. For example, if the type of the list was int[] then the type would be int.

The iteration variable is an identifier that you choose, which could be anything but should be meaningful. For example, if the list contained an array of people's ages, then a meaningful name for item name would be age. The in keyword is required.

```
using System;

class ForEachLoop

  {

    public static void Main()

    {

      string[] names = {"Cheryl", "Joe", "Matt", "Robert"};

      foreach (string person in names)

       {

         Console.WriteLine("{0} ", person);
```

```
            }

        }
```

Loop Control Statements:

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C# provides the following control statements. Click the following links to check their details.

Control Statement	Description
break statement	Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

EXAMPLE OF BREAK

```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            /* local variable definition */
            int a = 10;

            /* while loop execution */
            while (a < 20)
            {
                Console.WriteLine("value of a: {0}", a);
                a++;
                if (a > 15)
                {
                    /* terminate the loop using break statement */
                    break;
                }
```

```
        }
        Console.ReadLine();
      }
    }
}
```

When the above code is compiled and executed, it produces the following result:

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15

EXAMPLE OF CONTINUE

```
using System;

namespace Loops
{

  class Program
  {
    static void Main(string[] args)
    {
      /* local variable definition */
      int a = 10;

      /* do loop execution */
      do
      {
        if (a == 15)
        {
          /* skip the iteration */
          a = a + 1;
          continue;
        }
        Console.WriteLine("value of a: {0}", a);
        a++;

      } while (a < 20);
```

```
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 16

value of a: 17

value of a: 18

value of a: 19

GOTO STATEMENT

The goto statement is a jump statement that controls the execution of the program to another segment of the same program. You create label at anywhere in program then can pass the execution control via the goto statements.

```csharp
using System;

namespace goto_statement
{
  class Program
  {
    static void Main(string[] args)
    {
      string name;

      label: //creating label with colon(:)

      Console.WriteLine("Enter your name:");
      name = Console.ReadLine();

      Console.WriteLine("Welcome  {0}", name);

      goto label; //jump to label statement
    }
  }
}
```

note: To terminate the program press Ctrl+C

Enter your name :
Steven
Welcome Steven
Enter your name :
Clark
Welcome Clark
Enter your name :
Steven Clark
Welcome Steven Clark
Enter your name :

POLYMORPHISM

- Polymorphism is one of the primary characteristics of Object Oriented Programming.
- "Poly" means "many" and "morph" means "form".
- Thus polymorphism refers to being able to use many form of a type without regard to the details.
- **Types of Polymorphism**
- Compile Time Polymorphism.
- Runtime Polymorphism.

Compile Time Polymorphism

- Compile time Polymorphism also known as Method Overloading.
- Method Overloading means having two or more methods with the same name but with different parameters.
- It is also called as static polymorphism.
- Which method is to be called is decided at the compile time only.
- In static Polymorphism decision is taken at the Compile time.

EXAMPLE

```
using System;
namespace PolymorphismApplication
{
  class Printdata
  {
    void print(int i)
    {
      Console.WriteLine("Printing int: {0}", i );
    }

    void print(double f)
    {
      Console.WriteLine("Printing float: {0}" , f);
    }

    void print(string s)
    {
      Console.WriteLine("Printing string: {0}", s);
    }

Void print(int a,float b)
{
        Console.WriteLine("Printing int:{0}" , a);
        Console.WriteLine("Printing float:{0}" , b);

}


    Public static void Main(string[] args)
    {
      Printdata p = new Printdata();
      // Call print to print integer
      p.print(5);
      // Call print to print float
      p.print(500.263);
      // Call print to print string
      p.print("Hello C++");
     p.print(12.12.0);
      Console.ReadKey();

    }
  }
}
```

When the above code is compiled and executed, it produces the following result:

```
Printing int: 5
Printing float: 500.263
Printing int: 12
Printing float: 12.0
Printing string: Hello C++
```

Runtime Polymorphism

- Run Time Polymorphism also known as Method Overriding.

- Method Overriding means having two or more methods with the same name, same parameters but with different implementation.

- It is also called as Dynamic Polymorphism.

- In this process, an overridden method is called through reference variable of a superclass; the determination of the method to be called is based on the object being referred to by reference variable.

- The parent class uses the same virtual keyword. Every class that overrides the virtual method will use the override keyword.

EXAMPLE

```csharp
using System;
class Parent
{
    public virtual void show()

    {
        Console.WriteLine("Parent");
    }
}

class Child : Parent
{
    public override void show()

    {
        Console.WriteLine("Child");
    }
}
class grandchild : Parent
{
    public override void show()
```

```csharp
        {
            Console.WriteLine("grandChild");
        }
    }
}
class Test
{
    static void Main()
    {
        Parent p = new Parent();
        Parent c = new Child();
        Parent g = new grandchild();

        c.show();
        p.show();
        g.show();
        Console.ReadLine();
    }
}
```

Output
Child
Parent
grandChild

INTERFACES
- An interface is defined as a syntactical contract that all the classes inheriting the interface should follow.
- The interface defines the **'what'** part of the syntactical contract and the deriving classes define the **'how'** part of the syntactical contract.
- Interfaces define properties, methods and events, which are the members of the interface. Interfaces contain only the declaration of the members.
- It is the responsibility of the deriving class to define the members. It often helps in providing a standard structure that the deriving classes would follow.

Declaring Interfaces

Interfaces are declared using the interface keyword. It is similar to class declaration. Interface statements are public by default. Following is an example of an interface declaration:

```csharp
public interface ITransactions
{
    // interface members
    void showTransaction();
}
```

```csharp
    double getAmount();
}
```

EXAMPLE

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

namespace InterfaceApplication
{

    public interface ITransactions
    {
        // interface members
        void showTransaction();
        double getAmount();
    }
    public class Transaction : ITransactions
    {
        private string tCode;
        private string date;
        private double amount;
        public Transaction()
        {
            tCode = " ";
            date = " ";
            amount = 0.0;
        }
        public Transaction(string c, string d, double a)
        {
            tCode = c;
            date = d;
            amount = a;
        }
        public double getAmount()
        {
            return amount;
        }
        public void showTransaction()
        {
            Console.WriteLine("Transaction: {0}", tCode);
            Console.WriteLine("Date: {0}", date);
            Console.WriteLine("Amount: {0}", getAmount());
```

```
    }

  }
  class Tester
  {
    static void Main(string[] args)
    {
      Transaction t1 = new Transaction("001", "8/10/2012", 78900.00);
      Transaction t2 = new Transaction("002", "9/10/2012", 451900.00);
      t1.showTransaction();
      t2.showTransaction();
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result:

```
Transaction: 001
Date: 8/10/2012
Amount: 78900
Transaction: 002
Date: 9/10/2012
Amount: 451900
```

HANDLING EXCEPTION

An exception is a problem that arises during the execution of a program. A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: **try**, **catch**, **finally** and **throw**.

- **try**: A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.
- **catch**: A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- **finally**: The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
- **throw**: A program throws an exception when a problem shows up. This is done using a throw keyword.

Syntax

Assuming a block will raise and exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
   // statements causing exception
}
catch( ExceptionName e1 )
{
   // error handling code
}
catch( ExceptionName e2 )
{
   // error handling code
}
catch( ExceptionName eN )
{
   // error handling code
}
finally
{
   // statements to be executed
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Exception Classes in C#

C# exceptions are represented by classes. The exception classes in C# are mainly directly or indirectly derived from the **System.Exception** class. Some of the exception classes derived from the System.Exception class are the **System.ApplicationException** and **System.SystemException** classes.
The **System.ApplicationException** class supports exceptions generated by application programs. So the exceptions defined by the programmers should derive from this class.
The **System.SystemException** class is the base class for all predefined system exception.

The following table provides some of the predefined exception classes derived from the Sytem.SystemException class:

Exception Class	Description
System.IO.IOException	Handles I/O errors.
System.IndexOutOfRangeException	Handles errors generated when a method

	refers to an array index out of range.
System.ArrayTypeMismatchException	Handles errors generated when type is mismatched with the array type.
System.NullReferenceException	Handles errors generated from deferencing a null object.
System.DivideByZeroException	Handles errors generated from dividing a dividend with zero.
System.InvalidCastException	Handles errors generated during typecasting.
System.OutOfMemoryException	Handles errors generated from insufficient free memory.
System.StackOverflowException	Handles errors generated from stack overflow.

Handling Exceptions

C# provides a structured solution to the exception handling problems in the form of try and catch blocks. Using these blocks the core program statements are separated from the error-handling statements.

These error handling blocks are implemented using the **try**, **catch** and **finally** keywords.

EXAMPLE

```csharp
using System;
namespace ErrorHandlingApplication
{
   class DivNumbers
   {
      int result;
      DivNumbers()
      {
         result = 0;
      }
      public void division(int num1, int num2)
      {
         try
         {
            result = num1 / num2;
         }
         catch (DivideByZeroException e)
```

```
        {
            Console.WriteLine("Exception caught: {0}", e);
        }
        finally
        {
            Console.WriteLine("Result: {0}", result);
        }

    }
    static void Main(string[] args)
    {
        DivNumbers d = new DivNumbers();
        d.division(25, 0);
        Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result:

```
Exception caught: System.DivideByZeroException: Attempted to divide by zero.
at ...
Result: 0
```

DELEGATES

C# delegates are similar to pointers to functions, in C or C++. A **delegate** is a reference type variable that holds the reference to a method. The reference can be changed at runtime. Delegates are especially used for implementing events and the call-back methods. All delegates are implicitly derived from the **System.Delegate** class.

Declaring Delegates

Delegate declaration determines the methods that can be referenced by the delegate. A delegate can refer to a method, which have the same signature as that of the delegate.

For example, consider a delegate:

```
public delegate int MyDelegate (string s);
```

The preceding delegate can be used to reference any method that has a single *string* parameter and returns an *int* type variable.

Syntax for delegate declaration is:

delegate <return type> <delegate-name>

Instantiating Delegates

Once a delegate type has been declared, a delegate object must be created with the **new** keyword and be associated with a particular method. When creating a delegate, the argument passed to the **new**expression is written like a method call, but without the arguments to the method. For example:

```
public delegate void printString(string s);
...
printString ps1 = new printString(WriteToScreen);
printString ps2 = new printString(WriteToFile);
```

Following example demonstrates declaration, instantiation and use of a delegate that can be used to reference methods that take an integer parameter and returns an integer value.

```
using System;

delegate int NumberChanger(int n);
namespace DelegateAppl
{
   class TestDelegate
   {
      static int num = 10;
      public static int AddNum(int p)
      {
         num += p;
         return num;
      }

      public static int MultNum(int q)
      {
         num *= q;
         return num;
      }
      public static int getNum()
      {
         return num;
      }

      static void Main(string[] args)
      {
         //create delegate instances
         NumberChanger nc1 = new NumberChanger(AddNum);
         NumberChanger nc2 = new NumberChanger(MultNum);
         //calling the methods using the delegate objects
         nc1(25);
```

```
        Console.WriteLine("Value of Num: {0}", getNum());
        nc2(5);
        Console.WriteLine("Value of Num: {0}", getNum());
        Console.ReadKey();
      }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of Num: 35
Value of Num: 175
```

1. INDEXERS AND PROPERTIES

PROPERTIES

Properties are named members of classes, structures, and interfaces. Member variables or methods in a class or structures are called **Fields**. Properties are an extension of fields and are accessed using the same syntax. They use **accessors** through which the values of the private fields can be read, written or manipulated.

Properties do not name the storage locations. Instead, they have **accessors** that read, write, or compute their values.

For example, let us have a class named Student, with private fields for age, name and code. We cannot directly access these fields from outside the class scope, but we can have properties for accessing these private fields.

The **accessor** of a property contains the executable statements that helps in getting (reading or computing) or setting (writing) the property. The accessor declarations can contain a get accessor, a set accessor, or both.

General form of property

Type name

{

  Get

    {

    }

Set

    {

    }

}

Type-specifies the data type of the property

Name specifies the property name

Get-gets the value of the variable

Set-sets the value of the variable and it has implicit variable called value that contains the value to be assigned to the property.

EXAMPLE

```csharp
using System;
class simpleprop
{
    private int prop;
    public simpleprop()
    {
        prop = 0;
    }
    public int myprop
    {
        get
        {
            return prop;
        }
        set
        {
            if (value >= 0)
            {
                prop = value;
            }
        }
    }
}
    class propertydemo
    {
```

```
    public static void Main()
    {
        simpleprop ob=new simpleprop();
        Console.WriteLine("ORIGINAL VALUE OF OB.MYPROP"+ob.myprop);
        ob.myprop=100;
        Console.WriteLine("value of prop"+ob.myprop);
        ob.myprop=-10;
        Console.WriteLine("value of prop after assigning negative value -10"+ob.myprop);
    }
}
```

OUTPUT

ORIGINAL VALUE OF OB.MYPROP:0

Value of prop: 100

Value of prop after assigning negative value -10: 100

INDEXERS

An **indexer** allows an object to be indexed like an array. When you define an indexer for a class, this class behaves like a **virtual array**. You can then access the instance of this class using the array access operator ([ ]).

A one dimensional indexer has the following syntax:

```
element-type this[int index]
{
  // The get accessor.
  get
  {
    // return the value specified by index
  }

  // The set accessor.
  set
  {
    // set the value specified by index
  }
}
```

Here, element-type is the base type of the indexer. Thus, each element accessed by the indexer will be of type element-type. The element type corresponds to the base type of an array. The parameter index receives the index of the element being accessed. Except that it does not have a return type or parameter declarations. The accessors are automatically called when the indexer is used, and both accessors receive index as a parameter. If the indexer is on the left side of an assignment statement, then the set accessor is called, and the element specified by index must be set. Otherwise, the get accessor is called, and the value associated with index must be returned. The set accessor also receives an implicit parameter called value, which contains the value being assigned to the specified index.

```csharp
using System;
namespace IndexerApplication
{
    class IndexedNames
    {
        private string[] namelist = new string[size];
        static public int size = 10;
        public IndexedNames()
        {
            for (int i = 0; i < size; i++)
                namelist[i] = "N. A.";
        }
        public string this[int index]
        {
            get
            {
                string tmp;

                if (index >= 0 && index <= size - 1)
                {
                    tmp = namelist[index];
                }
                else
                {
                    tmp = "";
                }

                return (tmp);
            }
            set
            {
                if (index >= 0 && index <= size - 1)
                {
                    namelist[index] = value;
```

```
                }
            }
        }

        static void Main(string[] args)
        {
            IndexedNames names = new IndexedNames();
            names[0] = "Zara";
            names[1] = "Riz";
            names[2] = "Nuha";
            names[3] = "Asif";
            names[4] = "Davinder";
            names[5] = "Sunil";
            names[6] = "Rubic";
            for (int i = 0; i < IndexedNames.size; i++)
            {
                Console.WriteLine(names[i]);
            }
            Console.ReadKey();
        }
    }
}
```
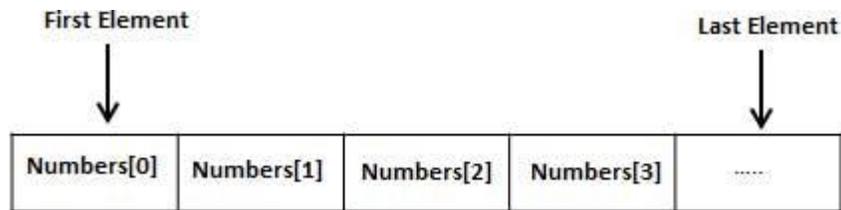
OUTPUT

Zara
Riz
Nuha
Asif
Davinder
Sunil
Rubic
N. A.
N. A.
N. A.

