Adrienne Berkowitz
& Associates

Advanced Marketing • Communication • Skills Development Training

Sepelang
Training & Consulting

INTRODUCTION TO VB.NET MANUAL

education

Department:
Education
REPUBLIC OF SOUTH AFRICA

Table of Contents

Developers School for Learning VB.Net

What is all of this?

This is a kind of interactive learning platform where those who want to learn .Net with VB.Net (Visual Basic.Net) can find help and support. The manual is divided into chapters each describing some areas of the VB Programming Language with the Microsoft.Net Platform. This is not a traditional passive tutorial where the author only writes and the reader only reads. There will be exercise problems at the end of each lesson, which you as the reader and student are supposed to solve after reading the lesson. The solutions to the exercises will be provided in the next lesson for you to reflect upon. There is also a dedicated message board coupled with the school where you can ask any questions relating to the lessons.

Learning path of the school

There will be three levels on this learning curve. In the first (beginner) level, we will be discussing the .Net Framework, VB.Net Language Fundamentals and Object Oriented Programming in VB.Net. In the second (intermediate) level, we will look in more details at Object Oriented constructs in VB.Net, such as inheritance, polymorphism, abstract classes, interfaces, structures, enumerations and exceptions. In the third (advanced) level we will look at a range of areas that you will need to be knowledgeable in to solve real world problems using VB.Net with the .NET Base Libraries. Later on topics, such as Collections, Delegates, Events, Windows Programming with lot of controls, Data Access with ADO.Net, Threads and Streams.

Tools you need to enter the school:

Most of the time these examples are written in the standard IDE; Visual Studio.Net. To follow precisely to the letter you will need this. There is a free Visual Studio.Net trial version available at http://msdn.microsoft.com/vstudio/products/trial/

The trial must be ordered on a CD and therefore it's appropriate to order it right away. You also need to download the .Net Framework, which can be downloaded freely from http://msdn2.microsoft.com/en-us/netframework/aa731542.aspx

The .Net Framework also contains the VB.Net Compiler so you can also use this to compile the examples given in the lessons using a text editor such as notepad if you do not have the IDE (As mentioned above) or just cant wait to start learning.

Introduction to .Net Framework & VB.NET

Pre Microsoft .Net Days...

There were days when computer programs were written using procedural languages like C, COBOL, Pascal, etc. Code was written around functions, i.e., logic is built to control which functions to perform. Then came the Object Oriented Programming (OOP) era where languages like C++ and Smalltalk became popular. Their code was written around data, i.e., logic was built by identifying the data in the system and performing functions around this data. The advent of the Object Oriented (OO) paradigm made it possible to build, manage, improve and debug bigger applications using components. However, resources (e.g. memory) were managed by programmer themselves and there was no runtime support provided by the programming language. This caused a lot of problems by assigning programmers a lot of responsibilities, the mishandling of which could easily crash the whole application (and sometimes Operating System (OS) itself).

The first commercially successful language to provide such runtime support was Java by Sun Microsystems (although such runtime support was present in languages like Smalltalk and even in VB). Java came with a runtime environment, called the Java Virtual Machine (JVM), which performs memory management, garbage collection, checking of runtime data types and memory access. Java also presented the idea of 'Platform independence' by providing their JVM implementation for different Operating Systems and H/w so a compiled java program can run on multiple Operating Systems and h/w without any change or re-compilation (at least in theory). Java did not stop here but also made drastic changes in other popular concepts that were present in most popular languages like C++ by eliminating pointers, multiple inheritance, operator overloading and templates. All this made Java a very popular language for both academic and professional development environments, especially for web applications. But does it mean that Java kills the C++? No! Java provides this ease and simplification at the penalty of performance by introducing the language translator in the runtime. Also, because of the platform independence of Java, it lacks in some Platform specific features like GUI and event handling.

Microsoft .Net

In the year 2000, Microsoft launched its new development environment, calling it Microsoft Visual Studio .Net. Microsoft .Net, at its core, is very much similar to J2EE (Java 2 Enterprise Edition), but offered more of a compromise between the traditional 'un-managed' and the newer 'managed' style of programming. It allows a programmer to run both managed (code managed by the .Net Runtime) and un-managed code (not managed by the .Net runtime). In managed code, .Net performs memory management, run-time type checking, memory access checking, and exception handling on behalf of your program. But

does this mean that it has the same performance penalty as Java? The answer is again NO! .Net uses Just In Time (JIT) compilers to translate your intermediate compiled code to native executable code, which significantly improves performance. .Net also provides 'Platform independence' along with 'Language Independence'. The concept of 'Platform independence' is somewhat changed from Java, in that Microsoft's implementation of .NET only provided support for running your one compiled program on any h/w running any variant of the Windows OS (except for Windows 95). Open source projects like DotGNU and Mono are now also bringing .Net to other operating systems and platforms, however. .Net also provides cross language support, meaning that modules and components written in different .Net compliant languages can call/use each other's modules and components. Hence, it is possible to write your class in C#.NET, inherit it in VB.Net and finally use it in VC++.Net. At the time of this writing, as many as 22 languages are supporting the .Net Platform (including VB.NET, VB, C++, J#, Cobol, Eiffel, Pascal, FORTRAN, RPG, Smalltalk and others), enjoying full use of .net runtime and huge Framework class library (FCL).

.Net is useful for building variety of Windows applications, web application, web services, database applications and even embedded applications (using .Net compact version). On the marketing and commercial side, Microsoft is apparently putting 80% of its development resources and investment on .Net by providing a number of .Net enabled/supported applications like SQL Server.Net and Windows Server 2003. It will be a bit bold to say that Microsoft has bet its existence on the success of .Net. So, when a company of this size and mass is putting that much effort in stabling the .Net platform, it will be wise for a developer to take notice and go for it!

What is VB.Net?

VB.Net is the successor of the Visual Basic 6 programming language. VB.Net has brought about a great number of architectural changes in the Visual Basic language that are not backward compatible. A number of core elements and concepts in VB6 have been modified or removed. A great many new features have been added to the VB.Net language. It will be right to say that VB.Net will change the way Visual Basic programmers perceive their development. VB.Net has gained some credibility among serious programmers. Visual Basic has gained much favour amongst many developers. Currently, there are more VB programmers in the world than there are for any other programming language and there is more application development done in VB than in any other programming language. In the Microsoft .Net platform, VB.Net (along with C#.NET) is the language of choice. It's probable that most of the current VB6 programmers would like to switch to VB.Net when they decide to move to the .Net platform (which ultimately all Windows developers will have to; certainly the .Net platform is very much a part of LongHorn development).

VB.Net as a Programming Language

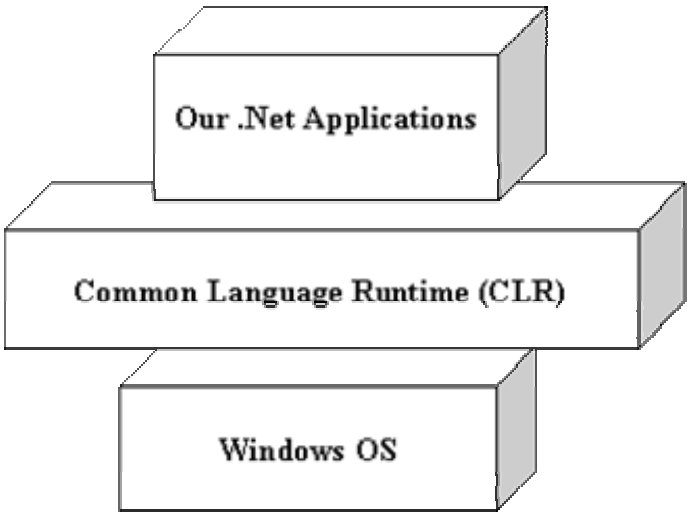
VB.Net is now a full-fledged object oriented programming language. It contains classes, objects, inheritance, polymorphism, abstract classes, interfaces, structures, enumerations, delegates and other common object oriented concepts. In VB.Net, the error handling mechanism has been modified and is now more structured. VB.Net uses the .Net standard garbage collector to release memory that is no more referenced by your apps. Probably the best thing about VB.Net is that it is a part of the .Net framework and is integrated in the .Net platform. It means that VB.Net programs can employ all the features and services exposed by the .Net framework. It can use the .Net framework class libraries, interact with program modules written in other .Net language and also use the old COM and ActiveX components. Previously, Visual Basic was mainly used for Windows applications and nothing else. Now VB.Net can be used to create console applications, Windows applications, web applications, .Net components, .Net Form controls, .Net Web controls, windows services, web services, database applications, and more. Using the .Net framework libraries you can program new exciting features like reflection, attributes, marshalling, remoting, threads, streams and also data access with ADO.Net. Compared to using VB6, VB.Net will feel extremely powerful.

