**Lecture # 1:**

**Lesson Summary:**

This lesson covers the dot net framework and its components in detail, how they works. What are different versions of dot net framework and what are the differences between them.

**The Evolution of C#**

Since its original 1.0 (2001) release, C# has been evolving at a rapid pace. Not long after C# 1.0, Microsoft released version 1.1 (2003). It contained many minor tweaks but added no major features. However, the situation was much different with the release of C# 2.0 (2005). C# 2.0 was a watershed event in the lifecycle of C# because it added many new features, such as generics, partial types, and anonymous methods, that fundamentally expanded the scope, power, and range of the language. Version 2.0 firmly put C# at the forefront of computer language development. It also demonstrated Microsoft’s long-term commitment to the language. The next major release of C# was 3.0 (2007). Because of the many new features added by C# 2.0, one might have expected the development of C# to slow a bit, just to let programmers catch up, but this was not the case. With the release of C# 3.0, Microsoft once again put C# on the cutting edge of language design, this time adding a set of innovative features that redefined the programming landscape. Here is a list of what 3.0 has added to the language:

• Anonymous types

• Auto-implemented properties

 • Extension methods

• Implicitly typed variables

• Lambda expressions

• Object and collection initializers

• Partial methods

New feature in C# 3.5 is LINQ.Although all of these features are important and have significant impact on the language, the two that are the most exciting are language-integrated query (LINQ) and lambda expressions. LINQ enables you to write database-style queries using C# programming elements. However, the LINQ syntax is not limited to only databases. It can also be used with arrays and collections. Thus, LINQ offers a new way to approach several common programming tasks. Lambda expressions are often used in LINQ expressions, but can also be used elsewhere. They implement a functional-style syntax that uses the lambda operator =>. Together, LINQ and lambda expressions add an entirely new dimension to C# programming. Throughout the course of this book, you will see how these features are revolutionizing the way that C# code is written.

**What Is the .NET Framework?**

The .NET Framework defines an environment that supports the development and execution of highly distributed, component-based applications. It enables differing computer languages to work together and provides for security, program portability, and a common programming model for the Windows platform. As it relates to C#, the .NET Framework defines two very important entities. The first is the Common Language Runtime (CLR). This is the system that manages the execution of your program. Along with other benefits, the Common Language Runtime is the part of the .NET Framework that enables programs to be portable, supports mixed-language programming, and provides for secure execution. The second entity is the .NET class library. This library gives your program access to the runtime environment. For example, if you want to perform I/O, such as displaying something on the screen, you will use the .NET class library to do it. If you are new to programming, then the term class may be new. Although it is explained in detail later in this book, for now a brief definition will suffice: a class is an object-oriented construct that helps organize programs. As long as your program restricts itself to the features defined by the .NET class library, your programs can run anywhere that the .NET runtime system is supported. Since C# automatically uses the .NET Framework class library, C# programs are automatically portable to all .NET environments.

**How the Common Language Runtime Works**

The Common Language Runtime manages the execution of .NET code. Here is how it works: When you compile a C# program, the output of the compiler is not executable code. Instead, it is a file that contains a special type of pseudocode called Microsoft Intermediate Language (MSIL). MSIL defines a set of portable instructions that are independent of any specific CPU. In essence, MSIL defines a portable assembly language. One other point: although MSIL is similar in concept to Java’s bytecode, the two are not the same. It is the job of the CLR to translate the intermediate code into executable code when a program is run. Thus, any program compiled to MSIL can be run in any environment for which the CLR is implemented. This is part of how the .NET Framework achieves portability. Microsoft Intermediate Language is turned into executable code using a JIT compiler. “JIT” stands for “Just-In-Time.” The process works like this: When a .NET program is executed, the CLR activates the JIT compiler. The JIT compiler converts MSIL into native code on demand as each part of your program is needed. Thus, your C# program actually executes as native code even though it is initially compiled into MSIL. This means that your program runs nearly as fast as it would if it had been compiled to native code in the first place, but it gains the portability benefits of MSIL. In addition to MSIL, one other thing is output when you compile a C# program: metadata. Metadata describes the data used by your program and enables your code to interact easily with other code. The metadata is contained in the same file as the MSIL.

**Managed vs. Unmanaged Code**

In general, when you write a C# program, you are creating what is called managed code. Managed code is executed under the control of the Common Language Runtime as just described. Because it is running under the control of the CLR, managed code is subject to certain constraints—and derives several benefits. The constraints are easily described and met: the compiler must produce an MSIL file targeted for the CLR (which C# does) and use the .NET class library (which C# does). The benefits of managed code are many, including modern memory management, the ability to mix languages, better security, support for version control, and a clean way for software components to interact. The opposite of managed code is unmanaged code. Unmanaged code does not execute under the Common Language Runtime. Thus, all Windows programs prior to the creation of the .NET Framework use unmanaged code. It is possible for managed code and unmanaged code to work together, so the fact that C# generates managed code does not restrict its ability to operate in conjunction with preexisting programs.

**The Common Language Specification**

Although all managed code gains the benefits provided by the CLR, if your code will be used by other programs written in different languages, then for maximum usability, it should adhere to the Common Language Specification (CLS). The CLS describes a set of features that different .NET-compatible languages have in common. CLS compliance is especially important when creating software components that will be used by other languages. The CLS includes a subset of the Common Type System (CTS). The CTS defines the rules concerning data types. Of course, C# supports both the CLS and the CTS.