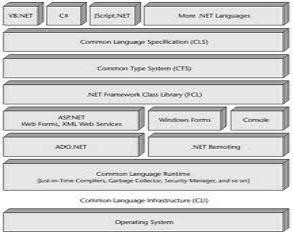
###### ABOUT THE SOFTWARE WHAT IS “.NET”?

Microsoft .net is a set of micro soft software technologies for rapidly building and integrating xml web services, micro soft windows-based applications, and web solutions. The .net framework is a language-neutral platform for writing programs that can easily and securely interoperate. There’s no language barrier with .net: there are numerous languages available to the developer including managed c++, c#, visual basic and java script.

The .net framework provides the foundation for components to interact seamlessly, whether locally or remotely on different platforms. It standardizes common data types and communications protocols so that components created in different languages can easily interoperate. “.net” is also the collective name given to various software components built upon the .net platform. These will be both products (visual studio.net and windows.net server, for instance) and services (like passport, .net my services, and so on).

###### The .NET Framework

Microsoft designed C# from the ground up to take advantage of its new .NET Framework. Because C# is a player in this new .NET world, you should have a good understanding of what the .NET Framework provides and how it increases your productivity.



The .NET Framework is made up of four parts, as shown in the Common Language Runtime, a set of class libraries, a set of programming languages, and the ASP.NET environment. The .NET Framework was designed with three goals in mind. First, it was intended to make Windows applications much more reliable, while also providing an application with a greater degree of security. Second, it was intended to simplify the development of Web applications and services that not only work in the traditional sense, but on mobile devices as well. Lastly, the framework was designed to provide a single set of libraries that would work with multiple languages. The following sections examine each of the .NET Framework components.

###### Web Development:

The .NET Framework was designed with one thing in mind: to fuel Internet development. This new fuel to add to Internet development is called *Web Services*. You can think of Web Services as a Web site that interacts with programs, rather than people. Instead of delivering Web pages, a Web Service takes a request formatted as XML, performs a particular function, and then returns a response to the requester as an XML message.

Note XML or eXtensible Markup Language is a self describing language much like that of HTML. XML on the other hand has no predefined tags thus allowing it great flexibility in representing a wide variety of objects.

A typical application for a Web Service would be to sit as a layer on top of a corporate billing system. When a user surfing the Web purchases products from your Internet site, the purchase information is then sent to the Web Services, which totals all the products, adds a record to the accounts receivable database, and then returns a response with an order confirmation number. Not only can this Web Service interact with Web pages, it can interact with other Web Services, such as a corporate accounts payable system.

In order for the Web Service model to survive the natural evolution of programming languages, it must include much more than a simple interface to the Web. The Web service model also includes protocols that enable applications to find Web Services available across a LAN or the Internet. This protocol also enables the application to explore the Web Service and determine how to communicate with it, as well as how to exchange information. To enable Web Service discovery, the Universal Discovery, Description and Integration (UDDI) was established. This allows Web Services to be registered and searched, based on key information such as company name, type of service, and geographic location.

###### Application Development

Aside from Web development, you can still build traditional Windows applications with the .NET Framework. Windows applications created with the .NET Framework are based upon *Windows Forms*. These Windows Forms are somewhat of a crossbreed between Visual Basic 6 forms and the forms of Visual C++. Though forms look the same as their predecessors, they are completely object-oriented and class-based, much like form objects in the Microsoft Foundation Class. These new Windows Forms now support many classic controls found in Visual Studio, such as the Button, TextBox, and Label, as well as ActiveX controls. Aside from the traditional controls, new components such as PrintPreview, LinkLabel, ColorDialog, and OpenFileDialog are also supported.

Building applications with .NET also provides you with many enhancements not found in other languages, such as security. These security measures can determine whether an application can write or read a disk file. They also enable you to embed digital signatures into

the application to ensure that the application was written by a trusted source. The .NET Framework also enables you to embed component information, and version information, within the actual code. This makes it possible for software to install on demand, automatically, or with no user intervention at all. Together, all of these features greatly reduce support costs within the enterprise.