.Net Architecture and the .Net Framework

In the .Net Architecture and the .Net Framework there are different important terms and concepts which we will discuss one by one.

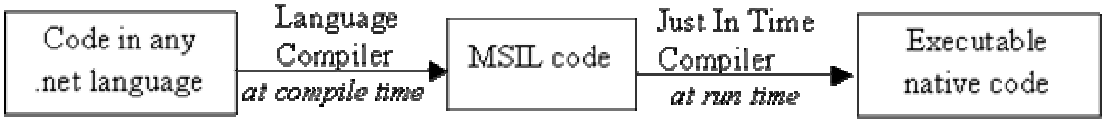
Common Language Runtime (CLR)

The most important concept of the .Net Framework is the existence and functionality of the .Net Common Language Runtime (CLR), also called .Net Runtime in short. It is a framework layer that resides above the OS and handles the execution of all the .Net applications. Our programs don't directly communicate with the OS but go through the CLR.



MSIL (Microsoft Intermediate Language) Code:

When we compile our .Net Program using any .Net compliant language (like C#, VB.Net, C++.Net) it does not get converted into the executable binary code but to an intermediate code, called MSIL or IL, understandable by CLR. MSIL is OS and hardware independent code. When the program needs to be executed, this MSIL, or intermediate code, is converted to binary executable code (native code). The presence of IL makes possible the Cross Language Relationship as all the .Net compliant languages produce similar, standard IL code.



Just In Time Compilers (JITers)

When our IL compiled code needs to be executed, CLR invokes the JIT compilers which compile the IL code to native executable code (.exe or .dll) that is designed for the specific machine and OS. JITers in many ways are different from traditional compilers as they compile the IL to native code only when desired; e.g., when a function is called, the IL of the function's body is converted to native code just in time. So, the part of code that is not used by that particular run is never converted to native code. If some IL code is converted to native code, then the next time it's needed, the CLR reuses the same (already compiled) copy without re-compiling. So, if a program runs for sometime (assuming that all or most of the functions get called), then it won't have any just-in-time performance penalty. As JITers are aware of the specific processor and OS at runtime, they can optimize the code extremely efficiently resulting in very robust applications. Also, since a JIT compiler knows the exact current state of executable code, they can also optimize the code by in-lining small function calls (like replacing body of small function when its called in a loop, saving

the function call time). Although Microsoft stated that C# and .Net are not competing with languages like C++ in efficiency and speed of execution, JITers can make your code even faster than C++ code in some cases when the program is run over an extended period of time (like web-servers).

Framework Class Library (FCL)

The .Net Framework provides a huge Framework (or Base) Class Library (FCL) for common, usual tasks. FCL contains thousands of classes to provide access to Windows API and common functions like String Manipulation, Common Data Structures, IO, Streams, Threads, Security, Network Programming, Windows Programming, Web Programming, Data Access, etc. It is simply the largest standard library ever shipped with any development environment or programming language. The best part of this library is they follow extremely efficient OO design (design patterns) making their access and use very simple and predictable. You can use the classes in FCL in your program just as you would use any other class. You can even apply inheritance and polymorphism to these classes.

Common Language Specification (CLS)

Earlier, we used the term '.Net Compliant Language' and stated that all the .Net compliant languages can make use of CLR and FCL. But what makes a language a '.Net compliant' language? The answer is the Common Language Specification (CLS). Microsoft has released a small set of specifications that each language should meet to qualify as a .Net Compliant Language. As IL is a very rich language, it is not necessary for a language to implement all the IL functionality; rather, it merely needs to meet a small subset of CLS to qualify as a .Net compliant language. This is the reason why so many languages (procedural and OO) are now running under the .Net umbrella. CLS basically addresses language design issues and lays down certain standards. For instance, there shouldn't be any global function declarations, no pointers, no multiple inheritance and things like that. The important point to note here is that if you keep your code within the CLS boundary, your code is guaranteed to be usable in any other .Net language.

Common Type System (CTS)

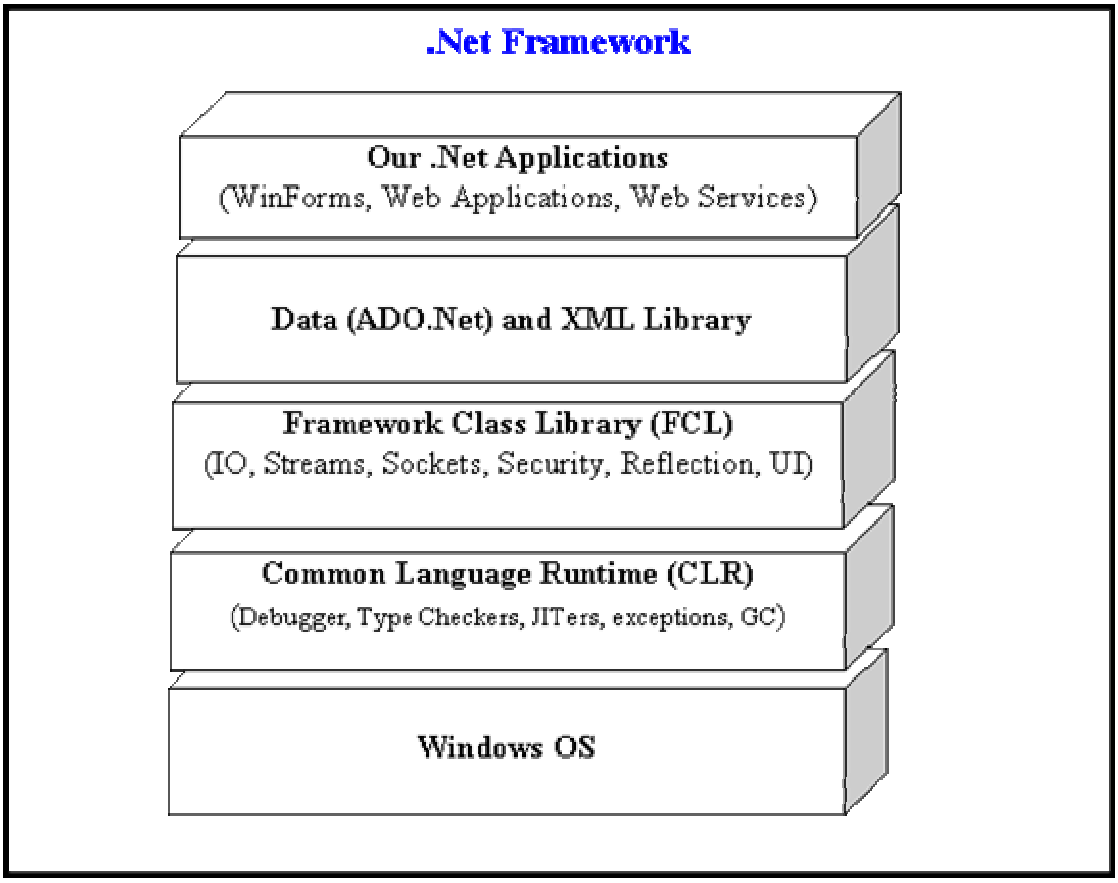
.Net also defines a Common Type System (CTS). Like CLS, CTS is also a set of standards. CTS defines the basic data types that IL understands. Each .Net compliant language should map its data types to these standard data types. This makes it possible for the 2 languages to communicate with each other by passing/receiving parameters to/from each other. For example, CTS defines a type, **Int32**, an integral data type of 32 bits (4 bytes) which is mapped by C# through **int** and VB.Net through its **Integer** data type.