ARRAYS

- An array is a collection of variables of same type that are referred by a common name.

- All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Example:

We can use an array to hold a record of the dialy high temperature for a month,a list of stock prices, employee details etc.

ONE DIMENSIONAL ARRAY

A one dimensional array is a list of related variables.

General form:

Type [] array-name=new type [size];

- Type declares the data type of an array.

- In c# the square bracket comes after type. Single square bracket indicates one-dimensional array.

- The numbers of array elements are determined by size.

- Since arrays are implemented as objects, the creation of an array is a two-step process.

- **Step 1:** First declare array reference variable.

    Example: int [] sample;

- **Step 2:** Allocate memory for the array, assigning a reference to that memory to the array variable by using new operator.

    Sample=new int [10];

ASSIGNING VALUES TO ARRAYS

1) We can assign values to individual array elements, by using the index number, like:

    double[] balance = new double[10];
    balance[0] = 4500.0;

2) We can assign values to the array at the time of declaration, like:

double[] balance = { 2340.0, 4523.69, 3421.0};

3) We can also create and initialize an array, like:

int [] marks = new int[5] { 99, 98, 92, 97, 95};

4) In the preceding case, We may also omit the size of the array, like:

int [] marks = new int[] { 99, 98, 92, 97, 95};

5) We can also copy an array variable into another target array variable. In that case, both the target and source would point to the same memory location:

int [] marks = new int[] { 99, 98, 92, 97, 95};
int[] score = marks;

EXAMPLE OF SINGLE-DIMENSIONAL ARRAY

```
using System;
namespace ArrayApplication
{
  class MyArray
  {
    static void Main(string[] args)
    {
      int []  n = new int[10]; /* n is an array of 10 integers */
      int i,j;


      /* initialize elements of array n */
      for ( i = 0; i < 10; i++ )
      {
        n[ i ] = i + 100;
      }

      /* output each array element's value */
      for (j = 0; j < 10; j++ )
      {
        Console.WriteLine("Element[{0}] = {1}", j, n[j]);
      }
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result:

Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109

1.4 MULTI-DIMENSIONAL ARRAY

A multi-dimensional array is an array that has two or more dimensions and an individual element is accessed through the combination of two or more indices.

TWO-DIMENSIONAL ARRAYS

- The simplest form of the multidimensional array is the 2-dimensional array. A 2-dimensional array is, in essence, a list of one-dimensional arrays.
- A 2-dimensional array can be thought of as a table, which will have x number of rows and y number of columns. Following is a 2-dimentional array, which contains 3 rows and 4 columns:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

- Thus, every element in array a is identified by an element name of the form a[i , j], where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

DECLARATION

- To declare a two-dimensional integer array of size 10, 20 (i.e.) 10 rows and 20 columns,

  Int [,] array_name = new int [10, 20];

- Note that the two-dimension arrays are separated from each other by a comma.

INITIALIZING TWO-DIMENSIONAL ARRAYS

- Two dimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int [,] a = int [3,4] = {
{0, 1, 2, 3} ,  /*  initializers for row indexed by 0 */
{4, 5, 6, 7} ,  /*  initializers for row indexed by 1 */
{8, 9, 10, 11}  /*  initializers for row indexed by 2 */
};
```

DECLARING ARRAY OF THREE OR MORE DIMENSIONS

Type [,…,] array_name = new type[size1,size2,….size N];

EXAMPLE OF MULTI-DIMENSIONAL ARRAYS

```
using System;
  class Program
  {
    static void Main(string[] args)
    {
      int t, i;
      int [,] table = new int [3,4];
      for (t = 0; t < 3; ++t)
      {
        for (i = 0; i < 4; i++)
        {
          table[t, i] = (t * 4) + i + 1;
          Console.WriteLine(table[t, i] + " ");
        }
      }

    }
  }
```
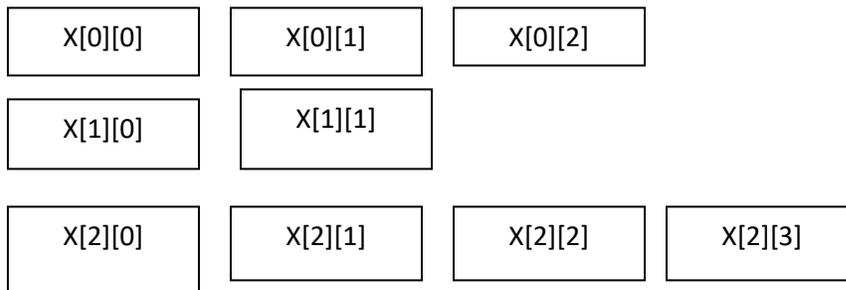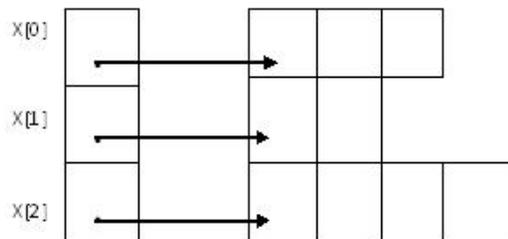
OUTPUT (CONCEPTUAL VIEW)

	0	1	2	3	
	1	2	3	4	0
	5	6	7	8	1
	9	10	11	12	2

JAGGED ARRAYS

A Jagged array is an array of arrays.

A jagged array is an array in which the length of each array can differ. Thus a jagged array can be used to create a table in which the lengths of the rows are not the same.

int [ ] [ ] x = new int [3] [ ];
x[0] = new int [3];
x[1] =new int [2];
x[2] =new int [4];

| X[0][0] | X[0][1] | X[0][2] |

| X[1][0] | X[1][1] |

| X[2][0] | X[2][1] | X[2][2] | X[2][3] |

EXAMPLE
using System;

namespace ArrayApplication
{
   class MyArray
   {
      static void Main(string[] args)
      {
         /* a jagged array of 5 array of integers*/
         int[][] a = new int[][]{new int[]{0,0},new int[]{1,2},

```
new int[]{2,4},new int[]{ 3, 6 }, new int[]{ 4, 8 } };

int i, j;

/* output each array element's value */
for (i = 0; i < 5; i++)
{
    for (j = 0; j <; 2; j++)
    {
        Console.WriteLine("a[{0}][{1}] = {2}", i, j, a[i][j]);
    }
}
Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result:

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

THE FOR EACH-LOOP

- The for each loop is used to cycle through the elements of a collection.

- A collection is a group of objects.

- C# defines several types of collections, of which one is an array.

- The general form of for each loop:

  Foreach(type var_name in collection)

  {

Statements;

}

- The type specifies the data type and var_name specuies the name of the iteration variable.

- Collections specifies the array.

- The array type and iteration variable type should be same.

- The iteration variable is read only and hence we cannot change the contents of the array by assigning new value to iteration variable.

EXAMPLE

```
using System;
  class Program
  {
    static void Main(string[] args)
    {
      int sum = 0;
      int[] nums = new int[10];
      for (int i = 0; i < 10; i++)
      {
        nums[i] = i;
      }
      foreach (int x in nums)
      {
        Console.WriteLine("Vaue is:" + x);
        sum += x;
      }
      Console.WriteLine("Summation:" + sum);

    }

  }
```

OUTPUT

Value is : 0
Value is : 1
Value is : 2
Value is : 3
Value is : 4
Value is : 5
Value is : 6

Value is : 7
Value is : 8
Value is : 9
Sumation : 45

STRINGS

- From a day-to-day programming standpoint, one of the most important of C#'s data types is **string**.
- **string** defines and supports character strings. In many other programming languages a string is an array of characters.
- This is not the case with C#. In C#, strings are objects. Thus, **string** is a reference type. For example, in the statement
      Console.WriteLine("In C#, strings are objects.");
- The string "In C#, strings are objects." is automatically made into a **string** object by C#.

Constructing Strings

- The easiest way to construct a **string** is to use a string literal.
- For example, here **str** is a **string** reference variable that is assigned a reference to a string literal: string str = "C# strings are powerful.";
- In this case, **str** is initialized to the character sequence "C# strings are powerful."
- You can also create a **string** from a **char** array. For example:

```
using System;
class StringDemo {
public static void Main() {
char[] charray = {'A', ' ', 's', 't', 'r', 'i', 'n', 'g', '.' };
string str1 = new string(charray);
string str2 = "Another string.";
Console.WriteLine(str1);
Console.WriteLine(str2);
}
}
The output from the program is shown here:
A string.
Another string.
```

Operating on Strings

The **string** class contains several methods that operate on strings. Table 7-1 shows a few. The **string** type also includes the **Length** property, which contains the length of the string. To obtain the value of an individual character of a string, you simply use an index. For example:

string str = "test";
Console.WriteLine(str[0]);

This displays "t", the first character of "test". Like arrays, string indexes begin at zero. One important point, however, is that you cannot assign a new value to a character within a string using an index. An index can only be used to obtain a character.

Method	Description
static string Copy(string *str*)	Returns a copy of *str*.
int CompareTo(string *str*)	Returns less than zero if the invoking string is less than *str*, greater than zero if the invoking string is greater than *str*, and zero if the strings are equal.
int IndexOf(string *str*)	Searches the invoking string for the substring specified by *str*. Returns the index of the first match, or −1 on failure.
int LastIndexOf(string *str*)	Searches the invoking string for the substring specified by *str*. Returns the index of the last match, or −1 on failure.
string ToLower( )	Returns a lowercase version of the invoking string.
string ToUpper( )	Returns an uppercase version of the invoking string.

TABLE 7-1   Some Common String Handling Methods

Here is a program that demonstrates several string operations:

```
// Some string operations.
using System;
class StrOps {
public static void Main() {
string str1 =
"When it comes to .NET programming, C# is #1.";
string str2 = string.Copy(str1);
string str3 = "C# strings are powerful.";
string strUp, strLow;
int result, idx;
Console.WriteLine("str1: " + str1);
Console.WriteLine("Length of str1: " +
str1.Length);
// create upper- and lowercase versions of str1
strLow = str1.ToLower();
strUp = str1.ToUpper();
Console.WriteLine("Lowercase version of str1:\n " +
strLow);
Console.WriteLine("Uppercase version of str1:\n " +
strUp);
Console.WriteLine();
// display str1, one char at a time.
Console.WriteLine("Display str1, one char at a time.");
```

```
for(int i=0; i < str1.Length; i++)
Console.Write(str1[i]);
Console.WriteLine("\n");
// compare strings
if(str1 == str2)
Console.WriteLine("str1 == str2");
else
Console.WriteLine("str1 != str2");
if(str1 == str3)
Console.WriteLine("str1 == str3");
else
Console.WriteLine("str1 != str3");
result = str1.CompareTo(str3);
if(result == 0)
Console.WriteLine("str1 and str3 are equal");
else if(result < 0)
Console.WriteLine("str1 is less than str3");
else
Console.WriteLine("str1 is greater than str3");
Console.WriteLine();
// assign a new string to str2
str2 = "One Two Three One";
// search string
idx = str2.IndexOf("One");
Console.WriteLine("Index of first occurrence of One: " + idx);
idx = str2.LastIndexOf("One");
Console.WriteLine("Index of last occurrence of One: " + idx);
}
}
```

This program generates the following output:

str1: When it comes to .NET programming, C# is #1.
Length of str1: 44
Lowercase version of str1:
when it comes to .net programming, c# is #1.
Uppercase version of str1:
WHEN IT COMES TO .NET PROGRAMMING, C# IS #1.
Display str1, one char at a time.
When it comes to .NET programming, C# is #1.
str1 == str2
str1 != str3
str1 is greater than str3
Index of first occurrence of One: 0
Index of last occurrence of One: 14

You can concatenate (join together) two strings using the + operator. For example, this statement:
string str1 = "One";

string str2 = "Two";
string str3 = "Three";
string str4 = str1 + str2 + str3;
initializes **str4** with the string "OneTwoThree".

Arrays of Strings
Like any other data type, strings can be assembled into arrays. For example:

```
// Demonstrate string arrays.
using System;
class StringArrays {
public static void Main() {
string[] str = { "This", "is", "a", "test." };
Console.WriteLine("Original array: ");
for(int i=0; i < str.Length; i++)
Console.Write(str[i] + " ");
Console.WriteLine("\n");
// change a string
str[1] = "was";
str[3] = "test, too!";
Console.WriteLine("Modified array: ");
for(int i=0; i < str.Length; i++)
Console.Write(str[i] + " ");
}
}
```

Here is the output from this program:
Original array:
This is a test.
Modified array:
This was a test, too!

STRUCTURES

- Classes are reference types.

- This means that class objects are accessed through a reference.

- This differs from the value types, which are accessed directly.

- However, sometimes it would be useful to be able to access an object directly, in the

- way that value types are.

- One reason for this is efficiency. Accessing class objects through a reference adds overhead onto every access. It also consumes space.

SYNTAX

Structures are declared using the keyword struct and are syntactically similar to classes.

Here is the general form of a struct:

struct name : interfaces {

 // member declarations

}

- Structures do not support inheritance; structure members cannot be specified as abstract, virtual, or protected.

- A structure object can be created using new in the same way as a class object, but it is not required. When new is used, the specified constructor is called. When new is not used, the object is still created, but it is not initialized. Thus, you will need to perform any initialization manually