###### Common Language Runtime

Programming languages usually consist of both a compiler and a runtime environment. The compiler turns the code that you write into executable code that can be run by users. The runtime environment provides a set of operating system services to your executable code. These services are built into a runtime layer so that your code does not need to worry about the low-level details of working with the operating system. Operations such as memory management and file I/O are good examples of services that might be provided by a runtime environment. Before .NET came along, each language shipped with its own runtime environment. Visual Basic shipped with a runtime called MSVBVM60.DLL. Visual C++ shipped with a DLL called MSVCRT.DLL. Each of these runtime modules provided a set of low-level services to code that developers wrote. Developers would write code and then build that code with the appropriate runtime in mind. The executable code would ship with the runtime, which would be installed on a user's machine if it weren't already present.

The main problem with these runtime environments is that they were designed for use with a single language. The Visual Basic runtime provided nice features for operations like working with memory and launching COM objects, but these features were only available to Visual Basic users. Developers using Visual C++ could not use the features of the Visual Basic runtime. Visual C++ users had their own runtime, with its own long list of features, but those features were unavailable to Visual Basic users. This "separate runtime" approach prevented languages from working together seamlessly.

It's not possible, for example, to grab some memory in a piece of Visual Basic code and then hand it off to a piece of Visual C++ code, which frees the memory. The different runtimes implement their own feature set in their own way. The feature sets of the various runtimes are

inconsistent. Even features that are found in more than one runtime are implemented in different ways, making it impossible for two pieces of code written in different languages to work together.

One of the design goals of the .NET Framework was to unify the runtime engines so that all developers could work with a single set of runtime services. The .NET Framework's solution is called the *Common Language Runtime (CLR)*. The CLR provides capabilities such as memory management, security, and robust error-handling to any language that works with the .NET Framework. Thanks to the CLR, all .NET languages can use a variety of runtime services without developers worrying about whether their particular language supports a runtime feature.

The CLR also enables languages to interoperate with one another. Memory can be allocated by code written in one language — Visual Basic .NET, for instance — and can be freed by code written in another language, say, C#. Similarly, errors can be raised in one language and processed in another language.

###### .NET Class Libraries

Developers like to work with code that has already been tested and shown to work, such as the Win32 API and the MFC Class libraries. Code re-use has long been the goal of the software development community. However, the practicality of code re-use has not lived up to expectations. Many languages have had access to bodies of pre-tested, ready-to-run code. Visual C++ has benefited from class libraries such as the Microsoft Foundation Classes (MFC), which enabled C++ developers to build Windows applications quickly, and the Active Template Library (ATL), which provided support for building COM objects. However, the languagespecific nature of these libraries has made them unavailable for use in other languages.

Visual Basic developers are locked out of using ATL when building their COM objects. The .NET Framework provides many classes that help developers re-use code. The .NET class libraries contain code for programming topics such as threading, file I/O, database support, XML parsing, and data structures, such as stacks and queues. Best of all, this entire class library is available to any programming language that supports the .NET Framework. Thanks to the CLR, any .NET language can use any class in the .NET class library. Because all languages now support the same runtime, they can re-use any class that works with the .NET Framework. This

means that any functionality available to one language will also be available to any other .NET language.

The class library re-use picture painted by the .NET Framework gets even better when you realize that re-use extends to your code, not just code that Microsoft ships with .NET. The code that Microsoft ships in the .NET class library code base is architecturally no different from the code you write. The Microsoft code is simply code that was written using a language supported by .NET and built using a .NET development tool. This means that Microsoft is using the same tools that you will use to write your code. You can write code that can be used in other .NET languages, just as Microsoft has with its class library. The .NET Framework enables you to write code in C#, for example, and hand it off to Visual Basic .NET developers, who can use your compiled code in their applications.

###### .NET Programming Language

The .NET Framework provides a set of tools that help you build code that works with the

.NET Framework. Microsoft provides a set of languages that are already ".NET-compatible". C# is one of those languages. New versions of Visual Basic and Visual C++ have also been created to take advantage of the .NET Framework, with a version of Jscript.NET on the way. The development of .NET-compatible languages is not restricted to Microsoft. The .NET group at Microsoft has published documentation showing how language vendors can make their languages work with .NET, and vendors are making languages such as COBOL and Perl compatible with the .NET Framework. There are currently 20 or more languages in the works from third party vendors and institutions that plug into the .NET Framework.