Garbage Collector (GC)

CLR also contains the Garbage Collector (GC), which runs in a low-priority thread and checks for un-referenced, dynamically allocated memory space. If it finds some data that is no longer referenced by any variable/reference, it re-claims it and returns it to the OS so it can be used by other programs as needed. The presence of a standard Garbage Collector frees the programmer from keeping track of dangling data. Ask any C++ programmer how big a relief it is!

.Net Framework

The .Net Framework is the combination of layers of CLR, FCL, Data and XML Classes and our Window, Web applications and Web Services. The much publicized diagram of the .Net Framework is presented here also for better understanding.



VB.Net compared to VB6

There are quite a few differences between VB6 and VB.Net. We will highlight some of these points here:

- The greatest change in VB6 and VB.Net is that of the runtime environment. VB6 used VB-Runtime while VB.Net uses the .Net Common Language Runtime (.Net CLR). The CLR is much better designed and implemented than the VB-Runtime. The CLR uses better code translation through a Just in Time compiler. The CLR Garbage Collector is more efficient than VB6 one.
- VB6 was not a type-safe language while VB.Net is a type safe language. There is no variant type in VB.Net and no magical type conversions happen in VB.Net
- VB6 used 'On Error Goto' syntax to handle exceptions at runtime. VB.Net uses Try…Catch…Finally syntax to handle exceptions at runtime.

- A lot of code (like user interface code) in VB6 was hidden from the developer. In VB.Net no code is hidden from developer and you can access and control each part of your application.
- VB.Net has much better object oriented support than VB6
- VB6 does not allow for the development of multithreaded applications. In VB.Net you can create multithreaded applications.
- VB6 was mainly used for desktop Windows application. In VB.Net more and more people will develop web applications, distributed applications, create .net and web controls and components, write windows and web services.
- In VB.Net, you can also use reflections to read the meta-data of types and emit. You can also generate code to define and invoke types at runtime.
- VB.Net uses the .Net framework class library along with a specialized VB library (System.VisualBasic) as its core. As a result the VB.Net is much enhanced and useful compared to the features offered in the VB6 standard library
- VB.Net is platform independent because of the .Net framework. Programs written in VB.Net can run on any platform where the .Net framework is present. The platform includes both hardware and software (operating system) platforms.
- VB.Net also supports language interoperability with various .Net compliant languages. This means that you can use and enhance the code written in other .Net compliant languages. Similarly, the code written in VB.Net can also be used and enhanced by other .Net compliant languages. Although VB6 also provided this functionality through COM (Component Object Model), it was limited and difficult to use and manage. VB.Net makes it easier because of the presence of the Intermediate Language (IL) and Common Language Specification (CLS) of the .Net architecture.
- VB6 uses COM as component architecture. VB.Net uses assemblies as its component architecture. The Assemblies architecture has removed a lot of problems with COM including DLL-Hell and version control.
- Components created in VB6 (COM) need to make and update registry entries. VB.Net does not require any registry entries, making the deployment easier
- VB6 used ASP to build web applications. VB.Net uses ASP.Net to build web applications.
- VB6 used ADODB and RecordSets to implement data access applications. VB.Net uses ADO.Net and DataSets to build data access applications. ADO.Net also supports disconnected data access.

Food for thought: Exercise 1

1. All modern compilers do optimization, but in what way are JITers (Just In Time Compilers) different from traditional compilers?
2. There are a lot of advantages of using VB.Net and .Net for the developers. But why should a client or a developer jump to VB.Net and .Net? It looks like the user has to additionally install the .Net Framework in order to run the .Net application. Does it mean that the "Just in Time Compilation" outweighs the overhead of the .Net framework with no other apparent benefits?
3. The term 'Disconnected Data Source' is heavily used when talking about ADO.Net. What does it actually mean?
4. Microsoft has introduced a new language for the .Net platform called C# with almost all the features that are present in VB.Net. Why should previous VB programmers switch to

VB.Net instead of C#.Net? (Especially when VB.Net is so different from VB6 and has introduced so many new concepts)
5. Can system programming such as programming an Operating System or Compiler be done in VB.Net?

What's Next...

Next lesson, the following will be on the table

- Visual Studio.Net (VS.Net) and its common features
- Visual Studio.Net's Integrated Development Environment (IDE)
- How to make our first 'Hello World' Program
- Different elements in the 'Hello World' application such as
    - Namespaces
    - Modules
    - Class
    - Main() Sub procedure
    - Shared keyword
    - System Namespace and the Console Class
    - Writing and reading at the Console
- Compiling the Hello world application using both the command prompt and VS.Net
- Executing the Hello world application both at the command prompt & through VS.Net
- Solutions and Projects in VS.Net

Visual Studio.Net and Hello World Console Application

Visual Studio.Net & Its Common Features

Microsoft Visual Studio.Net is an Integrated Development Environment (IDE) which is a successor of Visual Studio 6. It eases the development process of .Net Applications by a great deal for VC#.Net, VB.Net, VC++.Net, JScript.Net, J#.Net, ASP.Net, etc. The revolutionary approach in this new Visual Studio.Net is that for all the Visual Studio.Net Compliant Languages there is the same IDE, debugger, project and solution explorer, class view, properties tab, tool box, standard menu and toolbars. The key features of Visual Studio.Net include:

1. Keyword and syntax highlighting
2. Intellisense (autocomplete), which helps by automatically completing the syntax as you type a dot (.) with objects, enumerations, namespaces and when you use the new keyword.
3. Project and solution management with solution explorer that helps to manage applications consisting of multiple files, which is what usually happens.
4. Help building user interface with simple drag and drop over form window.
5. Properties tab that allow you to set different properties on a number of windows and web controls.
6. Standard debugger that allows you to debug your program by putting break points for observing run-time behavior of program.
7. Hot compiler that checks the syntax of your code as you type it and reports any errors present.
8. Dynamic Help on a number of topics using the Microsoft Development Network (MSDN) library.
9. Compilating and building applications.
10. Execution of your application with/without the debugger.
11. Deploying your .Net application over the Internet or on CDs.

Project and Solutions
A Project is a combination of executable and library files that make an application or module. A project's information is usually placed in a file with the extention '.vbproj' where 'vb' represents Visual Basic. Similarly, C#.Net projects are stored as '.csproj' files. There are several different kinds of projects such as Console Applications, Windows Applications, ASP.Net Web Applications, Class Libraries and more.

A solution on the other hand is a placeholder for different logically related projects that make some application. For example, a solution may consist of an ASP.Net Web Application project and a Windows Form project. The information for a solution is stored in '.sln' files and can be managed using Visual Studio.Net's Solution Explorer. Solutions are similar to VB 6's Project Group and VC++ 6's workspace.

Toolbox, Properties and Class View Tabs

Now there is a single toolbox for all the Visual Studio.Net's languages and tools. The toolbox (usually present on the left hand side) contains a number of common controls for windows, web and data applications like the text box, check box, tree view, list box, menus, file open dialog, etc.

The Properties Tab (usually present on the right hand side in the IDE) allows you to set the properties on controls and forms without getting into code.

The Class View Tab shows all the classes that your project contains along with the methods and fields in tree hierarchy. This is similar to VC++ 6's class view.

Author's Note: It is not necessary at all that you use Visual Studio.Net to build your .net applications. You can write your code in any text editor like notepad and then compile and run it from command prompt (which we will see shortly). But Visual Studio.Net takes a lot of these responsibilities and smoothes the development process a lot, which allows you to spend more time in your business logic rather than these compiling and building hacks.