```
// Demonstrate a structure.

using System;

// Define a structure.

struct Book {

 public string author;

 public string title;

 public int copyright;

 public Book(string a, string t, int c) {

 author = a;

 title = t;

 copyright = c;
```

```csharp
  }
}

// Demonstrate Book structure.

class StructDemo {

 public static void Main() {

 Book book1 = new Book("Herb Schildt","C#: The Complete Reference", 2005); // explicit
constructor

 Book book2; // no constructorPART I

 Console.WriteLine(book1.title + " by " + book1.author + ", (c) " + book1.copyright);

 Console.WriteLine();

// Console.WriteLine(book2.title); // error, must initialize first

 Book2.title = "Red Storm Rising";

 Console.WriteLine(book3.title); // now OK

 }
}
```

The output from this program is shown here:

C#: The Complete Reference by Herb Schildt, (c) 2005

Red Storm Rising


```csharp
// Copy a struct.

using System;

// Define a structure.

struct MyStruct {

 public int x;

}328 Part I: The C# Language
```

```
// Demonstrate structure assignment.

class StructAssignment {

 public static void Main() {

 MyStruct a;

 MyStruct b;

 a.x = 10;

 b.x = 20;

 Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);

 a = b;

 b.x = 30;

 Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);

 }

}
```

The output is shown here:

a.x 10, b.x 20

a.x 20, b.x 30

As the output shows, after the assignment

a = b;

The structure variables a and b are still separate and distinct. That is, a does not refer to or

relate to b in any way other than containing a copy of b's value.

This would not be the case if a and b were class references. For example, here is the class version of the preceding

program:

// Copy a class.

using System;

```
// Define a structure.

class MyClass {

 public int x;

}

// Now show a class object assignment.

class ClassAssignment {

 public static void Main() {

 MyClass a = new MyClass();

 MyClass b = new MyClass();

 a.x = 10;

 b.x = 20;

 Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);

 a = b;

 b.x = 30;
```

PART I

Chapter 12: Interfaces, Structures, and Enumerations 329

```
 Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);

 }

}
```

The output from this version is shown here:

a.x 10, b.x 20

a.x 30, b.x 30

As you can see, after the assignment of b to a, both variables refer to same object—the one

originally referred to by b.

COLLECTIONS

Collection classes are specialized classes for data storage and retrieval.

These classes provide support for stacks, queues, lists, and hash tables. Most collection classes implement the same interfaces.

Collection classes serve various purposes, such as allocating memory dynamically to elements and accessing a list of items on the basis of an index etc. These classes create collections of objects of the Object class, which is the base class for all data types in C#.

Various Collection Classes and Their Usage

The following are the various commonly used classes of the **System.Collection** namespace. Click the following links to check their detail.

Class	Description and Useage
ArrayList	It represents ordered collection of an object that can be **indexed** individually. It is basically an alternative to an array. However unlike array you can add and remove items from a list at a specified position using an **index** and the array resizes itself automatically. It also allows dynamic memory allocation, add, search and sort items in the list.
Hashtable	It uses a **key** to access the elements in the collection. A hash table is used when you need to access elements by using key, and you can identify a useful key value. Each item in the hash table has a **key/value** pair. The key is used to access the items in the collection.
SortedList	It uses a **key** as well as an **index** to access the items in a list. A sorted list is a combination of an array and a hash table. It contains a list of items that can be accessed using a key or an index. If you access items using an index, it is an ArrayList, and if you access items using a key , it is a Hashtable. The collection of items is always sorted by the key value.
Stack	It represents a **last-in, first out** collection of object. It is used when you need a last-in, first-out access of items. When you add an item in the list, it is called **pushing** the item and when you remove it, it is called**popping** the item.
Queue	It represents a **first-in, first out** collection of object. It is used when you need a first-in, first-out access of

	items. When you add an item in the list, it is called **enqueue** and when you remove an item, it is called**deque**.
BitArray	It represents an array of the **binary representation** using the values 1 and 0. It is used when you need to store the bits but do not know the number of bits in advance. You can access items from the BitArray collection by using an **integer index**, which starts from zero.

REGULAR EXPRESSION

A **regular expression** is a pattern that could be matched against an input text. The .Net framework provides a regular expression engine that allows such matching.

In theoretical computer science and formal language theory, a **regular expression** (abbreviated **regex** or **regexp**) is a sequence ofcharacters that forms a search pattern, mainly for use in pattern matching with strings, or string matching, i.e. "find and replace"-like operations.

For example, the simple regexp ^[\t]+|[\t]+$ matches excess whitespace at the beginning or end of a line.

The Regex Class

The Regex class is used for representing a regular expression.

The Regex class has the following commonly used methods:

S.N	Methods & Description
1	**public bool IsMatch(string input)** Indicates whether the regular expression specified in the Regex constructor finds a match in a specified input string.
2	**public bool IsMatch(string input, int startat)** Indicates whether the regular expression specified in the Regex constructor finds a match in the specified input string, beginning at the specified starting position in the string.
3	**public static bool IsMatch(string input, string pattern)** Indicates whether the specified regular expression finds a match in the specified input string.
4	**public MatchCollection Matches(string input)** Searches the specified input string for all occurrences of a regular

	expression.
5	**public string Replace(string input, string replacement)** In a specified input string, replaces all strings that match a regular expression pattern with a specified replacement string.
6	**public string[] Split(string input)** Splits an input string into an array of substrings at the positions defined by a regular expression pattern specified in the Regex constructor.

```csharp
using System;
using System.Text.RegularExpressions;

namespace RegExApplication
{
   class Program
   {
      private static void showMatch(string text, string expr)
      {
         Console.WriteLine("The Expression: " + expr);
         MatchCollection mc = Regex.Matches(text, expr);
         foreach (Match m in mc)
         {
            Console.WriteLine(m);
         }
      }
      static void Main(string[] args)
      {
         string str = "A Thousand Splendid Suns";

         Console.WriteLine("Matching words that start with 'S': ");
         showMatch(str, @"\bS\S*");
         Console.ReadKey();
      }
   }
}
```
Matching words that start with 'S':
The Expression: \bS\S*
Splendid
Suns

INHERITANCE

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create

and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class. The idea of inheritance implements the **IS-A** relationship. For example, mammal **IS A** animal, dog **IS-A**mammal hence dog **IS-A** animal as well and so on.

Base and Derived Classes

A class can be derived from more than one class or interface, which means that it can inherit data and functions from multiple base class or interface.

The syntax used in C# for creating derived classes is as follows:

```
<acess-specifier> class <base_class>
{
 ...
}
class <derived_class> : <base_class>
{
 ...
}
```

Consider a base class Shape and its derived class Rectangle:

```
using System;
namespace InheritanceApplication
{
   class Shape
   {
      public void setWidth(int w)
      {
         width = w;
      }
      public void setHeight(int h)
      {
         height = h;
      }
      protected int width;
      protected int height;
   }

   // Derived class
   class Rectangle: Shape
   {
```

```
    public int getArea()
    {
      return (width * height);
    }
  }

  class RectangleTester
  {
    static void Main(string[] args)
    {
      Rectangle Rect = new Rectangle();

      Rect.setWidth(5);
      Rect.setHeight(7);

      // Print the area of the object.
      Console.WriteLine("Total area: {0}",  Rect.getArea());
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result:

```
Total area: 35
```

OPERATOR OVERLOADING

Overloaded operators are functions with special names the keyword **operator** followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

For example, look at the following function:

```
public static Box operator+ (Box b, Box c)
{
  Box box = new Box();
  box.length = b.length + c.length;
  box.breadth = b.breadth + c.breadth;
  box.height = b.height + c.height;
  return box;
}
```

The above function implements the addition operator (+) for a user-defined class Box. It adds the attributes of two Box objects and returns the resultant Box object.

Implementation of Operator Overloading

The following program shows the complete implementation:

```csharp
using System;

namespace OperatorOvlApplication
{
    class Box
    {
        private double length;      // Length of a box
        private double breadth;     // Breadth of a box
        private double height;      // Height of a box

        public double getVolume()
        {
            return length * breadth * height;
        }
        public void setLength( double len )
        {
            length = len;
        }

        public void setBreadth( double bre )
        {
            breadth = bre;
        }

        public void setHeight( double hei )
        {
            height = hei;
        }
        // Overload + operator to add two Box objects.
        public static Box operator+ (Box b, Box c)
        {
            Box box = new Box();
            box.length = b.length + c.length;
            box.breadth = b.breadth + c.breadth;
            box.height = b.height + c.height;
            return box;
        }

    }

    class Tester
    {
```

```
    static void Main(string[] args)
    {
       Box Box1 = new Box();        // Declare Box1 of type Box
       Box Box2 = new Box();        // Declare Box2 of type Box
       Box Box3 = new Box();        // Declare Box3 of type Box
       double volume = 0.0;         // Store the volume of a box here

       // box 1 specification
       Box1.setLength(6.0);
       Box1.setBreadth(7.0);
       Box1.setHeight(5.0);

       // box 2 specification
       Box2.setLength(12.0);
       Box2.setBreadth(13.0);
       Box2.setHeight(10.0);

       // volume of box 1
       volume = Box1.getVolume();
       Console.WriteLine("Volume of Box1 : {0}", volume);

       // volume of box 2
       volume = Box2.getVolume();
       Console.WriteLine("Volume of Box2 : {0}", volume);

       // Add two object as follows:
       Box3 = Box1 + Box2;

       // volume of box 3
       volume = Box3.getVolume();
       Console.WriteLine("Volume of Box3 : {0}", volume);
       Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Overloadable and Non-Overloadable Operators

The following table describes the overload ability of the operators in C#:

Operators	Description
+, -, !, ~, ++, --	These unary operators take one operand and can be overloaded.
+, -, *, /, %	These binary operators take one operand and can be overloaded.
==, !=, <, >, <=, >=	The comparison operators can be overloaded
&&, ||	The conditional logical operators cannot be overloaded directly.
+=, -=, *=, /=, %=	The assignment operators cannot be overloaded.
=, ., ?:, ->, new, is, sizeof, typeof	These operators cannot be overloaded.

EVENTS

Events are basically a user action like key press, clicks, mouse movements, etc., or some occurrence like system generated notifications. Applications need to respond to events when they occur.

An **event** can have many handlers. With the event handler syntax, we create a notification system.

To declare an event inside a class, first a delegate type for the event must be declared.

Next: The event keyword is used to create an instance of an event that can store methods in its invocation list.

```
using System;

public delegate void EventHandler();

class Program
{
    public static event EventHandler _show;

    static void Main()
    {
        // Add event handlers to Show event.
```

```
        _show += new EventHandler(Dog);
        _show += new EventHandler(Cat);
        _show += new EventHandler(Mouse);
        _show += new EventHandler(Mouse);

        // Invoke the event.
        _show.Invoke();
    }

    static void Cat()
    {
        Console.WriteLine("Cat");
    }

    static void Dog()
    {
        Console.WriteLine("Dog");
    }

    static void Mouse()
    {
        Console.WriteLine("Mouse");
    }
}
```

Output

```
Dog
Cat
Mouse
Mouse
```

Control flow begins in the Main entry point. The _show event has four method instances added to its invocation list with the plus "+=" operator. This has no relation to the arithmetic addition operator.

Next:In the invocation list of the _show event, we add four events: Dog, Cat and two instances of Mouse.

Finally:When the Invoke method is called, each of those method instances are called. The strings are printed to the console.

Console.WriteLine

CLASSES AND OBJECTS

- A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package.

- The data and functions within a class are called members of the class.

- When you define a class, you define a blueprint for a data type.

- This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

- The keyword **public** determines the access attributes of the members of the class that follow it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section.

Objects:

- A class provides the blueprints for objects, so basically an object is created from a class.

```
Module mybox
  Class Box
    Public length As Double   ' Length of a box
    Public breadth As Double   ' Breadth of a box
    Public height As Double    ' Height of a box
  End Class
  Sub Main()
    Dim Box1 As Box = New Box()      ' Declare Box1 OBJECT of type Box
    Dim Box2 As Box = New Box()      ' Declare Box2 OBJECT of type Box
    Dim volume As Double = 0.0    ' Store the volume of a box here
    ' box 1 specification
    Box1.height = 5.0
    Box1.length = 6.0
    Box1.breadth = 7.0
     ' box 2 specification
    Box2.height = 10.0
    Box2.length = 12.0
    Box2.breadth = 13.0
    'volume of box 1
    volume = Box1.height * Box1.length * Box1.breadth
    Console.WriteLine("Volume of Box1 : {0}", volume)
    'volume of box 2
```

```
    volume = Box2.height * Box2.length * Box2.breadth
    Console.WriteLine("Volume of Box2 : {0}", volume)
    Console.ReadKey()
  End Sub
End Module
```

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

FIELDS

- Fields, also known as data members, hold the internal state of an object.

- Their declarations appear only within class and structure declarations.

- Field declarations include an access modifier, which determines how visible the field is from code outside the containing class definition.

- The value stored in a field is specific to a particular object instance.

- Two instances can have different values in their corresponding fields. For example:

  - ✓ Dim emp1 As New Employee( )

  - ✓ Dim emp2 As New Employee( )

  - ✓ emp1.EmployeeNumber = 10

  - ✓ emp2.EmployeeNumber = 20 ' Doesn't affect emp1.

- Sometimes it is desirable to share a single value among all instances of a particular class. Declaring a field using the Shared keyword does this, as shown here:

  - ✓ Public Class X

  - ✓ Public Shared a As Integer

  - ✓ End Class

- Changing the field value through one instance affects what all other instances see. For example:

  - ✓ Dim q As New X( )

- ✓ Dim r As New X( )

- ✓ q.a = 10

- ✓ r.a = 20

- ✓ Console.WriteLine(q.a) ' Writes 20, not 10.

- Shared fields are also accessible through the class name:

  - ✓ Console.WriteLine(X.a)

READ-ONLY FIELDS

- Fields can be declared with the ReadOnly modifier, which signifies that the field's value can be set only in a constructor for the enclosing class.

- This gives the benefits of a constant when the value of the constant isn't known at compile time or can't be expressed in a constant initializer.

- Here's an example of a class that has a read-only field initialized in the class's constructor:

  - ✓ Public Class MyDataTier

  - ✓ Public ReadOnly ActiveConnection As System.Data.SqlClient.SqlConnection

  - ✓ Public Sub New(ByVal ConnectionString As String)

  - ✓ ActiveConnection = NeW System.Data.SqlClient.SqlConnection(ConnectionString)

  - ✓ End Sub

  - ✓ End Class

- The ReadOnly modifier applies only to the field itself—not to members of any object referenced by the field. For example, given the previous declaration of the MyDataTier class, the following code is legal:

  - ✓ Dim mydata As New MyDataTier(strConnection)

  - ✓ mydata.ActiveConnection.ConnectionString = strSomeOtherConnection

METHODS

Methods are members that contain code. They are either subroutines (which don't have a return value) or functions (which do have a return value).

Subroutine definitions look like this:

[method_modifiers] Sub [attribute_list] method_name ([parameter_list]) [handles_or_implements]

[method_body]

End Sub

Function definitions look like this:

[method_modifiers] Function method_name ([parameter_list]) [As type_name] [handles_or_implements]

[method_body]

End Function

EXAMPLE:

```
Module mybox
  Class Box
    Public length As Double   ' Length of a box
    Public breadth As Double   ' Breadth of a box
    Public height As Double    ' Height of a box
    Public Sub setLength(ByVal len As Double)
       length = len
    End Sub
    Public Sub setBreadth(ByVal bre As Double)
       breadth = bre
    End Sub
    Public Sub setHeight(ByVal hei As Double)
       height = hei
    End Sub
    Public Function getVolume() As Double
       Return length * breadth * height
    End Function
  End Class
  Sub Main()
    Dim Box1 As Box = New Box()      ' Declare Box1 of type Box
    Dim Box2 As Box = New Box()      ' Declare Box2 of type Box
    Dim volume As Double = 0.0    ' Store the volume of a box here

    ' box 1 specification
    Box1.setLength(6.0)
```

```
    Box1.setBreadth(7.0)
    Box1.setHeight(5.0)

    'box 2 specification
    Box2.setLength(12.0)
    Box2.setBreadth(13.0)
    Box2.setHeight(10.0)

    ' volume of box 1
    volume = Box1.getVolume()
    Console.WriteLine("Volume of Box1 : {0}", volume)

    'volume of box 2
    volume = Box2.getVolume()
    Console.WriteLine("Volume of Box2 : {0}", volume)
    Console.ReadKey()
  End Sub
End Module
```

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

PROPERTY

- A **Property** is similar to a Function.With a getter and a setter, it controls access to a value.
- This value is called a backing store.
- With Get it returns a value. With Set it stores a value.

GET, SET
- This example program uses the Property keyword.
- On the Number () property, we provide a Get block and a Set block.
- In Get we return a value the backing store count.
- In Set we receive a parameter and then store it in the count field.

TYPES:
- A property can have any data type.

- It does not need to be an Integer.

- It can be a Class.

Program that uses property syntax: VB.NET

```
Class Example
    Private _count As Integer
    Public Property Number() As Integer
        Get
            Return _count
        End Get
        Set(ByVal value As Integer)
            _count = value
        End Set
    End Property
End Class

Module Module1
    Sub Main()
        Dim e As Example = New Example()
        ' Set property.
        e.Number = 1
        ' Get property.
        Console.WriteLine(e.Number)
    End Sub
End Module
```

Output
1

- **When the value 1** is assigned to the Number property, Set is executed.

- The count field stores the value 1.

- When the Number property is accessed but not assigned to, Get is executed.

- The value of the count field is returned.

- A property must have both Get and Set members. It is possible to use the ReadOnly or WriteOnly keywords to eliminate this requirement.

READONLY

- The ReadOnly modifier changes the Property type to only have a Get method.

- **In this program,** the Count() property returns a constant Integer.

- But Get could perform any calculation and return the value of a field.

- ReadOnly does not require a constant return value.

- If we try to assign a value to Count, we get this error: "Property Count is ReadOnly." So don't do that.

Program that uses ReadOnly Property: VB.NET

```vbnet
Class Example
    Public ReadOnly Property Count() As Integer
        Get
            Return 500
        End Get
    End Property
End Class

Module Module1
    Sub Main()
        Dim e As Example = New Example()
        Console.WriteLine(e.Count)
    End Sub
End Module
```

Output

500

INHERITANCE

- Visual Basic supports *inheritance*, which is the ability to define classes that serve as the basis for derived classes.

- Derived classes inherit, and can extend, the properties, methods, and events of the base class. Derived classes can also override inherited methods with new implementations.

- By default, all classes created with Visual Basic are inheritable.

- Inheritance lets you write and debug a class one time and then reuse that code as the basis of new classes.

- Inheritance also lets you use inheritance-based *polymorphism*, which is the ability to define classes that can be used interchangeably by client code at run time, but with functionally different, yet identically named methods or properties.

SYNTAX

The syntax used in VB.Net for creating derived classes is as follows:

```
<access-specifier> Class <base_class>
...
End Class
Class <derived_class>: Inherits <base_class>
...
End Class
```

```
' Base class
Class Shape
   Protected width As Integer
   Protected height As Integer
   Public Sub setWidth(ByVal w As Integer)
      width = w
   End Sub
   Public Sub setHeight(ByVal h As Integer)
      height = h
   End Sub
End Class
' Derived class
Class Rectangle : Inherits Shape
   Public Function getArea() As Integer
      Return (width * height)
   End Function
End Class
Class RectangleTester
   Shared Sub Main()
      Dim rect As Rectangle = New Rectangle()
      rect.setWidth(5)
      rect.setHeight(7)
      ' Print the area of the object.
      Console.WriteLine("Total area: {0}", rect.getArea())
      Console.ReadKey()
   End Sub
End Class
```

When the above code is compiled and executed, it produces the following result:

Total area: 35

POLYMORPHISM

- It is the ability of classes to provide different implementation of methods that are called by same name.

- Polymorphism is one of the primary characteristics of Object Oriented Programming.

- "Poly" means "many" and "morph" means "form".

- Thus polymorphism refers to being able to use many form of a type without regard to the details.

- Polymorphism is the ability to redefine method for derived classes.

Types of Polymorphism

Compile Time Polymorphism.

Runtime Polymorphism.

Compile Time Polymorphism

- Compile time Polymorphism also known as Method Overloading.

- Method Overloading means having two or more methods with the same name but with different parameters.

- It is also called as static polymorphism.

- Which method is to be called is decided at the compile time only.

- In static Polymorphism decision is taken at the Compile time.

```
Module Module1
  Public Class mul
    Public a, b As Integer
    Public c, d As Double

    Public Function mul(ByVal a As Integer) As Integer
      Return a
    End Function
    Public Function mul(ByVal a As Integer,
        ByVal b As Integer) As Integer
      Return a * b
```

```
      End Function
      Public Function mul(ByVal d As Double,
              ByVal c As Double) As Double
        Return d * c
      End Function

    End Class

    Sub Main()
       Dim res As New mul
       System.Console.WriteLine
            ("Overloaded Values of Class Mul is::")
       System.Console.WriteLine(res.mul(10))
       System.Console.WriteLine(res.mul(20, 10))
       System.Console.WriteLine(res.mul(12.12, 13.23))
       Console.Read()
    End Sub
End Module
```

Result:
   Overloaded values of Class Mul is:
   10
   200
   160.3476

Runtime Polymorphism

- Run Time Polymorphism also known as Method Overriding.

- Method Overriding means having two or more methods with the same name, same parameters but with different implementation.

- It is also called as Dynamic Polymorphism.

- In this process, an overridden method is called through reference variable of a superclass; the determination of the method to be called is based on the object being referred to by reference variable.

- The parent class uses the same virtual keyword. Every class that overrides the virtual method will use the override keyword.

Over riding in VB.NET

   Overriding in VB.net is method by which a inherited property or a method is overidden to perform a different functionality in a derived class. The base class function is declared using a

keyword Overridable and the derived class function where the functionality is changed contains an keyword Overrides.

Example:
```
  Module Module1

  Class Over
    Public Overridable Function add(ByVal x As Integer,
        ByVal y As Integer)
      Console.WriteLine('Function Inside Base Class')
      Return (x + y)
    End Function
  End Class

  Class DerOver
   Inherits Over
   Public Overrides Function add(ByVal x As Integer,
        ByVal y As Integer)
      Console.WriteLine(MyBase.add(120, 100))
      Console.WriteLine('Function Inside Derived Class')
      Return (x + y)
   End Function
  End Class

  Sub Main()
    Dim obj As New DerOver
    Console.WriteLine(obj.add(10, 100))
    Console.Read()
  End Sub
 End Module
```
Result:
  Function Inside Base Class
  220
  Function Inside Derived Class
  110

Description:

In the above overriding example the base class function **add** is overridden in the derived class using the **MyBase.add(120,100)**statement. So first the overidden value is displayed, then the value from the derived class is displayed.

OPERATOR OVERLOADING

- Operator overloading is the ability for you to define procedures for a set of operators on a given type.
- This allows you to write more intuitive and more readable code.
- When an operator can perform more than one operations, specially with objects, known as operator overloading.
- A function name can be replaced with the operator using **operator** keyword.

Important points related to Operator overloading or guidelines:

- Don't change an operator's semantic meaning. The result of an operator should be intuitive.
- Make certain overloaded operators are shared methods.
- You cannot overload assignment e.g., the "=" operator .

The operators that can be defined on your class or structure are as follows:

- ✓ Unary operators:
- ✓   + - Not IsTrue IsFalse CType
- ✓ Binary operators:
- ✓   + - * / \ & Like Mod And Or Xor
- ✓   ^ << >> = <> > < >= <=
- ✓ You can't overload member access, method invocation, or the **AndAlso**, **OrElse**, **New, TypeOf... Is**, **Is**, **IsNot**, **AddressOf**, **GetType**, and **AsType** operators.

- Note that assignment itself (=) is a statement, so you can't overload it either.

- Secondly, operator overloading cannot be used to change to order in which operators are executed on a given line.

- For example, multiplication will always happen before addition, unless parentheses are used appropriately.

```
Module Module1
    Class Complex
      Private x As [Double]
      Private y As [Double]
      Public Sub New()
      End Sub
      Public Sub New(ByVal real As [Double], ByVal image As [Double])
        x = real
        y = image
      End Sub
      Public Shared Operator +(ByVal c1 As complex, ByVal c2 As complex) As complex
        Dim c3 As New Complex()
        c3.x = c1.x + c2.x
        c3.y = c1.y + c2.y
        Return (c3)
```

```
        End Operator
      Public Sub Display()
        Console.Write(x)
        Console.Write("+j" & Convert.ToString(y))
        Console.WriteLine()
      End Sub
    End Class
    Sub Main()
      Dim a As Complex, b As Complex, c As Complex
      a = New Complex(2.5, 3.5)
      b = New Complex(1.6, 2.7)
      c = a + b
      Console.Write("a=")
      a.Display()
      Console.Write("b=")
      b.Display()
      Console.Write("c=")
      c.Display()
    End Sub
  End Module
```

OUTPUT

a=2.5+3.5j

b=1.6+2.7j

c=4.1+6.2j

INTERFACES

- *Interfaces in VB.net* are used to define the class members using a keyword **Interface**, without actually specifying how it should be implemented in a Class.
- Interfaces are examples for multiple Inheritances and are implemented in the classes using the keyword **Implements** that is used before any Dim statement in a class.

Example:
```
  Module Module1
  Public Interface Interface1
     Function Add(ByVal x As Integer) As Integer
  End Interface

  Public Class first Implements Interface1
     Public Function Add(ByVal x As Integer)  As Integer Implements Interface1.Add
        Console.WriteLine("Implementing x+x in first class::" & (x + x))
     End Function
  End Class
```

```
Public Class second Implements Interface1
    Public Function Add(ByVal x As Integer)  As Integer Implements Interface1.Add
        Console.WriteLine("Implementing x+x+x  in second class::" & (x + x + x))
    End Function
End Class

Sub Main()
    Dim obj1 As New first
    Dim obj2 As New second
    obj1.Add(10)
    obj2.Add(50)
    Console.Read()
End Sub
End Module
```

Result:
   Implementing x+x in first class:: 20
    Implementing x+x+x in second class:: 150

Description:

In the above example interface **Interface1** is implemented in classes **first** and **second** but differently to add the value of 'x' twice and thrice respectively.

ARRAY

- An array stores a fixed-size sequential collection of elements of the same type.

- An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

- All arrays consist of contiguous memory locations.

- The lowest address corresponds to the first element and the highest address to the last element.
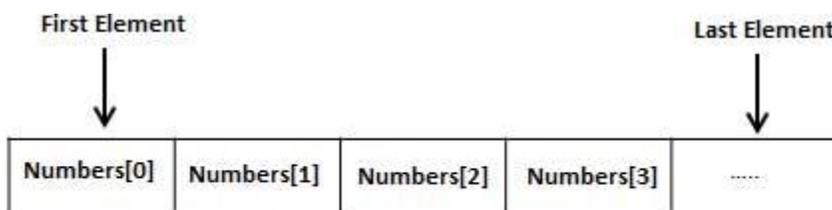
CREATING ARRAYS IN VB.NET

To declare an array in VB.Net, you use the Dim statement. For example,

```
Dim intData(30)      ' an array of 31 elements
Dim strData(20) As String   ' an array of 21 strings
Dim twoDarray(10, 20) As Integer     'a two dimensional array of integers
Dim ranges(10, 100)          'a two dimensional array
```

You can also initialize the array elements while declaring the array. For example,

```
Dim intData() As Integer = {12, 16, 20, 24, 28, 32}
Dim names() As String = {"Karthik", "Sandhya", _
"Shivangi", "Ashwitha", "Somnath"}
Dim miscData() As Object = {"Hello World", 12d, 16ui, "A"c}
```

The elements in an array can be stored and accessed by using the index of the array. The following program demonstrates this:

```
Module arrayApl
   Sub Main()
      Dim n(10) As Integer  ' n is an array of 11 integers '
      Dim i, j As Integer
      ' initialize elements of array n '
      For i = 0 To 10
         n(i) = i + 100 ' set element at location i to i + 100
      Next i
      ' output each array element's value '
      For j = 0 To 10
         Console.WriteLine("Element({0}) = {1}", j, n(j))
      Next j
      Console.ReadKey()
   End Sub
End Module
```

When the above code is compiled and executed, it produces the following result:

```
Element(0) = 100
Element(1) = 101
Element(2) = 102
Element(3) = 103
Element(4) = 104
Element(5) = 105
Element(6) = 106
Element(7) = 107
Element(8) = 108
Element(9) = 109
```

Element(10) = 110

MULTI-DIMENSIONAL ARRAYS

VB.Net allows multidimensional arrays. Multidimensional arrays are also called rectangular arrays.

You can declare a 2-dimensional array of strings as:

```
Dim twoDStringArray(10, 20) As String
```

or, a 3-dimensional array of Integer variables:

```
Dim threeDIntArray(10, 10, 10) As Integer
```

The following program demonstrates creating and using a 2-dimensional array:

```
Module arrayApl
   Sub Main()
      ' an array with 5 rows and 2 columns
      Dim a(,) As Integer = {{0, 0}, {1, 2}, {2, 4}, {3, 6}, {4, 8}}
      Dim i, j As Integer
      ' output each array element's value '
      For i = 0 To 4
         For j = 0 To 1
            Console.WriteLine("a[{0},{1}] = {2}", i, j, a(i, j))
         Next j
      Next i
      Console.ReadKey()
   End Sub
End Module
```

When the above code is compiled and executed, it produces the following result:

```
a[0,0]: 0
a[0,1]: 0
a[1,0]: 1
a[1,1]: 2
a[2,0]: 2
a[2,1]: 4
a[3,0]: 3
a[3,1]: 6
a[4,0]: 4
a[4,1]: 8
```

INDEXERS

- An indexer is just like a method containing indexer parameter and return type. The return type can be any valid type. An Indexer must at least have one parameter or the complier throws an error.

- The main concept behind the indexer is to; treat the object and variables acting as a protected or Internal.

- The keyword which is used to identify the indexer is "default" and "this".

- "Default keyword is used when the variables act as an array.

- "this" keyword is used when the objects act As an array.

- An indexer also uses the get, set method as property uses the get, set method.

- The difference between the Indexers and Properties is, the set method of the Indexer should have a signature. (i.e.) Parameters are passed to the set method of the Indexers.

Example for how the 'default' and 'this' keyword are used with get, set method:

Default public property Item (ByVal index As Integer) As string

Get

Return mobiles (index -1)

End Get

Set (ByVal value As string)

Index = value

End Set

End Property

this [parameter]

Get

// Get codes here

End Get

Set (Parameter)

//Set codes here

End Set

Now let us consider the following Example:

Public class WeekDays

```
Private weekDays (7) As string
Public Sub New ()
weekDays (0) = "Monday"
weekDays (1) = "Tuesday"
weekDays (2) = "Wednesday"
weekDays (3) = "Thursday"
weekDays (4) = "Friday"
weekDays (5) = "Saturday"
weekDays (6) ="Sunday"
End Sub
Public Function GetDayByNumber (ByVal dayNumber As Integer) As String
Return weekDays (dayNumber)
End Function
End Class
Sub Main ()
Dim wk As New WeekDays
Console.WriteLine (wk.GetDayByNumber (3))
End Sub
```

OUTPUT is: 'Thursday'

Now let us see how we can define an indexer in the WeekDays class. In this case, indexer is defined using the 'default' keyword because; we are going to make use of the variables.

```
 Public class WeekDays
Private weekDays (7) As String
Public Sub New ()
weekDays (0) = "Monday"
weekDays (1) = "Tuesday"
weekDays (2) = "Wednesday"
weekDays (3) = "Thursday"
weekDays (4) = "Friday"
weekDays (5) = "Saturday"
weekDays (6) = "Sunday"
```

End Sub

Default public property WeekDay (ByVal dayNumber As Integer) As string

Get

Return weekDays (dayNumber)

End Get

Set (ByVal value As string)

weekDays (dayNumber) = value

End Set

End Property

End class

We use the indexer as the following manner; The indexer has essentially turned the WeekDays class into an array. We can index the wk object directly.

Dim wk As New WeekDays

Console.WriteLine (wk (3))

OUTPUT is: 'Thursday'

COLLECTIONS

- A collection is a way of grouping and managing related objects. A collection is either zero-based or one-based, depending on what its starting index is. The zero based collection means that the index of the first item in the collection is Zero. One based collection means that the index of the first item in one.

- Visual Basic .Net Collections are data structures that holds Data in different ways for flexible operations. The important data structures in the collections are ArrayList, HashTable, Stack and Queue etc.

- **ArrayList**

- ArrayList is one of the most flexible data structures from VB.Net collection. ArrayList contains a simple list of values and very easily we can add, insert, delete, and view etc. to do with ArrayList. It is very flexible because we can add without any size information, that is, it can grow dynamically and also shrink.

Important Functions in ArrayList

Add: Add an item in an ArrayList

Syntax: ArrayList.add (Item)

Item – the item to be added to the array list.

Example:

Dim ItemList As New ArrayList ()

ItemList.Add ("Item4")

Insert:Insert an item in a specified position in an ArrrayList.

Syntax: ArrayList. Insert (index, item)

Index – The position of the item in an ArrayList

Item – The Item to be inserted in the ArrayList.

Example:

ItemList.Insert (3, "item6")

Remove: Remove an item from ArrayList.

Syntax: ArrayList. Remove (item)

Item – the item to be removed in the ArrayList

Example:

ItemList. Remove ("item 2")

Remove At:Remove An item from a specified position.

Syntax: ArrayList. RemoveAt (index)

Index – the position of an item to remove from an array list.

Example:

ItemList. RemoveAt (2)

Sort: Sort Items in an ArrayList.

Syntax: ArrayList.Sort ()

Stack

Stack is one of another easy method used in VB.Net. Stack follows the push – pop operations, that is we can push Items into stack and pop it later also it follows the last in First Out (LIFO) system. That is we can push the items into a stack and get it in reverse order. Stack returns the last item first.

Commonly Used Methods

Push:Add (Push) an item in the stack Data Structure.

Syntax: Stack.Push (object)

Object – The item to be inserted.

Pop:Pop returns the item, last item inserted in stack.

Syntax: Stack.Pop ()

Returns the last object in the stack.

Contains:Check the object contains in the Stack.

Syntax: Stack.Contains (object)

Object – The specified object to be searched.

Queue

Queue works like First in First Out Method. The item added first in the queue is first to get out from queue. We can Enqueue (add) items in Queue and we can dequeue (remove from queue) or we can peek (that is get the reference of first item added in Queue) the item from Queue.

Commonly Used Functions

Enqueue:Add an item in Queue.

Syntax: Stack.Enqueue (object)

Object – the item to add in Queue.

Dequeue:Remove the oldest item from Queue (We don't get the item later)

Syntax: Stack.Dequeue ()

Returns – remove the oldest item and return

Peek:Get the reference of the oldest item (it is not removed permanently)

Syntax: Stack.Peek ()

Returns – Get the reference of the oldest item in the Queue.

HashTable

HashTable stores a key value pair type collection of Data. We can retrieve items from HashTable to provide the Key. Both Key and value are objects.

Common Functions Using in HashTables

Add:To add a pair of value in HashTable

Syntax: HashTable.Add (key, value)

  key – The Key value

  value – The value of corresponding key

Constraint Key:Check if a specified key exist or not

Syntax: HashTable.constraintkey (key)

   Key – The key value for search in HashTable

Contains value:Check the specified value exists in Hash Table.

Syntax: HashTable.containsValue (value)

  Value – search the specified value in HashTable.

Remove: Remove the specified key and corresponding value

Syntax: HashTable.Remove (key)

  Key – The argument key of deleting pairs.

Example Program for ArrayList:

Public class Items

Dim i As Integer

Dim ItemList As New ArrayList ()

ItemList.Add ("Item 4")

ItemList.Add ("Item 5")

ItemList.Add ("Item 2")

ItemList.Add ("Item 1")

ItemList.Add ("Item 3")

MsgBox ("shows Added Items")

For i=0 to ItemList.count-1

MsgBox (ItemsList.Item (i))

Next

ItemList.Insert (3, "Item6")

ItemList.Sort ()

ItemList.Remove ("Item 1")

ItemList.RemoveAt (3)

MsgBox ("shows final Items the ArrayList")

For i=0 to ItemList.count-1

MsgBox (ItemList.Item (i))

Next

End class

STRINGS

The string class represents character strings. The string Object is Immutable; it cannot be modified once it is created. That means every time we use any operation in the string object, we must create a new String object.

Now we are going to see the important methods used in string class.

Length ( )

Insert ( )

Index of ( )

Equals ( )

Compare ( )

Substring ( )

Split ( )

Endswith ( )

Concat ( )

String.Length ( )

<u>Syntax:</u> string.Length () As Integer.

The method returns the number of characters in the specified string.

<u>Example:</u> "This is a Test".

Length () returns 14.

<u>Program:</u>

Public class As String

str = "This is a Test"

MsgBox (str.length ())

End class

String.Insert ( )

<u>Syntax:</u> String.Insert (Integer ind, String str) As String

Ind – the index of the specified string to be inserted.

Str – the string to be inserted.

Example: "This is Test".

Insert (8, "Insert") returns "This is Insert Test"

Program:

Public class Insert

Dim str As string = "This is VB.NET Test"

Dim insStr As string = "Insert"

Dim strRes As string = str.Insert (15, insStr)

MsgBox (strRes)

End class

String.Indexof ( )

Syntax: String.Indexof (string str) As Integer

str – parameter string to check its occurrences.

Example: "This is a test"

Indexof ("Test") returns 10.

Program:

Public class Index

Dim str As string

str = "VB.NET Top 10 Books"

MsgBox (str.Indexof ("Book"))

End class

String.Equals ( )

Syntax: String.Equals (String str1, String str2) As Boolean

str1 – the string argument

str2 – the string argument

Example:

str1 = "Equals ()"

str2 = "Equals ()"

String.Equals (str1, str2) returns True.

Program:

Public class Equals

Dim str1 As string ="Equals"

Dim str2 As string = "Equals"

If string.Equals (str1, str2) Then

MsgBox ("Strings are not Equal ( )")

End If

End class

String.Compare ()

Syntax: string.Compare (string str1, string str2, Boolean) As Integer

str1 – parameter string

Str2 – parameter string

Boolean TRUE/FALSE – Indication to check the string with case sensitive or without case sensitive.

Returns 0 or <0 or >0

>0 – str1 is greater than str2.

0 – str1 is Equal to str2.

<0 – str1 is less than str2.

Program:

Public class Compare

Dim str1 As string

Dim str2 As string

str1= "vb.net"

str2= "VB.Net"

Dim result As Integer

result= string.compare (str1, str2)

MsgBox (result)

result= string.compare (str1, str2, True)

MsgBox (result)

End class

String.substring ( )

String.Substring (Integer StartIndex, Integer length) As string

StartIndex – The Index of the start of the substring.

length – The number of characters in the substring

Program:

```
Public class stringName
Dim str As string
Dim retstring As string
str ="This is substring test"
retstring =str.Substring (8, 9)
MsgBox (retstring)
End class
String.Split ( )
```
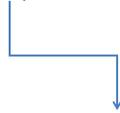
Syntax: String.Split (" ") As string

Delimiter (separator)

Program:

```
Public class split
Dim str As string
Dim strArr ( ) As String
Dim count As Integer
str= "Vb.Net Split test"
strArr= str.split (" ")
for count -0 to strArr.Length-1
MsgBox (strArr(count))
Next
End class
```

String.Endswith ( )

Syntax: String.Endswith (string suffix) As Boolean

  Suffix- the passing string for it Ends with.

Example:

"This is a Test".

EndsWith ("Test") returns True.

"This is a Test".

EndsWith ("is") returns False.

String.Concat ( )

Syntax: String.Concat (string str1, string str2) As string

      str1 – parameter string

      str2 – parameter string

Returns a new string with str1 concat with str2.

REGULAR EXPRESSIONS

Regular Expressions provide a language specifically for parsing and processing strings. They are highly optimized and can be created as an instance of the Regex Class.

Regular Expression is useful for parsing and processing character based Data, particularly the following actions can be performed.

Replace or Delete sub strings at matches (REPLACE method).

Parsing input text (SPLIT method)

Extracting data from structured sources such as XML or HTML (Matches or Match method)

Detecting complex patterns of Characters(Is Match method)

These methods can all be used on pre-created instances for speed, they can be or executed on the fly using static methods on the Regex class.

The relevant code is located in the System.Text.RegularExpressions Namespaces.

There is a collection of special characters ' , $ ^ { [(|) * + ? \' which is used to represent special characters, groups of characters, repeated characters of words.

To represent these without their special meaning they are escaped using the \ character.

Character escapes match single (non printable) characters or characters with special meaning.

? – Match one or Zero occurrences.

( ) – Sub expression treated as a single element.

\n – newline character

\s – white space character

\w – word character

^ - Beginning of String

& - End of the string or Line.

The various regular Expressions are:

Regex Matches

Word count

Regex ('\w+')

Count chars: Regex.Matches (quote,.)

Word count: Regex.Matches (quote, '\w+')

Count Line: Regex.Matches (quote, '.+\n*')

Regex.Compare to assembly

Match Regular Expression to string and print out all the matches.

Using Regex method Replace: substituted for * with another String.

Using Regex method Replace: Replace one string with another string.

Every word replaces by another word.

Replace first 3 digits.

String split at Commas.

 Examples for Regular Expressions

Word Count

Public class Tester

Sub main ( )

Dim quote As String = " q d e w"

Do while (quote.Indexof(space (2))>=0)

quote = quote.replace (space (2), space (1))

loop

Dim wordcount As Integer= split (quote, space (1)).Length

Console.WriteLine (quote & vbNewLine & "Number of words:" & wordcount.ToString)

End sub

End class

OUTPUT:

q d e w   No. of words: 5

Regex ('\w+')

Public class Tester

Sub main ( )

Dim quote As sting = "The important thing is not to" & - "Stop Questioning. …. Albert Einstein"

Dim parser as New Regex ("\w+")

Dim total Matches As Integer = parser.Matches (quote).count

Console.WriteLine (quote & vbNewLine & " Number of Words:" & - TotalMatches.Tostring)

End Sub

End class

OUTPUT:

The Important thing is not to stop Questioning. – Albert Einstein.

Number of Words: 10

Using Regex method Replace: Replace one string with another

Public class Tester

Sub Main ( )

Console.WriteLine (New Regex ("stars"). Replace ("This sentence ends in 5 stars *****", "carets"))

End sub

End class

OUTPUT:

This sentence ends in 5 carets*****

Every word replaced by another word

Public class Tester

Sub Main ( )

Console.WriteLine (Regex.Replace("This sentence ends in 5 stars *****", "\w+", "word"))

End sub

End class

<u>OUTPUT:</u>

Word word word word word word *****

Replace first 3 digits

Public class Tester

Sub Main ( )

Console.WriteLine (New Regex ("\d").Replace ("1,2,3,4,5,6,7,8", "digit", 3))

End Sub

End class

<u>OUTPUT:</u>

digit, digit, digit, 4,5,6,7,8.

PLATFORM TECHNOLOGY

UNIT-IV

EXCEPTION HANDLING

An exception is a problem that arises during the execution of a program. An exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. VB.Net exception handling is built upon four keywords: **Try**, **Catch**, **Finally** and **Throw**.

- **Try**: A Try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more Catch blocks.
- **Catch**: A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The Catch keyword indicates the catching of an exception.
- **Finally**: The Finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
- **Throw**: A program throws an exception when a problem shows up. This is done using a Throw keyword.

Syntax

Assuming a block will raise an exception, a method catches an exception using a combination of the Try and Catch keywords. A Try/Catch block is placed around the code that might generate an exception. Code within a Try/Catch block is referred to as protected code, and the syntax for using Try/Catch looks like the following:

```
Try
    [ tryStatements ]
    [ Exit Try ]
[ Catch [ exception [ As type ] ] [ When expression ]
    [ catchStatements ]
    [ Exit Try ] ]
[ Catch ... ]
[ Finally
    [ finallyStatements ] ]
End Try
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Exception Classes in .Net Framework

In the .Net Framework, exceptions are represented by classes. The exception classes in .Net Framework are mainly directly or indirectly derived from the **System.Exception** class. Some of the exception classes derived from the System.Exception class are the **System.ApplicationException** and **System.SystemException** classes.