###### Introducing C#

C#, the new language introduced in the .NET Framework, is derived from C++. However, C# is a modern, objected-oriented (from the ground up) type-safe language.

###### Language features

The following sections take a quick look at some of the features of the C# language. If some of these concepts don't sound familiar to you, don't worry. All of them are covered in detail in later chapters.

###### Classes

All code and data in C# must be enclosed in a class. You can't define a variable outside of a class, and you can't write any code that's not in a class. Classes can have *constructors,* which execute when an object of the class is created, and a *destructor,* which executes when an object of the class is destroyed. Classes support single inheritance, and all classes ultimately derive from a base class called *object*. C# supports versioning techniques to help your classes evolve over time while maintaining compatibility with code that uses earlier versions of your classes.

###### Data types

C# lets you work with two types of data: value types and reference types. *Value types* hold actual values. *Reference types* hold references to values stored elsewhere in memory. Primitive types such as char, int and float, as well as enumerated values and structures, are value types. Reference types hold variables that deal with objects and arrays. C# comes with predefined reference types (object and string), as well as predefined value types (sbyte, short, int, long, byte, ushort, uint, ulong, float, double, bool, char, and decimal). You can also define your own value and reference types in your code. All value and reference types ultimately derive from a base type called object.

C# allows you to convert a value of one type into a value of another type. You can work with both *implicit* conversions and *explicit* conversions. Implicit conversions always succeed and don't lose any information (for example, you can convert an int to a long without losing any data because a long is larger than an int). Explicit conversions may cause you to lose data (for example, converting a long into an int may result in a loss of data because a long can hold larger values than an int). You must write a cast operator into your code to make an explicit conversion happen.

You can work with both one-dimensional and multidimensional arrays in C#. Multidimensional arrays can be rectangular, in which each of the arrays has the same dimensions,

or jagged, in which each of the arrays has different dimensions. Classes and structures can have data members called *properties* and *fields*. *Fields* are variables that are associated with the enclosing class or structure. You may define a structure called Employee, for example, that has a field called Name. If you define a variable of type Employee called CurrentEmployee, you can retrieve the employee's name by writing CurrentEmployee.Name. *Properties* are like fields, but enable you to write code to specify what should happen when code accesses the value. If the employee's name must be read from a database, for example, you can write code that says, "when someone asks for the value of the Name property, read the name from the database and return the name as a string."

###### Functions

A function is a callable piece of code that may or may not return a value to the code that originally called it. An example of a function would be the FullName function shown earlier, in this chapter, in the Family class. A *function* is generally associated to pieces of code that return information whereas a *method* generally does not return information. For our purposes however, we generalize and refer to them both as functions. Functions can have four kinds of parameters:

* Input parameters have values that are sent into the function, but the function cannot change those values.
* Output parameters have no value when they are sent into the function, but the function can give them a value and send the value back to the caller.
* Reference parameters pass in a reference to another value. They have a value coming in to the function, and that value can be changed inside the function.
* Params parameters define a variable number of arguments in a list.

C# and the CLR work together to provide automatic memory management. You don't need to write code that says "allocate enough memory for an integer" or "free the memory that this object was using." The CLR monitors your memory usage and automatically retrieves more when you need it. It also frees memory automatically when it detects that it is no longer being used (this is also known as Garbage Collection). C# provides a variety of operators that enable

you to write mathematical and bitwise expressions. Many (but not all) of these operators can be redefined, enabling you to change how the operators work.

C# supports a long list of statements that enable you to define various execution paths within your code. Flow control statements that use keywords such as if, switch, while, for, break and continue enable your code to branch off into different paths, depending on the values of your variables. Classes can contain code and data. Each class member has something called an *accessibility scope,* which defines the member's visibility to other objects. C# supports public, protected, internal, protected internal, and private accessibility scopes.

###### Variables

Variables can be defined as constants. *Constants* have values that cannot change during the execution of your code. The value of pi, for instance, is a good example of a constant, because its value won't be changing as your code runs. *Enum type declarations* specify a type name for a related group of constants. For example, you could define an enum of Planets with values of Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune and Pluto, and use those names in your code. Using the enum names in code makes code more readable than if you used a number to represent each planet.