Writing Your First "Hello World" Console Application in VB.Net
As follows, we will build our first VB.Net application without and then with Visual Studio.Net. Instructions below show, how to write, compile, and execute a VB.Net application. An explanation of the different concepts in the program will follow later in the chapter.

Without Visual Studio.Net
Open "Notepad" or a text editor of your choice and type the following code:

```
Imports System
Module Module1
    Sub Main()
        Console.WriteLine("Hello World!")
    End Sub
End Module
```

Save this with any file name with the extension ".vb" (for example, 'MyFirstApplication.vb'). To compile the program, go to command prompt and type:

```
vbc MyFirstApplication.vb
```

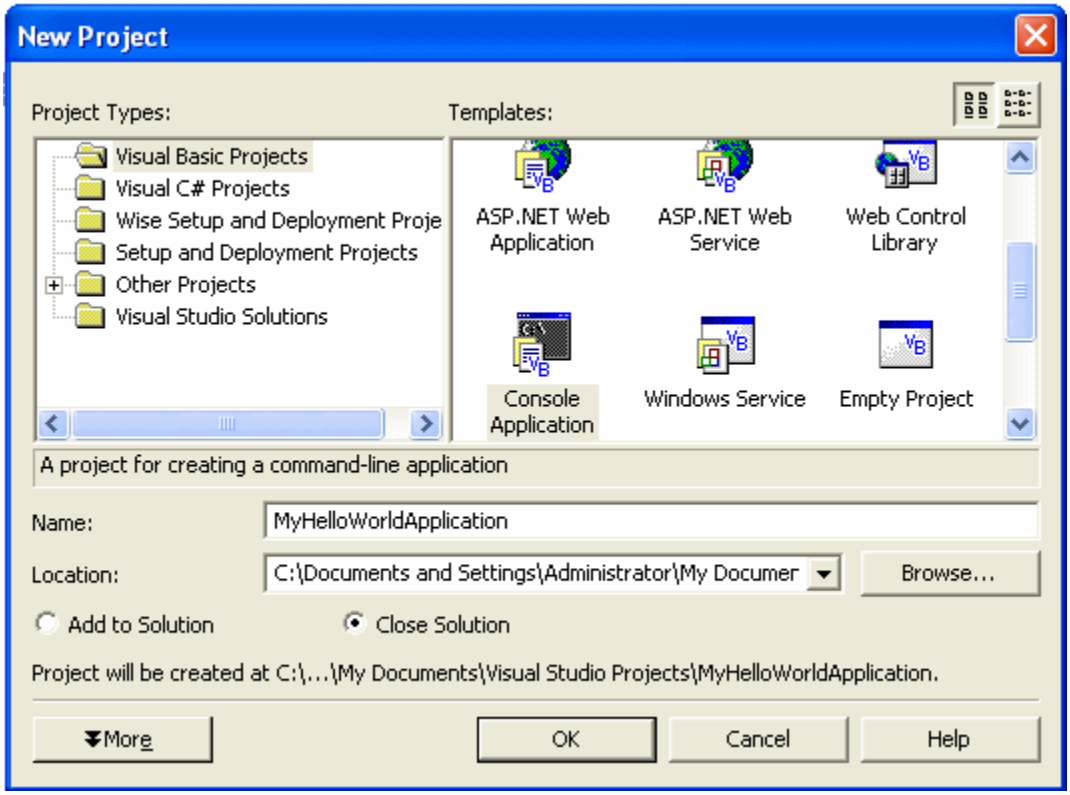
This will compile your program and create an .exe file (MyFirstApplication.exe) in the same directory. Errors will be reported if there are any. To run your program, type:

```
MyFirstApplication
```

This action will print "Hello World!" on the console screen. Simple, isn't it? Let's try it in the Visual Studio.Net IDE

With Visual Studio.Net

Start Microsoft Visual Studio.Net and from the menu select File > New > Project. A "New Project" dialog will now be displayed. Select "Visual Basic Project" from "Project Type" and select "Console Application" from "Templates". Type "**MyHelloWorldApplication**" (without "") in the "Name" text box below, then click OK.



This will show you the initial default code for your Hello World application.

```
Module Module1

    Sub Main()

    End Sub

End Module
```

Change the name of module from Module1 to "**MyHelloWorldApplication**" and type

Console.WriteLine("Hello World") inside the Sub Main() functions' body like whats shown below:

```
Module MyHelloWorldApplication
    Sub Main()
        Console.WriteLine("Hello World")
    End Sub
End Module
```

To compile and execute your application, select "Start" from the "Debug" menu or to run the application without Debug press Ctrl+F5. A new console window containing the words Hello World will now be displayed. Press any key to terminate the program and close the console window.

Let's build on this code and include some more VB.NET syntax. Modify your code as below:

```
Imports System
Namespace MyHelloWorldApplication
    Module MyHelloWorldModule
        Sub Main()
            Console.WriteLine("Hello World")
        End Sub
    End Module
End Namespace
```

Understanding the Hello World Application Code:

The first line of our program (Imports System) usually appears in all VB.Net programs. It gives us access to the core functionality of programming. This shall be covered later. Before then, the second line (Namespace MyHelloWorldApplication) shall be discussed.

Namespaces in VB.Net
A namespace is simply a logical collection of related classes in VB.Net. We bundle our related classes (like those related with database activity for example) in a named collection, hence calling it a namespace (e.g., DataActivity).

VB.Net does not allow two classes with the same name to be used in a program. The sole purpose of using namespaces is to prevent the name conflict, which may happen if your working with a large number of classes. It is the same case in the **Framework Class Library (FCL)**. It is highly possible that the Connection Class in **DataActivity** conflicts with the Connection Class of **InternetActivity**. To avoid this, these classes are made part of their respective namespace. The fully qualified name of these classes will be **DataActivity.Connection** and **InternetActivity.Connection**, hence resolving any ambiguity for the compiler.

In the second line of the code there is a declaration classes (enclosed in Namespace...EndNamespace block) which are part of the MyHelloWorldApplication namespace.

```
Namespace MyHelloWorldApplication
...
```

```
End Namespace
```

The VB.Net namespaces have NO physical mapping, as is the case in Java. Classes within the same namespace can be in different folders. The C# concept of mapping is similar to "packages" in Java and "namespace" in standard C++. The namespace may contain modules, classes, interfaces, events, exceptions, delegates and even other namespaces which are known as "Internal namespace". These internal namespaces can be defined like this:

```
        Namespace Parent
                Namespace Child
                ...
        End Namespace
End Namespace
```

The Imports Keyword

The first line of our program is

```
Imports System
```

The "Imports" keyword in the code sample above enables us to use classes in the "System" namespace. For example, Its possible to access the Console class from the Main() sub. One point to remember here is that "Imports" allows access to classes in the referenced namespace only and not in its internal/child namespaces. Hence we might need to write:

```
        Imports System.Collections
```

In order to access the classes defined in Collection namespace which is a sub/internal namespace of the System namespace.

The Module Keyword

A VB.Net program may contain one or more modules. The Main() sub-procedure usually resides in one of these modules. Modules in VB.Net are a combination of general data (fields) and general functions (methods) that are accessible to any code that can access the namespace of a module. All the members (fields, methods, properties) defined inside a module are shared by default.

The concept of a Module will be discussed in more detail in future lessons. Modules in VB.Net are defined using the **Module** statement, followed by the name of the module. The end of a module is marked with the **End Module** statement.

```
    Module MyHelloWorldModule
        ...
    End Module
```

The Main() Sub-Procedure

In the next line the Main() sub-procedure of our program is defined:

```
        Sub Main()
                ...
        End Sub
```

This is the standard layout of a Main sub-procedure within a VB.Net module. The Main() sub-procedure is the entry point of a program, i.e. a VB.Net program starts its execution from the first line in the Main sub-procedure and ceases to exist with the termination of the Main sub-procedure. We can also define the Main method inside a class, e.g.

```
Imports System

Namespace MyHelloWorldApplication
    Class MyHelloWorldClass
        Public Shared Sub Main()
            Console.WriteLine("Hello World")
        End Sub
    End Class
End Namespace
```

The main sub-procedure is designated as "Shared" as it can be called by the Common Language Runtime (CLR) without creating any objects from our MyHelloWorldClass (this is the definition of Shared methods, fields and properties). The sub-procedure is also declared as "Public" so that classes outside its namespace and assembly may call this method. Main is the (standard) name of this method. More evidence of this shall be shown later.