The **System.ApplicationException** class supports exceptions generated by application programs. So the exceptions defined by the programmers should derive from this class.

The **System.SystemException** class is the base class for all predefined system exception.

Handling Exceptions

VB.Net provides a structured solution to the exception handling problems in the form of try and catch blocks. Using these blocks the core program statements are separated from the error-handling statements.

These error handling blocks are implemented using the **Try**, **Catch** and **Finally** keywords. Following is an example of throwing an exception when dividing by zero condition occurs:

```vbnet
Module exceptionProg
   Sub division(ByVal num1 As Integer, ByVal num2 As Integer)
      Dim result As Integer
      Try
         result = num1 \ num2
      Catch e As DivideByZeroException
         Console.WriteLine("Exception caught: {0}", e)
      Finally
         Console.WriteLine("Result: {0}", result)
      End Try
   End Sub
   Sub Main()
      division(25, 0)
      Console.ReadKey()
   End Sub
End Module
```

When the above code is compiled and executed, it produces the following result:

```
Exception caught: System.DivideByZeroException: Attempted to divide by zero.
at ...
Result: 0
```

Delegates in vb.net

Delegates are pointers that are used to store and tranfer information like the memory address, event handled by functions and subroutines. Delagates are type safe, since they check for the signatures of functions and subroutines only if same, they transfer information. A delegate is declared using the keyword Delegate to a function or procedure name.

Example:
```
Module Module1
  Public Delegate Function del1(ByVal x As Integer,
      ByVal y As Integer) As Integer
  Public Delegate Sub Display()


   Public Function check(ByVal x As Integer,
       ByVal y As Integer) As Integer
     Return (x + y)
    End Function

   Public Function multi(ByVal x As Integer,
       ByVal y As Integer) As Integer
     Return (x * y)
    End Function

   Class deleg
     Public Sub Disp()
       Console.WriteLine("Method inside the Class")
     End Sub
   End Class

  Sub Main()

    Dim a As New del1(AddressOf check)
    Dim b As New del1(AddressOf multi)
    Dim ob As New deleg
    Dim c As New Display(AddressOf ob.Disp)
    Console.WriteLine(a(10, 20))
    Console.WriteLine(b(10, 20))
    c()
    Console.Read()
  End Sub

 End Module
```
Result:
```
  30
  200
```

Method Inside Class

Description:

In the above Delegate example **del1** is a delegate that gets two integer arguments and return an integer. Using this delegate with the keyword **AddressOf** the values stored in the memory location of the procedures **check** and **multi** are retreived.

Using the delegate **Display** the sub procedure **Disp()** inside the class **deleg** is also displayed.

The **AddressOf** operator creates a function delegate that points to the function specified by *procedurename*. When the specified procedure is an instance method then the function delegate refers to both the instance and the method. Then, when the function delegate is invoked the specified method of the specified instance is called.

EVENTS

Events are basically a user action like key press, clicks, mouse movements, etc., or some occurrence like system generated notifications. Applications need to respond to events when they occur.

Clicking on a button, or entering some text in a text box, or clicking on a menu item, all are examples of events. An event is an action that calls a function or may cause another event.

Event handlers are functions that tell how to respond to an event.

VB.Net is an event-driven language. There are mainly two types of events:

- Mouse events

- Keyboard events

Mouse events occur with mouse movements in forms and controls. Following are the various mouse events related with a Control class:

- **MouseDown** - it occurs when a mouse button is pressed
- **MouseEnter** - it occurs when the mouse pointer enters the control
- **MouseHover** - it occurs when the mouse pointer hovers over the control
- **MouseLeave** - it occurs when the mouse pointer leaves the control
- **MouseMove** - it occurs when the mouse pointer moves over the control
- **MouseUp** - it occurs when the mouse pointer is over the control and the mouse button is released
The event handlers of the mouse events get an argument of type **MouseEventArgs**

Following are the various keyboard events related with a Control class:

- **KeyDown** - occurs when a key is pressed down and the control has focus
- **KeyPress** - occurs when a key is pressed and the control has focus
- **KeyUp** - occurs when a key is released while the control has focus

The event handlers of the KeyDown and KeyUp events get an argument of type **KeyEventArgs**

EXAMPLE
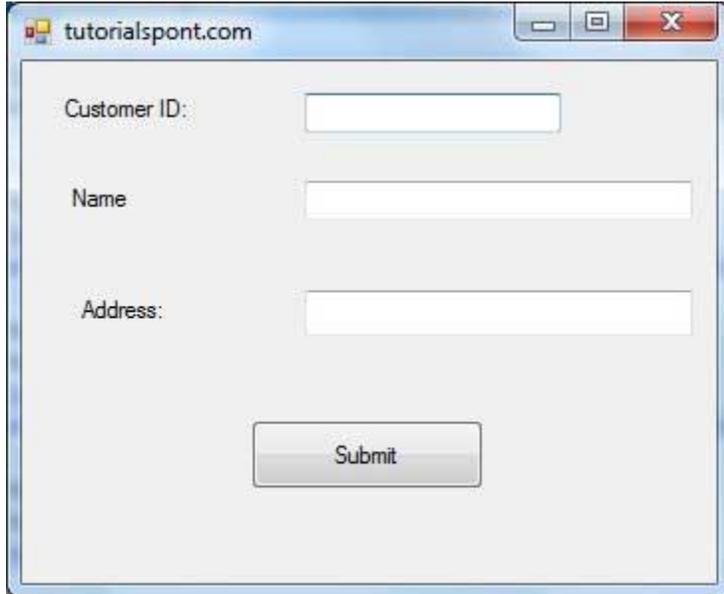
Add the above mentioned controls and add the following mouse events code.

```
Public Class Form1
 Private Sub txtID_MouseEnter(sender As Object, e As EventArgs)Handles txtID.MouseEnter
    'code for handling mouse enter on ID textbox
    txtID.BackColor = Color.CornflowerBlue
    txtID.ForeColor = Color.White
  End Sub
  Private Sub txtID_MouseLeave(sender As Object, e As EventArgs) Handles txtID.MouseLeave
    'code for handling mouse leave on ID textbox
    txtID.BackColor = Color.White
    txtID.ForeColor = Color.Blue
  End Sub
End class
```

Add the following key events handling code

```
Private Sub txtID_KeyUP(sender As Object, e As KeyEventArgs) Handles txtID.KeyUp
If (Not Char.IsNumber(ChrW(e.KeyCode))) Then
MessageBox.Show("Enter numbers for your Customer ID")
txtID.Text = " "
End If
```

- In the above code for keyboard and mouse event, you can see, there are basically two new things added:

  - The event subroutine arguments **sender** and **e**
  - The **Handles** clause
- The **Handles** clause allows the same subroutine to be used for all kinds of event calls like mouse,key etc.
- The event handler arguments, **sender** and **e**, are the same for all event subroutines, in part, so you can code an event subroutine that handles different types of events like the example above.
- The **sender** argument provides information about the event that was raised. There's a fairly small set of properties that sender provides directly.
- These properties carry just the basic information you need.
  - Equals
    GetHashCode
    GetType
    ReferenceEquals
    ToString

- The sender arguement is often more useful if you instantiate an object using it. For example, if you are processing a Button control:

  - **Dim myButton As Button = sender**

- If you are processing a PictureBox control:

  Dim myPicBox As PictureBox = sender
- Using these instances of controls, you can take advantage of the methods and properties of the control. For example, you can get the Name of a control:

  Dim myButton As Button = sender
  Debug.WriteLine(myButton.Name)

<div align="center">or</div>

Dim myTextBox As TextBox = sender

Debug.WriteLine(myTextBox.Name)

- The argument e carries whether it s a mouse arguments or keyboard arguments.

ADO.NET

- Every organization maintains data storage for its business, employees and clients.
- This data must be easily accessible and updateable.
- The machine in which the backend database is stored is known as server.
- This Database can be accessed and used by front end applications through some connectivity.
- The machine in which the application run is known as client.
- Previously ODBC was used for this purpose.
- ODBC – open database connectivity local/remote.

ADO – Active Data Object

- It is defined as object model.
- It comprises of objects, methods and properties.
- Methods – Access and update Data Sources.
- Properties – Control the behavior of the method.

ADO.NET OBJECT MODEL
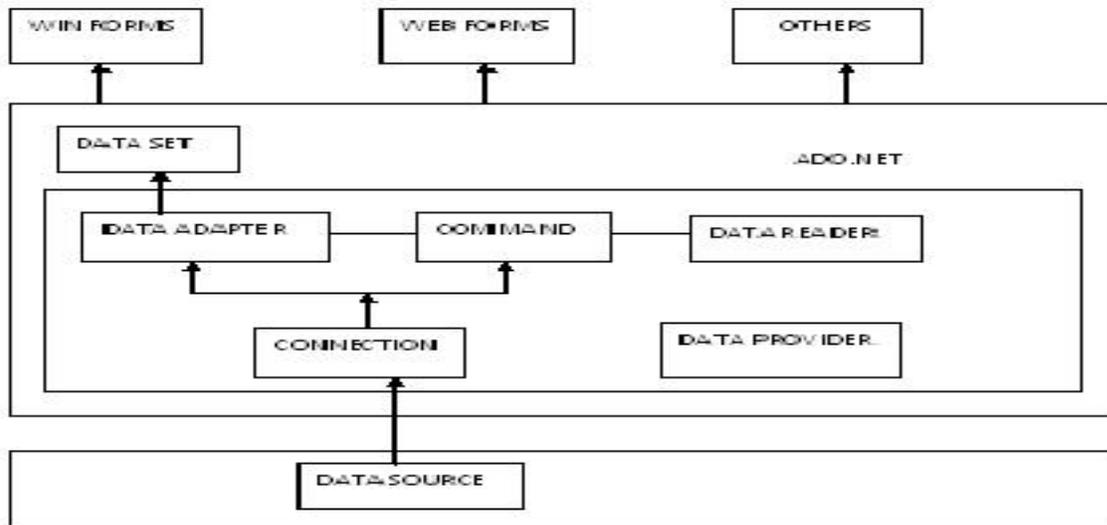
ADO.NET Object Model

The .Net Framework comprises of

1. Connection Oriented (or) Data Providers.

2. Disconnection Oriented (or) Datasets.

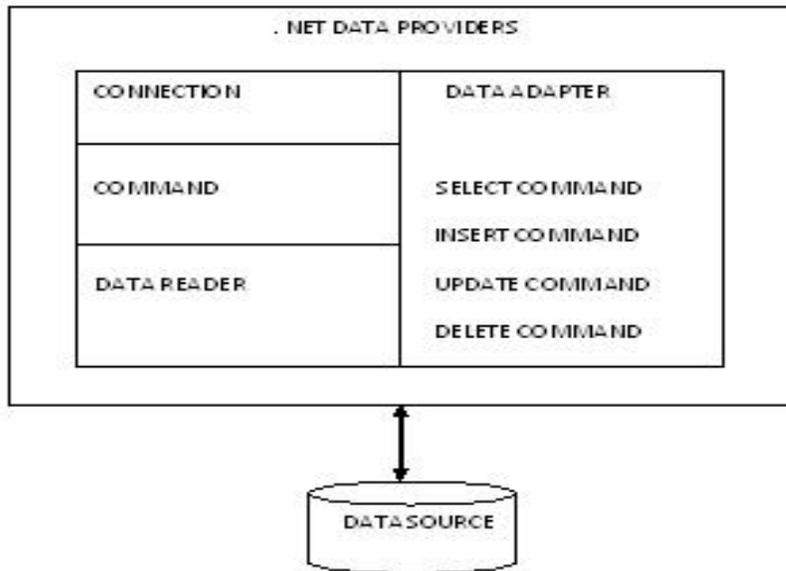**1. Connection Environment/ Data Providers:**

In connected the application makes a connection to the Data source and then interacts with it through SQL request using the same connection. In these cases the application stays connected to DB system even when it is not using any Database operations.

2. Disconnected Oriented/ Datasets:

A datasets is an in – memory data store that can hold multiple tables at the same time. Datasets can only hold data and do not interact with a data source.

DIRECT ACCESS (OR) .NET DATA PROVIDERS (OR) CONNETION ENVIRONMENT

Data providers act as a bridge between an application and database. It is used to retrieve data from a database and send back to the database after changes.