C# provides a built-in mechanism for defining and handling events. If you write a class that performs a lengthy operation, you may want to invoke an event when the operation is completed. Clients can subscribe to that event and catch the event in their code, which enables them to be notified when you have completed your lengthy operation. The event handling mechanism in C# uses *delegates*, which are variables that reference a function. Note An event handler is a procedure in your code that determines the actions to be performed when an event occurs, such as the user clicking a button.

If your class holds a set of values, clients may want to access the values as if your class were an array. You can write a piece of code called an *indexer* to enable your class to be accessed as if it were an array. Suppose you write a class called Rainbow, for example, that

contains a set of the colors in the rainbow. Callers may want to write MyRainbow[0] to retrieve the first color in the rainbow. You can write an indexer into your Rainbow class to define what should be returned when the caller accesses your class, as if it were an array of values.

###### Interfaces

C# supports *interfaces*, which are groups of properties, methods, and events that specify a set of functionality. C# classes can implement interfaces, which tells users that the class supports the set of functionality documented by the interface. You can develop implementations of interfaces without interfering with any existing code, which minimizes compatibility problems. nce an interface has been published, it cannot be changed, but it can evolve through inheritance.

C# classes can implement many interfaces, although the classes can only inherit from a single base class.Let's look at a real-world example that would benefit from interfaces to illustrate its extremely positive role in C#. Many applications available today support add-ins. Assume that you have created a code editor for writing applications. This code editor, when executed, has the capability to load add-ins. To do this, the add-in must follow a few rules. The DLL add-in must export a function called CEEntry, and the name of the DLL must begin with CEd. When we run our code editor, it scans its working directory for all DLLs that begin with CEd. When it finds one, it is loaded; and then it uses the GetProcAddress to locate the CEEntry functionwithin the DLL, thus verifying that you followed all the rules necessary to create an add- in.

This method of creating and loading add-ins is very burdensome because it burdens the code editor with more verification duties than necessary. If an interface were used in this instance, your add-in DLL could have implemented an interface, thus guaranteeing that all necessary methods, properties, and events were present with the DLL itself, and functioning as documentation specified.

###### Attributes

Attributes declare additional information about your class to the CLR. In the past, if you wanted to make your class self-describing, you had to take a disconnected approach in which the documentation was stored in external files such as IDL or even HTML files. Attributes solve this

problem by enabling you, the developer, to bind information to classes — any kind of information. For example, you can use an attribute to embed documentation information into a class. Attributes can also be used to bind runtime information to a class, defining how it should act when used. The possibilities are endless, which is why Microsoft includes many predefined attributes within the .NET Framework.

###### Compiling C#

Running your C# code through the C# compiler produces two important pieces of information: code and metadata. The following sections describe these two items and then finish up by examining the binary building block of .NET code: the assembly.

###### Microsoft Intermediate Language (MSIL)

The code that is output by the C# compiler is written in a language called Microsoft Intermediate Language, or MSIL. MSIL is made up of a specific set of instructions that specify how your code should be executed. It contains instructions for operations such as variable initialization, calling object methods, and error handling, just to name a few. C# is not the only language in which source code changes into MSIL during the compilation process. All .NET- compatible languages, including Visual Basic .NET and Managed C++, produce MSIL when their source code is compiled. Because all of the .NET languages compile to the same MSIL instruction set, and because all of the .NET languages use the same runtime, code from different languages and different compilers can work together easily.

MSIL is not a specific instruction set for a physical CPU. It knows nothing about the CPU in your machine, and your machine knows nothing about MSIL. How, then, does your .NET code run at all, if your CPU can't read MSIL? The answer is that the MSIL code is turned into CPU-specific code when the code is run for the first time. This process is called "just- in-time" compilation, or JIT. The job of a JIT compiler is to translate your generic MSIL code into

machine code that can be executed by your CPU. You may be wondering about what seems like an extra step in the process. Why generate MSIL when a compiler could generate CPU-specific code directly? After all, compilers have always done this in the past.