One interesting point is that it is legitimate to have multiple Main() methods in VB.Net program. However, you have to explicitly identify which Main method is the entry point for the program.

Printing on the Console

The next line of code prints "Hello World" on the Console screen:

```
        Console.WriteLine("Hello World")
```

In the code,WriteLine() is called. It is a "Shared" method of the Console class that is defined in the System namespace. This method takes a string (enclosed in double quotes) as its parameter and prints it on the Console window.

VB.Net, like other Object Oriented languages, uses the dot (.) operator to access the member variables (fields) and methods of a class. Also, parenthatsis () are used to identify methods in the code. String literals are enclosed in double quotation marks ("). Lastly, it must be remembered that VB.Net is a **case-insensitive language**; hence Console and conSole are the same words (identifiers) in VB.Net.

Comments

Comments are created by programmers who wish to explain the code. Comments are ignored by the compiler and are not included in the executable code. VB.Net uses similar syntax for comments as used in VB and assembly language. The text following a single quotation mark (' any comment) is a line comment. the ending is the end of the line.

```
        ' This is my main method
        Public Shared Sub Main()
            Console.WriteLine("Hello World") ' It will print Hello World
        End Sub
```

Important points to remember

- Your VB.Net executable program resides in a class or module.
- The entry point to a program is the Shared sub-procedure Main()
- VB.Net is not a case sensitive language so integer and Integer mean the same thing
- Horizontal whitespaces (tabs and spaces) are ignored by the compiler between the code. Hence, the following is also a valid declaration of the Main() method (although not recommended):

```
•           Public   Shared           Sub      Main()
•               Console.WriteLine    (   "Hello World"     )
[/ul]           End Sub
```

  - You DON'T need to save your program with the same file name as that of the class or module containing the Main() method
  - There can be multiple Main() methods in your program, but you have to specify which one is the entry point
  - The boundaries of a namespace, class, module and method are defined by their respective statements and closed with an End statement
  - A namespace is only a logical collection of classes with no physical mapping on disk (unlike Java)
  - The "Imports" keyword is used to inform the compiler where to look for the definition of the classes (namespaces) that you want to use
  - Comments are ignored by the VB.Net compiler and are used only to enhance the readability and understandability of the program for developers only.
  - Enclosing your classes or modules in a namespace is optional. Its possible to write a program where any classes or modules are not enclosed in a namespace
  - It is not mandatory that the Main method of a program does not take any argument. It may take arguments, such as:

```
o             Public Sub Main(ByVal CmdArgs() As String)
o                 Console.WriteLine("Hello World")
[/ul]         End Sub
```

A more interactive Hello World Application

Up to this point we have seen a very static Hello World application that greets the whole world when it is executed. Lets now make a more interactive hello world that greets its current user. This program will ask the user's name and will greet them using their name, like 'Hello Faraz', when the user named 'Faraz' runs it. Let's see the code first:

```
Module MyHelloWorldModule
    Sub Main()
        Console.Write("Plz, write your good name: ")  ' line 1
        Dim name As String = Console.ReadLine()               ' line 2
        Console.WriteLine("Hello {0}, Good Luck in VB.Net", name)    '
line 3
    End Sub
End Module
```

Discussing a more interactive Hello World Application
In the first line of Main, there is another method, Write(). Which is part of the Console class. This is similar to the WriteLine() method discussed in the previous program, but the cursor does not move to a new line after printing the string on the console.

In the second line, there is a declared String variable named "name". Then, a line of input is taken from the user through the ReadLine() method of the Console class. The result is stored in the "name" variable. The variables are placeholders (in memory) for storing data temporarily during the execution of the program. Variables can hold different types of data depending on their data-type, e.g., Integer variables can store integers (numbers with no decimal places), while String variables can store strings (a series) of characters. The ReadLine() method of the Console class (contrary to WriteLine()) reads a line of input typed at the Console Window. It returns this input as a string, in which the "name" variable is stored.

Author's Note: String is implicit data-type in VB.Net contrary to other languages

The third line prints the name given by the user at second line along with a greeting text. Once again, the WriteLine() method of the Console Class is used. The substitution **parameter {0}** is used to specify the position in the line of text where the data from the variable "name" should be written after the WriteLine() method is called.

```
Console.WriteLine("Hello {0}, Good Luck in VB.Net", name);
```

When the compiler finds a substitution **parameter {n}** it replaces it with the (n+1) variable following the string in double quotation marks separated by comma. Hence, when the compiler finds {0}, it replaces it with (0+1), i.e., 1st variable "name" following the double quotes separated by comma. At run-time, the CLR will read it as:

```
Console.WriteLine("Hello Faraz, Good Luck in VB.Net");
```

If the value of the variable "name" = "Faraz" at run-time.

Alternatively, it can also be written as:

```
Console.WriteLine("Hello " + name + ", Good Luck in VB.Net");
```

Removing the substitution parameter altogether. Here we concatenate (add) the strings together to form a message. (The first approach is similar to C's printf() function while the second is similar to Java's System.out.println() method)

When we compile and run this program the output will be as follows:

"Plz, write your good name: Faraz Hello Faraz, Good Luck in VB.Net"

Food for thought: Exercise 2

1. I wrote, compiled and executed these programs just like the programs of other languages. Where are all those .Net framework items I keep hearing about: CLR, JITers, MSIL, etc...?
2. How is it possible to call the methods of the Console Class when I did not create an object from it? (I know some Object Oriented Concepts from other languages)
3. When there is more than one class in a program, how would the compiler recognize which class has the Main() method? If more than one class has a valid Main method, what does the compiler do then?
4. Is it possible to have more than one Main method in a program? If yes, then how would compiler decide which Main method to call when executing the program?
5. Write a program that asks for the name and phone number of the user at runtime. Print that data 3 times on the same line and then 3 times on seperate lines.
6. How can you do compilation and execution separately using Visual Studio.Net (VS.Net)?

Solution of Last Issue's Exercise (Exercise 1)

1. All modern compilers do optimization, but in what way are JITers (Just In Time Compilers) different from traditional compilers?
The basic difference between the two is that while traditional compilers do optimizations at compile time, JITers optimize code at run-time where things are more clear and less ambiguous. Also, as JITers work at run-time, they know the exact state of your program as well as the Operating System and Microprocessor. So, they can produce more suitable assembly language instructions, call more appropriate API's and optimize more effectively than traditional compilers do!

2. There are a lot of advantages of using VB.Net and .Net for the developers. But why should a client or a developer jump to VB.Net and .Net? It looks like the user has to additionally install the .Net Framework in order to run the .Net application. Does it mean that the "Just in Time Compilation" outweighs the overhead of the .Net framework with no other apparent benefits?
For clients, the biggest advantages are faster development of more stable, manageable, scalable, faster (in cases like ASP.Net applications which are faster than previous ASP applications) and secure applications at (maybe) lower production cost. As the .Net Framework is installed with your program, if needed, and as many programs are now

deployed through CDs, your client might not even notice the 20-30 MB Framework. For the speed of execution, as I mentioned earlier, because of JIT compilation, the speed of .Net programs tend to improve as the code executes repetitively in a single run, so after some time the speed factor would also not be noticeable for user.

Also to get away from JIT compilation, you can use Native Image Generator (NGen.exe) utility provided with .Net. The Native Image Generator creates a native image from a managed assembly and installs it into the native image cache on the local computer. Running Ngen.exe on an assembly allows the assembly to load and execute more quickly, because it restores code and data structures from the native image cache rather than generating them dynamically. The syntax for generating a native image for userAccount.exe with the specified path is

ngen c:\userAccount.exe

See the MSDN Library to learn more about NGen.exe.