Connection Environment

.NET DATA PROVIDER

- We know that ADO.NET allows us to interact with different types of data sources and different types of databases.
- However, there isn't a single set of classes that allow you to accomplish this universally.
- Since different data sources expose different protocols, we need a way to communicate with the right data source using the right protocol.
- Some older data sources use the ODBC protocol, many newer data sources use the OleDb protocol, and there are more data sources every day that allow you to communicate with them directly through .NET ADO.NET class libraries.
- ADO.NET provides a relatively common way to interact with data sources, but comes in different sets of libraries for each way you can talk to a data source.
- These libraries are called Data Providers and are usually named for the protocol or data source type they allow you to interact with.

- Table 1 lists some well known data providers, the API prefix they use, and the type of data source they allow you to interact with.

Table 1. ADO.NET Data Providers are class libraries that allow a common way to interact with specific data sources or protocols. The library APIs have prefixes that indicate which provider they support.

Provider Name	API prefix	Data Source Description
ODBC Data Provider	Odbc	Data Sources with an ODBC interface. Normally older data bases.
OleDb Data Provider	OleDb	Data Sources that expose an OleDb interface, i.e. Access or Excel.
Oracle Data Provider	Oracle	For Oracle Databases.
SQL Data Provider	Sql	For interacting with Microsoft SQL Server.
Borland Data Provider	Bdp	Generic access to many databases such as Interbase, SQL Server, IBM DB2, and Oracle.

Connection Object

- These objects are used to create a connection to the database for moving data between the database and user application.
- The connection can be created using.

SQL Connection Object

Open (), close (), and dispose () are the methods of connection object used to open and close the defined connection.

Dim conn As SQLConnection

conn = New SQLConnection ()

conn.connectionstring ="datasource =asdf; initial catalog =stud; userId =sa; pwd =123"

conn.open ()

> Datasource is the name of the SQL server which is to be connected.

> Initial Catalog is the name of the database.

> Used Id is the SQL server login account and pwd is the password.

The user must close the connection after using it. This is done by either close (or) dispose method.

> conn.close ()

Command Objects

These command objects are used to read, add, update and delete records in a database. These operations are done by sql command objects.

Methods: Execute Reader ()

> Execute Non Query ()

Properties: connection, command text.

Execute Reader () method is used to execute the commands and retrieve rows from the database.

Execute Non Query () method is used to execute the command without retrieving any row.

> Dim conn As sqlconnection
>
> Dim comm As sqlcommand
>
> comm. = New sql command ()
>
> comm.connection = conn
>
> comm.commandText = "select * from Stud Table"

Data Reader Objects

These objects are used to read row wise data in read only format from databases. Sql Data Reader object is used.

Execute Reader () – is used to retrieve a row from database.

Read () – is used to obtain a row from the result of the query.

Get String () – is used to access a column value in a selected row.

Close () – must be called after the use of Data Reader Objects.

Example:

> Dim conn As Sqlconnection
>
> Dim comm As Sqlcommand
>
> Dim rd As SqlDataReader

```
conn = New Sqlconnection ()

conn = New Sqlconnection ()

conn.connection string = "Datasource = asdf; Initial catalog =college; UserId =sa; pwd
```
=123"
```
conn.open ()

comm = New Sqlcommand ()

comm.connection = conn

comm.commandText = "select * from staff"

rd = comm.ExecuteReader
```

Sample C # Program for Connection Environment:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;
namespace ConsoleApplication1
{
class Program
{
 static void Main(string[] args)
{
string connectionString = "Data Source=CENTRE-I-62;Initial Catalog=management;Integrated
Security=True";
string query1 = "select * from employee";
SqlConnection cn = new SqlConnection(connectionString);
cn.Open();
SqlCommand cmd1 = new SqlCommand(query1, cn);
SqlDataReader dr1 = cmd1.ExecuteReader();
while (dr1.Read())
```

```
{
Console.WriteLine(dr1[0] + " " + dr1[1] + " " + dr1[2] + " " + dr1[3] + " " + dr1[4]);
}
cn.Close();
Console.ReadLine();
 }   }   }
```

INDIRECT ACCESS (OR) DATASETS (OR) DISCONNECTED ENVIRONMENT

Dataset is a temporary storage of records. The dataset represents a complete set of data including tables and their relationships.Dataset must contain atleast one data table object in its data table collection.

Each Data Table contains both a

     1. Data Row Collection

     2. Data Column Collection

Objects of Datasets are

     1. Data Table Collection

     2. Data Tables

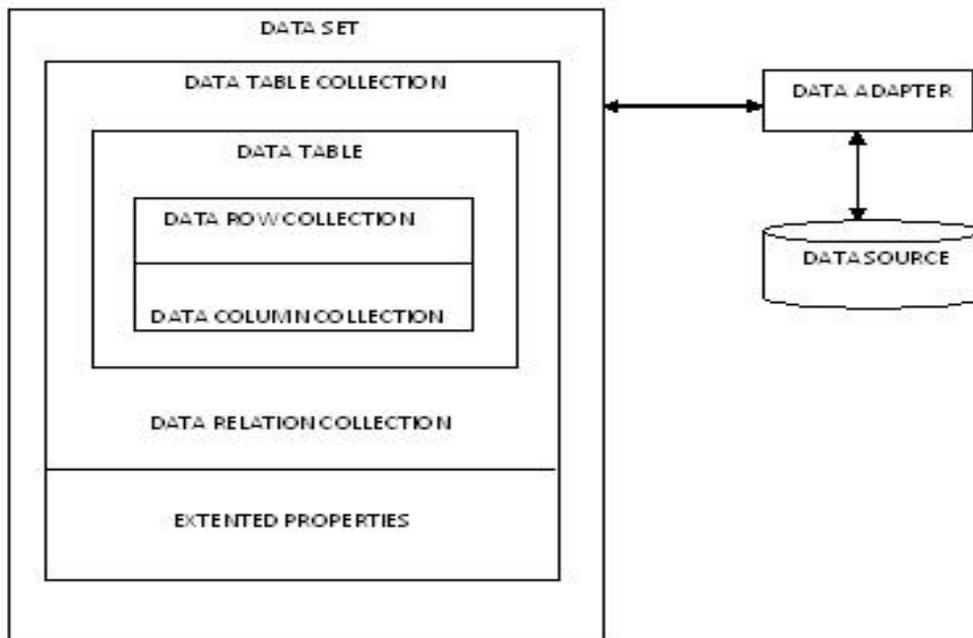   3. Data Row **Data Adapter Objects**

Fig 4.3 Disconnected Environment

SqlDataAdapter OBJECT

This object is used to exchange Data between a database and a Dataset. It reads data from the database to the dataset and then writes the changed data back to the database.

Properties:

Select Command (Accessing Rows in a Database)

Insert Command (Inserting rows in the Database)

Update Command (For modifying rows in the Database)

Delete Command (For deleting rows)

FILL Method:

It is used to write the result of select command into the dataset.

Example:

     Dim Myadapter As New SqlDataAdapter

     Myadapter.select command = mycomm

     Dim Mydataset As New Dataset ()

     Myadapter.Fill (Mydataset, stud Table)

     conn.close ()

Data Table Collection Object:

It is used to provide one or more tables represented by data Table object. It contains collections such as Data Column Collection, Data Row Collection.

Data Table Object:

It contains a single table of memory resident data. It contains a collection of rows represented by Data Row Collection, a collection of columns represented by data Column collection.

Data Row Object:

It contains methods and propeties, to retrieve, insert, delete and update the values in the Data Table.

Add () – Add a new Data Row in the Data Collection.

Remove () – Delete a Data Row from the Data Row Collection

Accept Changes () – Confirm the Addition.

New Row () – Add New Row.

EXAMPLE

POPULATING COMBO BOX WITH SOURCE CITY NAMES DATASET

note:
1. **Create a database with table containing two fields source_id,source_name and enter some city names.**
2. **In c# window form put a combo box and in form load enter the following coding**

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

using System.Data.SqlClient;
namespace WindowsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
```

```csharp
private void Form1_Load(object sender, EventArgs e)
{


    string connetionString = null;
    SqlConnection connection;
    SqlCommand command;
    SqlDataAdapter adapter = new SqlDataAdapter();
    DataSet ds = new DataSet();
    //int i = 0;
    string sql = null;

    connetionString = "Data Source=MIT-LAB2-SYS9;Initial Catalog=source;Persist
Security Info=True;User ID=sa;Password=mit123";
    sql = "select source_id,source_name from source_table";

    connection = new SqlConnection(connetionString);


    connection.Open();
    command = new SqlCommand(sql, connection);
    adapter.SelectCommand = command;
    adapter.Fill(ds);
    adapter.Dispose();
    command.Dispose();
    connection.Close();

    comboBox1.DataSource = ds.Tables[0];
    comboBox1.ValueMember = "source_id";
    comboBox1.DisplayMember = "source_name";


}
}
}
```

CHAPTER V: J2EE

5.1 ENTERPRISE EDITION OVERVIEW

- The Java 2 Platform, Enterprise Edition (J2EE) defines the standard for developing multitier enterprise applications.

- The J2EE platform simplifies enterprise applications by basing them on standardized, modular components, by providing a complete set of services to those components, and by handling many details of application behavior automatically, without complex programming.

- The J2EE platform takes advantage of many features of the Java 2 Platform, Standard Edition (J2SE), such as "Write Once, Run Anywhere" portability, JDBC API for database access, CORBA technology for interaction with existing enterprise resources, and a security model that protects data even in internet applications.

- Building on this base, the Java 2 Platform, Enterprise Edition adds full support for Enterprise JavaBeans components, Java Servlets API, JavaServer Pages and XML technology.

- The J2EE standard includes complete specifications and compliance tests to ensure portability of applications across the wide range of existing enterprise systems capable of supporting the J2EE platform. In addition, the J2EE specification now ensures Web services interoperability through support for the WS-I Basic Profile.

- The Java 2 Enterprise Edition (J2EE) provides a standard for developing multitier, enterprise services.

- J2EE architecture supports component-based development of multi-tier enterprise applications. A J2EE application system typically includes the following tiers:

  - ✓ **Client tier**: In the client tier, Web components, such as Servlets and JavaServer Pages (JSPs), or standalone Java applications provide a dynamic interface to the middle tier.

  - ✓ **Middle tier**: In the server tier, or middle tier, enterprise beans and Web Services encapsulate reusable, distributable business logic for the application. These server-tier components are contained on a J2EE Application Server, which provides the platform for these components to perform actions and store data.

- ✓ **Enterprise data tier**: In the data tier, the enterprise's data is stored and persisted, typically in a relational database.

- J2EE applications are comprised of components, containers, and services. Components are application-level components.
- Web components, such as Servlets and JSPs, provide dynamic responses to requests from a Web page.
- EJB components contain server-side business logic for enterprise applications.
- Web and EJB component containers host services that support Web and EJB modules.

5.1.1. Java 2 Enterprise Edition Architecture

- J2EE uses a 4-level model for web development.
- The browser runs on the client displaying HTML and optionally runs JavaScript.
- The middle tier is comprised of two layers: a Presentation Layer and a Business Logic Layer.
- The data manages persistent data in a database and, where appropriate, legacy data stores.
- J2EE implements the Presentation Layer with Servlets and, more recently, Java provides the option to generate webpages with dynamic content using JavaServer Pages (JSP).
- Servlets/JSP generate webpages with dynamic content (typically originating from the database).
- They also parse webpages submitted from the client and pass them to Enterprise JavaBeans for handling. Servlets and JSPs run inside a Web Server.
- J2EE implements the Business Logic layer with Enterprise JavaBeans (EJB). Enterprise JavaBeans are responsible for logic like validation and calculations as well as provided data access (e.g. database I/O) for the application. Enterprise JavaBeans run inside an Application Sever.
- Under J2EE, EJBs access a database through one of two means:
  1. •using a JDBC interface which requires a lower level of coding and/or

2. •using SQLJ which provides a higher level interface to the database

- In addition to these components for web application, J2EE provides for access by non-web clients to the business logic layer. A standalone Java application (IIOP client) can access an EJB directly using J2EE's Remote Method Invocation (RMI) API.

5.1.2 Supporting J2EE APIs

J2EE provides a number of "supporting" APIs. The purpose of most of these APIs is to enable interaction between the "main" software layers/components in the J2EE architecture.

•Remote Method Interface (RMI)

•Java Naming and Directory Interface (JNDI)

•Java Message Service (JMS)

•Java Transaction API (JTA)

•Java Database Connectivity (JDBC) / SQLJ

•JavaMail /JMC

Remote Method Interface (RMI)

- RMI is an important API used for supporting distributed computing and has been supported in core Java since version 1.1. RMI allows a Java client application to communicate with a Java server application by invoking methods on that remote object.
- With RMI, the client gets a reference to a server object and then it can invoke methods on that object as if it were a local object within the same virtual machine.
- For server objects developed in other languages, you must employ other techniques like using Java IDL with CORBA or RMI/IIOP to access the server object.

Java Naming and Directory Interface (JNDI)

- JNDI allows Java programs to use name servers and directory servers to look up objects or data by name. This important feature allows a client object to locate a remote server object or data.

- JNDI is a generic API that can work with any name or directory servers. Server providers have been implemented for many common protocols (e.g. NIS, LDAP and NDS) and for CORBA object registries.
- Of particular interest to users of J2EE, JNDI is used to locate Enterprise JavaBean (EJB) components on the network.

Java Message Service (JMS)

- JMS is an API for using networked messaging services.
- Data sent in a message is often intended as a sort of event notification (e.g. a Email-handling process may need to be notified when a request is enqueued).
- Another common use for messaging (thus JMS) is for interfacing with "external", third party or legacy applications, typically via a Message Oriented Middleware product like IBM's MQ Series (now WebSphere MQ).
- It can be complex/risky to use RPC/RMI to directly invoke remote applications while a messaging solution can provide a simpler and more reliable interconnection.