There are a couple of reasons for this. First, MSIL enables your compiled code to be easily moved to different hardware. Suppose you've written some C# code and you'd like it to run on both your desktop and a handheld device. It's very likely that those two devices have different types of CPUs. If you only had a C# compiler that targeted a specific CPU, then you'd need two C# compilers: one that targeted your desktop CPU and another that targeted your handheld CPU. You'd have to compile your code twice, ensuring that you put the right code on the right device. With MSIL, you compile once. Installing the .NET Framework on your desktop machine includes a JIT compiler that translates your MSIL into CPU-specific code for your desktop.

Installing the .NET Framework on your handheld includes a JIT compiler that translates that same MSIL into CPU-specific code for your handheld. You now have a single MSIL code base that can run on any device that has a .NET JIT compiler. The JIT compiler on that device takes care of making your code run on the device.

Another reason for the compiler's use of MSIL is that the instruction set can be easily read by a verification process. Part of the job of the JIT compiler is to verify your code to ensure that it is as clean as possible. The verification process ensures that your code is accessing memory properly and that it is using the correct variable types when calling methods that expect a specific type. These checks ensure that your code doesn't execute any instructions that could make the code crash.

The MSIL instruction set was designed to make this verification process relatively straightforward. CPU-specific instruction sets are optimized for quick execution of the code, but they produce code that can be hard to read and, therefore, hard to verify. Having a C# compiler that directly outputs CPU-specific code can make code verification difficult or even impossible. Allowing the .NET Framework JIT compiler to verify your code ensures that your code accesses memory in a bug-free way and that variable types are properly used.

###### Metadata

The compilation process also outputs metadata, which is an important piece of the .NET codesharing story. Whether you use C# to build an end-user application or you use C# to build a class library to be used by someone else's application, you're going to want to make use of some already-compiled .NET code. That code may be supplied by Microsoft as a part of the .NET Framework, or it may be supplied by a user over the Internet. The key to using this external code is letting the C# compiler know what classes and variables are in the other code base so that it can match up the source code you write with the code found in the precompiled code base that you're working with.

Think of metadata as a "table of contents" for your compiled code. The C# compiler places metadata in the compiled code along with the generated MSIL. This metadata accurately describes all the classes you wrote and how they are structured. All of the classes' methods and variable information is fully described in the metadata, ready to be read by other applications. Visual Basic .NET, for example, may read the metadata for a .NET library to provide the IntelliSense capability of listing all of the methods available for a particular class. If you've ever worked with COM (Component Object Model), you may be familiar with *type libraries.* Type libraries aimed to provide similar "table of contents" functionality for COM objects.

However, type libraries suffered from some limitations, not the least of which was the fact that not all of the data relevant to the object was put into the type library. Metadata in .NET does not have this shortcoming. All of the information needed to describe a class in code is placed into the metadata. You can think of metadata as having all of the benefits of COM type libraries without the limitations.

###### Assemblies

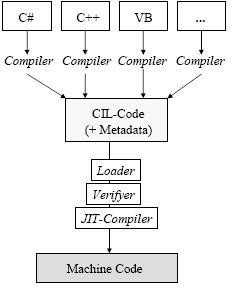
Sometimes, you will use C# to build an end-user application. These applications are packaged as executable files with an extension of .EXE. Windows has always worked with .EXE files as application programs, and C# fully supports building .EXE files. However, there may be times when you don't want to build an entire application. Instead, you may want to build a code library that can be used by others. You may also want to build some utility classes in C#, for example, and then hand the code off to a Visual Basic .NET developer, who will use your classes

in a Visual Basic .NET application. In cases like this, you won't be building an application. Instead, you'll be building an *assembly*.

An assembly is a package of code and metadata. When you deploy a set of classes in an assembly, you are deploying the classes as a unit; and those classes share the same level of version control, security information, and activation requirements. Think of an assembly as a "logical DLL." If you're familiar with Microsoft Transaction Server or COM+, you can think of an assembly as the .NET equivalent of a package.

There are two types of assemblies: *private assemblies* and *global assemblies*. When you build your assembly, you don't need to specify whether you want to build a private or a global assembly. The difference is apparent when you deploy your assembly. With a private assembly, you make your code available to a single application. Your assembly is packaged as a DLL, and is installed into the same directory as the application using it. With a deployment of a private assembly, the only application that can use your code is the executable that lives in the same directory as your assembly.