3. The term 'Disconnected Data Source' is heavily used when talking about ADO.Net. What does it actually mean?
ADO.Net (Active Data Objects.Net) uses a different approach to connect to the Database server than traditional data access components. It makes the connection to a database server only when it needs to do a transaction with the server, and gets disconnected once the transaction is over (like a HTTP connection over internet). This greatly reduces the overhead involved in staying connected with the server even when no transaction is being performed. It also lessens the burden on the server, making it more productive.

4. Microsoft has introduced a new language for the .Net platform called C# with almost all the features that are present in VB.Net. Why should previous VB programmers switch to VB.Net instead of C#.Net? (Especially when VB.Net is so different from VB6 and has introduced so many new concepts)
Yes, the two languages are quite similar, but they do have minor differences. Most of the VB6 developers are likely to shift to VB.Net while most of the programmers from Java and C++ are likely to shift to C# when writing the managed code for .Net platform. VB.Net (and the whole .Net platform) has inherited a lot from VB6. VB programmers will feel very much at home when designing windows application.

5. Can system programming such as programming an Operating System or Compiler be done in VB.Net?
For compilers, I would say YES! The job of compilers is to check syntax and generate the assembly code, which is later converted to machine language code by assemblers in the traditional compiler case. For the .Net framework, compilers generate the MSIL code. So both of these types of compilers are perfectly possible in a managed environment. For Operating systems…it might be possible in unmanaged code but the intent of the Vb.Net language is not to do system programming but application programming, as stated in the VB.Net language specification.

What's Next...

Next time, we will be discussing VB.Net Language Fundamentals including

- Basic data types and their mapping to CTS (Common Type System)
- Declaring & using variables
- Operators (Mathematical, incremental/decremental, logical, relational)
- Flow Control using if….else and switch…case
- Function declaration and calling
- Loops (for, do…while, repeat…until)
- Arrays (one dimensional)

VB.Net Language Fundamentals

Lesson Plan

This lesson is about learning the language fundamentals of VB.Net. We will explore the data types in VB.Net, using variables, different kinds of operators in VB.Net, flow control statements like If...Then...Else, looping structures and how to use arrays.

Basic Data Types and their mapping to the CTS (Common Type System)

There are two kinds of data types in VB.Net

1. Value type (implicit data types, Structure and Enumeration)
2. Reference Type (objects, delegates)

Value types are passed to methods by passing an exact copy while Reference types are passed to methods by passing only their reference (handle). Implicit data types are defined in the language core by the language vendor, while explicit data types are types that are made by using or composing implicit data types.

As we saw in the first lesson, implicit data types in .net compliant languages are mapped to types in Common Type System (CTS) and CLS (Common Language Specification). Hence, each implicit data type in VB.Net has its corresponding .Net type. The implicit data types in VB.Net are:

VB.Net type	Corresponding .Net type	Size in bytes	Description
Boolean	Boolean	1	Contains either True or False
Char	Char	2	Contains any single Unicode character enclosed in double quotation marks followed by a c, for example "x"c
Integral types			
Byte	Byte	1	May contain integers from 0-255
Short	Int16	2	Ranges from -32,768 to 32,767
Integer(default)	Int32	4	Ranges from -2,147,483,648 to 2,147,483,647
Long	Int64	8	Ranges from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
Floating point types			
Single	Single	4	Ranges from ±1.5 × 10-45 to ±3.4 × 1038 with 7 digits precision. Requires the suffix 'f' or 'F'
Double(default)	Double	8	Ranges from ±5.0 × 10-324 to ±1.7 × 10308 with 15-16 digits precision.
Decimal	Decimal	12	Ranges from 1.0 × 10-28 to 7.9 × 1028 with 28-29 digits precision. Requires the suffix 'm' or 'M'

Implicit data types are represented in language using 'keywords'; so each of above is a keyword in VB.Net (Keyword are the words defined by the language and can not be used as identifiers). It is worth-noting that string is also an implicit data type in VB.Net, so String is a keyword in VB.Net. Last point about implicit data types is that they are value types and

thus stored at the stack, while user defined types or referenced types are stored at heap. Stack is a data structure that store items in last in first out (LIFO) fashion. It is an area of memory supported by the processor and its size is determined at the compile time. Heap is the total memory available at run time. Reference types are allocated at heap dynamically (during the execution of program). Garbage collector searches for non-referenced data in heap during the execution of program and returns that space to Operating System.

Variables
During the execution of program, data is temporarily stored in memory. A variable is the name given to a memory location holding particular type of data. So, each variable has associated with it a data type and value. In VB.Net, a variables is declared as:

```
Dim <variable> as <data type>
```

e.g.,

```
Dim i As Integer
```

The above line will reserve an area of 4 bytes in memory to store integer type values, which will be referred in the rest of program by identifier 'i'. You can initialize the variable as you declare it (on the fly) and can also declare/initialize multiple variables of same type in a single statement, e.g.,

```
Dim isReady As Boolean = True
Dim percentage = 87.88, average = 43.9 As Single
Dim digit As Char = "7"c
```

VB.Net Option Strict and Option Explicit Settings
There are two 'bad' features in VB.Net, which are inherent from earlier versions (VB5 and VB6):

- You can declare a variable without specifying its type. VB.Net, in this case, assumes the type of the variable as System.Object class
- You can convert values (or objects) to incompatible types, e.g., String to Integer.

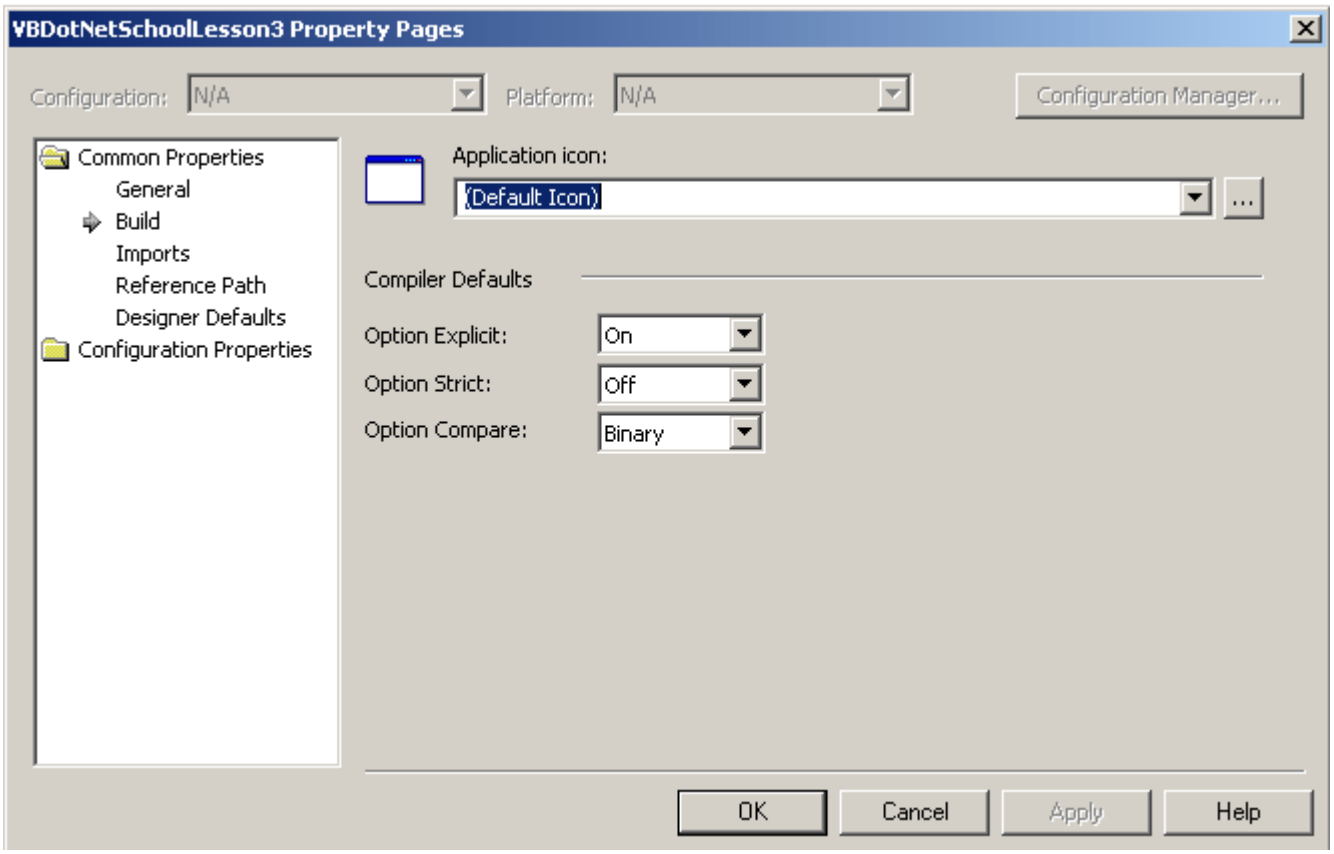
Why I called the two options bad? The use of these two features results in quite a number of bugs and makes the overall design of application bad, complex and difficult to follow. With incompatible type conversion, the program does compile without any error but throw a runtime error (exception). But these two features can be turned off by using the Option Explicit and Option Strict statements.