5.2 Multi-Tier Architecture

- The J2EE platform uses a multitiered distributed application model.
- Application logic is divided into components according to function, and the various application components that make up a J2EE application are installed on different machines depending on the tier in the multitiered J2EE environment to which the application component belongs. Client-tier components run on the client machine.
- J2EE multi-tiered applications are generally considered to be three-tiered applications because they are distributed over three different locations
    - ✓ Client machines
    - ✓ The J2EE server machine
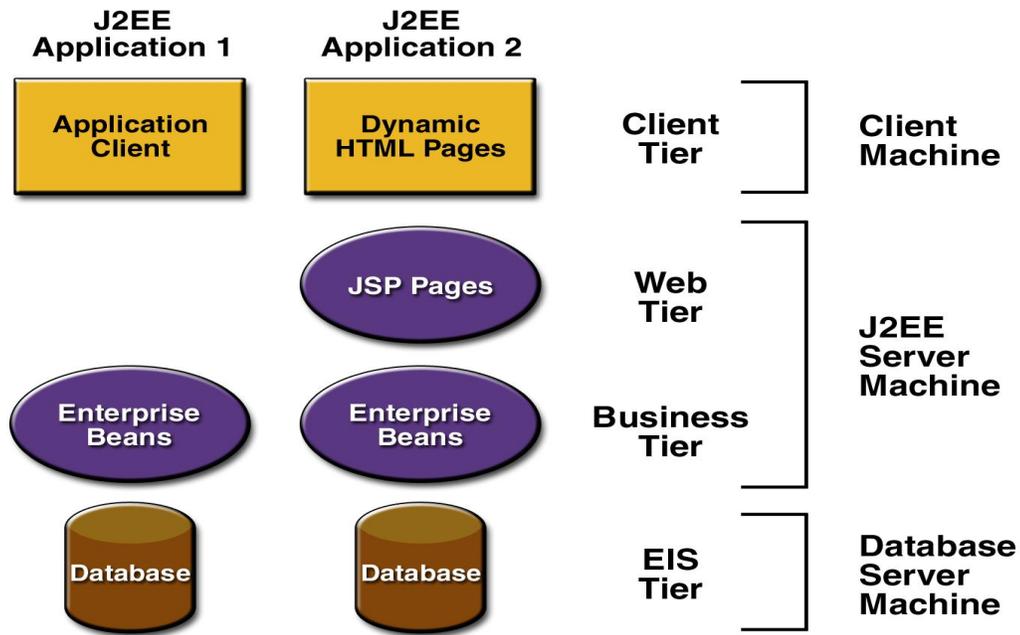   - ✓ The database or legacy machines at the back end

Fig 5.1 Multitiered Applications

- Web-tier components run on the J2EE server.

- Business-tier components run on the J2EE server.

- Enterprise information system (EIS)-tier software runs on the EIS server.

✓ Although a J2EE application can consist of the three or four tiers shown, J2EE multitiered applications are generally considered to be three-tiered applications because they are distributed over three different locations: client machines, the J2EE server machine, and the database or legacy machines at the back end.

✓ Three-tiered applications that run in this way extend the standard two-tiered client and server model by placing a multithreaded application server between the client application and back-end storage.
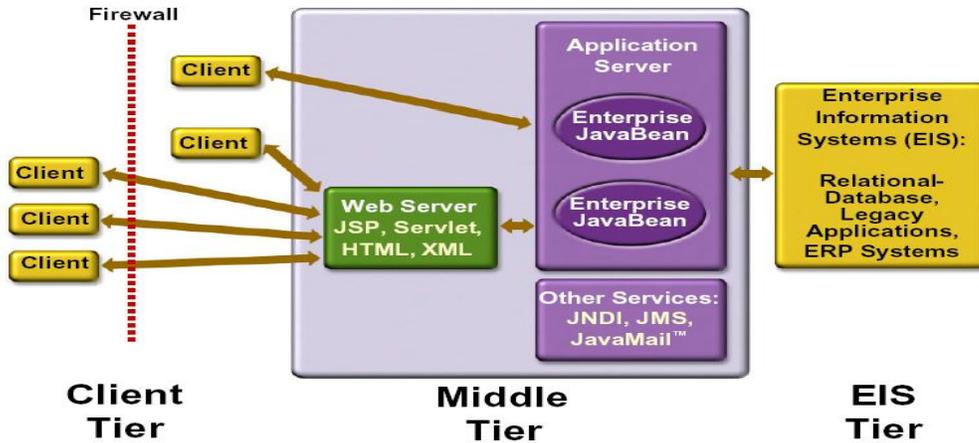
Fig 5.2 J2EE Components

- J2EE applications are made up of components.

- A J2EE component is a self-contained functional software unit that is assembled into a J2EE application with its related classes and files and that communicates with other components.

- The J2EE specification defines the following J2EE components:

    - ✓ Application clients and applets are components that run on the client.

    - ✓ Java Servlet and JavaServer Pages (JSP) technology components are Web components that run on the server.

    - ✓ Enterprise JavaBeans (EJB) components (enterprise beans) are business components that run on the server.

- J2EE components are written in the Java programming language and are compiled in the same way as any program in the language.

- The difference between J2EE components and "standard" Java classes is that J2EE components are assembled into a J2EE application, verified to be well formed and in compliance with the J2EE specification, and deployed to production, where they are run and managed by the J2EE server.

5.2.1 J2EE Clients

- A J2EE client can be a Web client or an application client.

Web Clients

- A Web client consists of two parts: dynamic Web pages containing various types of markup language (HTML, XML, and so on), which are generated by Web components running in the Web tier, and a Web browser, which renders the pages received from the server.

- A Web client is sometimes called a thin client.

- Thin clients usually do not do things like query databases, execute complex business rules, or connect to legacy applications.

- When you use a thin client, heavyweight operations like these are off-loaded to enterprise beans executing on the J2EE server where they can leverage the security, speed, services, and reliability of J2EE server-side technologies.

Applets

- A Web page received from the Web tier can include an embedded applet.

- An applet is a small client application written in the Java programming language that executes in the Java virtual machine installed in the Web browser.

- However, client systems will likely need the Java Plug-in and possibly a security policy file in order for the applet to successfully execute in the Web browser.

Application Clients

- A J2EE application client runs on a client machine and provides a way for users to handle tasks that require a richer user interface than can be provided by a markup language.

- It typically has a graphical user interface (GUI) created from Swing or Abstract Window Toolkit (AWT) APIs, but a command-line interface is certainly possible.

- Application clients directly access enterprise beans running in the business tier. However, if application requirements warrant it, a J2EE application client can open an HTTP connection to establish communication with a servlet running in the Web tier.

5.3 Best Practices in J2ee

A Best Practice is a proven technique for achieving a desired result. It is a repeatable technique that has been successful in real life situations, and can be broadly applied across many problems.

Best practices in two major sections:

1. Best practices in theory.

2. Best practices in practice.

5.3.1 Development Life Cycle

The most critical element of any application development philosophy is the methodology that defines the entire application development cycle. Since methodologies used to make modern J2EE applications are so diverse, we will not endorse any particular methodology.Instead, we will define five relatively generic steps that any significant development method will need to support. The best practices under each can then be integrated into your development cycle.
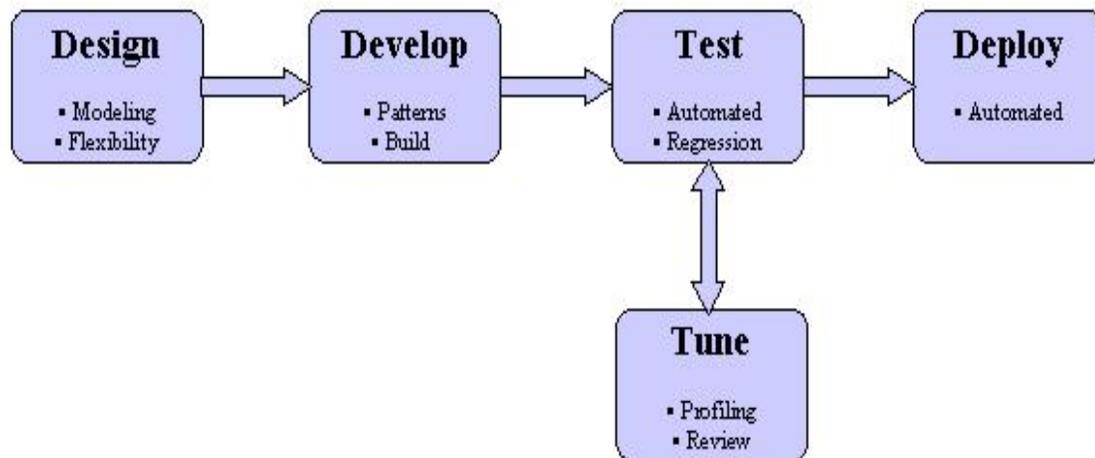

Fig 5.3 Development Life Cycle

The above Fig shows the major steps in a development cycle, and the areas in which Best Practices can be applied for each step.

Development life cycle

Attack risk as early as possible

Design

- ✓ Design for change with dynamic domain model
- ✓ Use a standard modeling language
- ✓ Recycle your resources

Develop

- ✓ Use proven design patterns
- ✓ Automate the build process
- ✓ Integrate often
- ✓ Optimize communication costs

Test

- ✓ Build test cases first
- ✓ Create a testing framework
- ✓ Automate testing

Deploy

- ✓ Use j2ee standard packaging specification
- ✓ Use tools to help in deployment
- ✓ Back up your production data and environment

Tune

- ✓ Build a performance plan
- ✓ Manage memory and plug leaks
- ✓ Focus on priorities

Environments

- ✓ Do not restrict deployment options at design time
- ✓ Create a responsive development environment

5.4 COMPARISON BETWEEN J2EE AND .NET

Both .NET and J2EE target the enormous market for enterprise applications and Web services. This quick-comparison provides a basic sense of advantages and disadvantages carried by both frameworks in various areas.

Orientation

J2EE is Java-centric and platform neutral, while .NET is Windows-centric and language-neutral. This means developers are restricted to Java language in the J2EE and Windows in the .NET framework.

Difference in Strategies

J2EE is basically a series of standards. One the other hand, .NET is Microsoft's product strategy based on evolution of its Visual Studio 6.0.

Industry Involvement

Sun has rallied the entire industry behind its J2EE standard, especially the top software vendors who have adapted the J2EE interface such as BEA, IBM, and Oracle. On the other hand, .NET is based on Microsoft's sole efforts to grab the Web services market share.

Rapid Application Development

- Both .NET and J2EE offer rapid application deployment features in their own way.
- Some are more powerful in .NET compared with J2EE and vice versa. J2EE offers state management services to ease up developers on writing code and managing state.
- On the other hand, the real advantage behind ASP.NET is that it is independent of client device and allows for user interfaces to be rendered to alternative user interfaces without rewriting code.

Pros and Cons of Single Vendor Solutions

- J2EE offers solutions from multiple vendors with a wide variety of tools, products and applications.
- This does provide additional and varied functionality but at a cost

- J2EE tools often times are not interoperable due to problems in portability.

- On the other hand, .NET provides a complete solution of tools and services from Microsoft, but which may lack some of the higher-end features found in J2EE solutions.

Platform Maturity

- Large J2EE vendors such as BEA, Borland, Sybase offer tried and tested solutions to their large customer base.

- J2EE does have certain aspects that are new and immature such as automatic persistence provided by EJB, Java Connector Architecture, and Web services support.

- In the case of Microsoft, some of the .NET is based on Windows DNA but most of it is rewritten as far as the new CLR is concerned.

- Moreover, C# language and Web services support is new and according to many testers, .NET VB is very different from the earlier VB version.

- Hence, we can say that in the overall sense J2EE is the more mature platform.

The Language Factor
- Java programming language is at the center of J2EE.

- All components such as EJB and servlets that are deployed in J2EE framework are also written in Java.

- Although JVM byte code is language-neutral, but in practice this byte code can only be used with Java.

- On the other hand, the .NET framework, based on the new CLR, enables development in any language that is supported by Microsoft's tools.

- The pro of this multi-language support is that a single .NET component can be written in several languages.

- As far as the cons is concerned, first you need experts in different languages to fully develop, debug and maintain a particular application written in multiple languages.

- it requires a significant cost in training and retaining knowledgeable developers.

The Portability Factor

- Since Java Runtime Environment (JRE) is available on any platform -- Win 32, Unix, Mainframe -- there is no doubt about easy and effective J2EE portability.

- In the .NET case, the portability aspect is not promising since industry analysts are skeptical of Microsoft's claims regarding Multiple platform support based on the company's lackluster performance in meeting industry expectations.

Web Services Support

- J2EE enables eBusiness collaboration and Web services through its JAXP (Java API for XML Parsing).
- .NET does create Web services but due to its beta release it does not earn much credibility as a realistic deployment platform.
- Moreover, .NET's lack of support for ebXML poses a real problem for its industry-wide acceptability.

Tools

- Current IDEs for Java development include WebGain's Visual Cafe, IBM's VisualAge for Java, Borland's JBuilder, etc.
- These tools are not entirely interoperable because they do not originate from a single vendor.
- On the other hand, Microsoft's Visual Studio .NET has remarkable productivity features that include Web forms, .NET's GUI component set, and much more.
- The aspect of single-vendor integration, wizards, and other user-friendly features are highly advantageous for building Web services.

Hardware Costs

Both .NET and J2EE support Win32 platform, a less expensive alternative to Unix and Mainframe systems. In general, Microsoft's solution has an aggressive price comparative to J2EE that offers varied price and service levels from high-end to low-end inexpensive solutions.

Performance and Developer Empowerment

J2EE is an ideal platform for educated and knowledgeable developers to leverage the programming control over lower-level services such as state management and caching. In comparison, .NET architecture lacks lower-level performance improvement services, blocks

introduction of errors into the system and is thus more suitable for developers who require more hand-holding.

5.4.1 The Features Of .Net And J2ee

Table 5.1 Comparison of .net and J2ee Features

Feature	.NET Implementation	J2EE Implementation
Languages supported	VB.NET, C#, C++, J#, others	Java
Development environment	Visual Studio.NET	Java Studio Standard 5 update 1, IBM WebSphere, others.
Runtime environment	.NET Framework	Java Runtime Environment (JRE)
First compilation output from source (both systems use Just In Time (JIT) compilation at execution)	Microsoft Intermediate Language (MSIL) and Bootstrap code in a windows executable (.EXE)	Bytecode (.CLASS)
Execution management	Common Language Runtime (CLR)	Java Virtual Machine (JVM)
Support library	Framework Class Library (FCL)	Java API
Database support	ADO.NET both SQLServer and OleDb objects	JDBC
Web support	ASP.NET	Java Servlet JavaServer Pages (JSP) Enterprise JavaBeans
Desktop support	Windows Forms	SWING and the AWT
Web Services	Integrated into Visual Studio.NET using UDDI, WSDL, SOAP, and XML	Java API for XML Processing (JAXP) Java API for XML Registries (JAXR) Java API for XML-based RPC (JAX-RPC) SOAP with Attachments API for Java (SAAJ)