If you want to share your code among many applications, you might want to consider deploying your code as a global assembly. Global assemblies can be used by any .NET application on the system, regardless of the directory in which it is installed. Microsoft ships assemblies as a part of the .NET Framework, and each of the Microsoft assemblies is installed as a global assembly. The .NET Framework contains a list of global assemblies in a facility called the *global assembly cache,* and the .NET Microsoft Framework SDK includes utilities to both install and remove assemblies from the global assembly cache.



***SQL Server 2005***

SQL Server 2005 is the successor to SQL Server 2000. It included native support for managing XML data, in addition to relational data. For this purpose, it defined an xml data type that could be used either as a data type in database columns or as literals in queries. XML columns can be associated with XSD schemas. XML data being stored is verified against the schema. XML is converted to an internal binary data type before being stored in the database.

Specialized indexing methods were made available for XML data. XML data is queried using XQuery. Common Language Runtime (CLR) integration was a main feature with this edition, enabling one to write SQL code as Managed Code by the CLR.

SQL Server 2005 added some extensions to the T-SQL language to allow embedding XQuery queries in T-SQL. In addition, it also defines a new extension to XQuery, called XML DML that allows query-based modifications to XML data.

SQL Server 2005 also allows a database server to be exposed over web services using Tabular Data Stream (TDS) packets encapsulated within SOAP (protocol) requests. When the data is accessed over web services, results are returned as XML.For relational data, T-SQL has been augmented with error handling features (try/catch) and support for recursive queries with CTEs (Common Table Expressions). SQL Server 2005 has also been enhanced with new indexing algorithms, syntax and better error recovery systems.

Data pages are checksummed for better error resiliency, and optimistic concurrency support has been added for better performance. Permissions and access control have been made more granular and the query processor handles concurrent execution of queries in a more efficient way. Partitions on tables and indexes are supported natively, so scaling out a database onto a cluster is easier. SQL CLR was introduced with SQL Server 2005 to let it integrate with the .NET Framework.

SQL Server 2005 introduced "MARS" (Multiple Active Results Sets), a method of allowing usage of database connections for multiple purposes.SQL Server 2005 introduced DMVs (Dynamic Management Views), which are specialized views and functions that return server state information that can be used to monitor the health of a server instance, diagnose problems, and tune performance.

##### SQL Server Configuration Manager

SQL Server Configuration Manager is a new tool in SQL Server 2005. It is used to manage SQL Server 2005 services and connections. It has been developed as a Microsoft Management Console (MMC) plug-in application Its window is divided into a Console tree (left pane) and a Details pane. It can be managed through SQL Server services and connection configurations by navigating objects in the Console tree.

##### Managing Services

* + SQL Server
  + SQL Server Agent
  + SQL Server Browser
  + SQL Server Integration Services
  + SQL Server Anaylysis Services

##### SQL Server Management Studio

This tool is a new feature in SQL Server 2005. It replaces Enterprise Manager and Query Analyzer from earlier versions. It has been developed using a Visual Studio shell as a base. It follows the paradigm of Visual Studio, in which most tools are organized as tabbed, dockable, or floating windows.

The registered server’s pane allows viewing and managing parameters for connecting to servers. The tool includes both script editors and graphical tools which work with objects and features of the server.A central feature of SQL Server Management Studio is the Object Explorer, which allows the user to browse, select, and act upon any of the objects within the server.

##### SQL Server 2005 Features

* Database mirroring
* **T-SQL (Transaction SQL) enhancements**
* CLR integration
* Service Broker
* DDL triggers
* Ranking functions
* Row versioning-based isolation levels
* XML integration
* TRY...CATCH
* Database Mail

###### Data encryption

**Ease Of Installation, Deployment, And Use**

**SQL Server includes a set of administrative and development tools that improve ability to install, deploy, manage, and use SQL Server across several sites.**

**Scalability**

The same database engine can be used across platforms ranging from laptop computers running Microsoft Windows® 95/98 to large, multiprocessor servers running Microsoft Windows NT®, Enterprise Edition.

**Data Warehousing**

SQL Server includes tools for extracting and analyzing summary data for online analytical processing (OLAP). SQL Server also includes tools for visually designing databases and analyzing data using English-based questions. In this project SQL server is used because of the above features.