Option Explicit Statement
Option Explicit, when turned on, do not allow to use any variable without proper declaration.

There are two methods to apply the Option Explicit Statement.

- To apply the Option Explicit settings to the complete project in Visual Studio.Net, right click the project name in the solution explorer and select Properties. It will

open the Property Pages window. Now in the Common Properties tree at left, select Build, it will show the following window



From here, you can turn the Option Explicit (as well as Option Strict) on or off.

- To apply the Option Explicit settings to the current file, use the Option Explicit statement before any statement as,

```
Option Explicit On
```

When Option Explicit is on, it will cause the compile time error to write

```
    myName = "Faraz"     ' compile time error with Option Explicit On
```

Rather, you would have to write,

```
    Dim myName As String = "Faraz"
```

Option Strict Statement
When the Option Strict statement is turned on, incompatible type conversion are not allowed. Option Strict can be turned on or off in the similar fashion as Option Explicit. You can either use Option Strict Statement as

```
Option Strict On
```

Or you can set it from the project properties. When Option Strict is On, the following program will cause a compile time error

```
Sub Main()
Dim strNum As String = "1"
Dim intNum As Integer = strNum
Console.WriteLine(intNum)
End Sub
```

But if the Option Strict is turned off, the above program will actually compile and run without error to print 1 on the Console!

It is important to remember that Option Strict also does not allow using un-declared types and hence there is no use turning the Option Explicit on if you are already using Option Strict.

Finally, we do not discourage our readers to turn Option Explicit and Option Strict off; It's strongly advised not to do so! Throughout the VB.Net School, we will implicitly assume that the Option Strict is turned On

Constant or Symbols
Constants values once defined cannot be changed in the program. Constants are declared using Const keyword, like:

```
Dim Const PI As Double = 3.142
```

Constants must be initialized as they are declared.

```
Dim Const MARKS As Integer
```

It is a notation convention to use capital letters while naming constants.

Naming Conventions for variables and methods
Microsoft suggests using **Camel Notation** (first letter in lowercase) for variables and **Pascal Notation** (first letter in uppercase) for methods. Each word after the first word in the name of both variable and method should start with capital letter. For example, variable names following Camel notation could be

```
salary                  totalSalary
myMathsMarks            isPaid
```

Some typical names of method following Pascal Notation are

```
GetTotal()                          Start()
WriteLine()                         LastIndexOf()
```

Although it is not mandatory to follow this convention, it is highly recommended to follow it. Microsoft no longer uses Hungarian notation. An example wooul be "iMarks" for an integer variable. Also, using underscores _ in names is also not encouraged.

Breaking lines in VB.Net
The VB.Net compiler identifies the end of statement by the end of line. Hence, it is not possible to write a single statement on multiple lines (as done in C/C++, Java, C#). The following code will raise a syntax error:

```
Dim myName As String = "My name is
Faraz Rasheed"
```

The compiler treats the two lines as two instructions and will cause syntax errors upon finding these two lines. To expand a single statement on to multiple lines, you must use the underscore _ character at line breaks. For example, the above code is perfectly valid when modified as below

```
Dim myName As String =    "My name is " & _
                                        "Faraz Rasheed and, " & _
                                        "I like .Net as ..."
Console.WriteLine(myName)
```

The above code fragment will result in following output at Console

My name is Faraz Rasheed and, I like .Net as...

Operators in VB.Net

Arithmetic Operators
Several common arithmetic operators are allowed in VB.Net like

```
+       (add)
-       (subtract)
*       (multiply)
/       (divide)
Mod     (remainder or modulo)
```

The program below uses these operators

```
Imports System
Module ArithmeticOperators
    ' The program shows the use of arithmetic operators
```

```
    ' + - * / Mod
    Sub Main()
        ' result of addition, subtraction, multiplication and modulus operator
        Dim sum, difference, product, modulo As Integer
        sum = 0
        difference = 0
        product = 0
        modulo = 0
        Dim quotient As Double = 0  ' result of division
        Dim num1 As Integer = 10    ' operand variables
        Dim num2 As Integer = 2
        sum = num1 + num2
        difference = num1 - num2
        product = num1 * num2
        quotient = num1 / num2
        modulo = 3 Mod num2         ' remainder of 3/2
        Console.WriteLine("num1 = {0}, num2 = {1}", num1, num2)
        Console.WriteLine()
        Console.WriteLine("Sum        of {0} and {1} is {2}", num1, num2, sum)
        Console.WriteLine("Difference of {0} and {1} is {2}", num1, num2, difference)
        Console.WriteLine("Product    of {0} and {1} is {2}", num1, num2, product)
        Console.WriteLine("Quotient   of {0} and {1} is {2}", num1, num2, quotient)
        Console.WriteLine()
        Console.WriteLine("Remainder when 3 is divided by {0} is {1}", num2, modulo)
    End Sub
End Module
```

Although the program above is quite simple, Lets discuss some concepts. In the Console.WriteLine() method, we have used format-specifiers {int} to indicate the position of variables in the string.

```
        Console.WriteLine("Sum of {0} and {1} is {2}", num1, num2, sum)
```

Here, {0}, {1} and {2} will be replaced by the values of num1, num2 and sum variables. In {i}, i specifies that the (i+1)th variable after the double quotes will replace it when printed on the Console. Hence, {0} will be replaced by first, {1} will be replaced by second variable and so on.

Assignment Operators
Assignment operators are used to assign values to variables. Common assignment operators in VB.Net are:

```
=       (simple assignment)
+=      (additive assignment)                   -=      (subtractive assignment)
```

- `=  (multiplicative assignment)  /=  (division assignment)[/pre]`
-

```
The Equal operator is used to assign a value to a variable or a reference.
For example, the instruction
```

```
        Dim isPaid As Boolean = false
```

assigns the value 'False' to the isPaid variable of Boolean type. The Left and right hand side of the equal or any other assignment operator must be compatible otherwise the compiler will complain of a syntax error.

Sometimes casting is used for type conversion, e.g., to convert and store values in a variable of type Double to a variable of type Integer. We need to apply integer cast using VB.Net's CType() built-in method

```
Dim doubleValue As Double = 4.67
Dim intValue As Integer    = CType(doubleValue, Integer)    ' intValue would be equal to 4
```

The method CType() is used for compatible type conversions. It takes two arguments; the first being the source variable to convert to, while the second argument is the target type of conversion. Hence, the above call to the method CType() will convert the value in the variable 'doubleValue' of type Double to a variable of type Integer and will return the converted Integer type value that will be stored in the Integer variable 'intValue'

Of course, with narrow casting (from bigger type to smaller type) there is always a danger of some loss of precision; as in the case above, we only got 4 of the original 4.67. Sometimes, the casting may result in a runtime error.

```
Dim intValue As Integer   = 32800
Dim shortValue As Short   = CType(intValue, Short)
```

When the second of these lines is run an error will be given, stating that "Arithmetic operation resulted in an overflow." Why is it so? Variables of type Short can only take a maximum value of 32767. The cast above can not assign 32800 to a shortValue. This is detected at runtime and an error is given.

If you try to do an invalid cast of incompatible types like below

```
Dim strValue As String = "Faraz"
Dim intValue As Integer = CType(strValue, Integer)
```

Then again it will get compiled but will crash the program at runtime.

Author's Note: You might be wondering where did the CType() method come from? Which namespace or class does it belongs to? All the VB.Net applications by default import the System.VisualBasic namespace. This namespace contains a lot of useful utility methods for common usage. CType() is also defined in this namespace. You can check the exact location for any type or member definition, you can right-click it in Visual Studio.Net and select 'Go to definition'. When you select this option with CType() method, it will open an Object Browser window showing the hierarchy of the selected member.

Relational Operators
Relational operators are used for comparison purposes in conditional statements. The common relational operators in VB.Net are:

```
=       (equality check)                <>      (un-equality check)
>       (greater than)                  <       (less than)
>=      (greater than or equal to)      <=      (less than or equal to)
```

Relational operators always result in a Boolean statement; either True or False. For example if we have two variables

```
        Dim num1 = 5, num2 = 6 As Integer
```

then,

```
        num1 =  num2    will result in false
        num1 <> num2    will result in true
        num1 >  num2    will result in false
        num1 <  num2    will result in true
        num1 <= num2    will result in true
        num1 >= num2    will result in false
```

Only compatible data types can be compared. It is invalid to compare a Boolean with an Integer, if

```
        Dim i = 1 As Integer
        Dim b = True As Boolean
```

then it is a syntax error to compare i and b for equality (i=b)

Logical and Bitwise Operators
These operators are used for logical and bitwise calculations. The common logical and bitwise operators in VB.NET are:

```
And     (bitwise AND)                           Or      (bitwise OR)
Xor     (bitwise XOR)                           Not     (bitwise NOT)
AndAlso (Logical or short circuit AND) OrElse  (Logical or short circuit OR)
```

The operators And, Or and Xor are rarely used in usual programming practice. The Not operator is used to negate a Boolean or bitwise expression like:

```
Dim b = False As Boolean
Dim bb As Boolean = Not b ' bb would be true
```

Logical Operators And, Or, AndAlso and OrElse are also used to combine comparisons like

```
Dim i=6, j=12 As Integer
Dim firstVar  As Boolean = i>3 And j < 10 ' firstVar would be false
Dim secondVar As Boolean = i>3 Or j < 10 ' secondVar would be true
```

In the first comparison case: i>3 And j<10 will result in true only if **both** the conditions i>3 and j<10 result in true. While in the second comparison: i>3 Or j<10 will result in true if **any** of the conditions i>3 and j<10 result in true. You can of course use the combination of And, Or, AndAlso and OrElse in a single statement like:

```
bool firstVar    = (i>3 And j<10) OrElse (i<7 And j>10)
                    'firstVar would be true
```

In the above statement we used brackets to group our conditional expressions to avoid any ambiguity.

You can also use And and Or operators in place of AndAlso and OrElse respectively; but for combining conditional expressions, AndAlso and OrElse are more efficient as they use "short circuit evaluation", i.e., if in (i>3 AndAlso j<10) expression, i>3 evaluates to false, it would not check the second expression j<10 and will return false (as in AND, if one of the participant operand is false, the whole operation will result in false). Hence, one should be very careful to use assignment expressions with AndAlso and OrElse operators. The And and Or operators don't do short circuit evaluation and do execute all the comparisons before returning the result.

Other Operators
There are other operators present in VB.Net. A short description of these is given below:

```
(member access for objects)    ()      (indexing operator used in arrays and
collections)
```

Operator Precedence
All operators are not treated equally. There is a concept of operator precedence in VB.Net as in

```
Dim i As Integer = 2 + 3 * 6                ' i would be 20 not 30
```

3 will be multiplied by 6 first then the result will be added to 2. This is because the multiplication operator * has precedence over the addition operator +. For a complete table of operator precedence, consult msdn or the .net framework documentation.

Flow Control And Conditional Statements

If...Then...Else statement
Condition checking has always been the most basic and important construct in any language. VB.Net provides conditional statements in the form of the If...Then...Else statement. The structure of this statement is:

```
If Boolean expression Then
```

```
        Statement or block of statement
Else
        Statement or block of statement
End If
```

The Else clause above is optional. The typical example is

```
If i=5 Then
        Console.WriteLine("Thanks God, i finally becomes 5")
End If
```

In the above example, the console message will be printed only if the expression i=5 evaluates to True. If action is needed in the case when the condition does not evaluate to true you can use the Else clause.

```
If i=5 Then
       Console.WriteLine("Thanks God, I finally becomes 5")
Else
        Console.WriteLine("Missed...When will I become 5?")
End If
```

Only the first message will be printed in the case of i being 5. In any other case (when i is not 5), the second message will be printed. You can also use a block of statements (more than one statement) under any If and Else.

```
If i=5 Then
j = i*2
Console.WriteLine("Thanks God, i finally becomes 5")
Else
j = i/2
Console.WriteLine("Missed...When will i become 5?")
End If
```

You may write If...Then or If...Then...Else in the single line, like

```
If i=5 Then Console.WriteLine("Thanks God, I finally became 5")
```

Or,

```
If i=5 Then j = i*2 Else j = i/2
```

As you might have picked from the above two statements. When an If...Then and If...Then...Else are used on the same line, we do not need to write an End If. The reason is quite simple; End is used to mark the end of a block in VB.Net. With these two statements, we do not create or use any blocks of statements

I would always recommend to use If...Then and If...Then...Else statements in block format with End If. It increases the readability and prevents many bugs that otherwise can be produced in the code.

You can also have an If after an Else for further conditioning

```
If i=5 Then       'line 1
        Console.WriteLine("Thanks God, i finally becomes 5")
ElseIf i=6        'line 3
        Console.WriteLine("Ok, 6 is also closer to 5")
Else              'line 5
        Console.WriteLine("Missed...When will i become 5 or closed to 5?")
End If
```

ElseIf i=6 is executed only if the first condition i=5 is false. An Else at line 5 will be executed only if the second condition i=6 (line 3) executes and fails (that is both the first and second condition fails). The point being is that Else at line 5 is related to the If on line 3

As If...Then...Else is also an statement, you can use it under other If...Then...Else statements, like:

```
If i>5 Then       ' line 1
If i=6 Then       ' line 2
        Console.WriteLine("Ok, 6 is also closer to 5")
Else              ' line 4
        Console.WriteLine("Oops! i is greater than 5 but not 6")
End If
        Console.WriteLine("Thanks God, i finally becomes greater than 5")
Else              ' line 8
        Console.WriteLine("Missed...When will i become 5 or closed to 5?")
End If
```

The Else on line 4 is clearly related to the If...Then on line 2 while the Else on line 8 belongs to the If on line 1. Finally, do note (VB6 and C/C++ programmers especially) that the If statement expects only Boolean expression and not an Integer value. It is indeed an error to write

```
Dim flag As Integer = 0
If flag Then
        ' do something...
End If
```

Instead, you can either use

```
Dim flag As Integer = 0
If flag = 1       Then              ' note ==
        ' do something…
End If
```

or,

```
Dim flag As Boolean = False
If flag Then      ' Boolean expression
                  ' do something...
End If
```

The key to avoid confusion when using a complex combination of If...Then...Else is the

- Habit of using blocked If...Then...Else...End If at all times
- Indentation: aligning the code to enhance readability. If you are using Visual Studio.Net or some other editor that supports coding, the editor will do the indentation for you. Otherwise, you have to take care of this yourself.

I strongly recommend to follow the above two guideline or not to use the If...Then...Else statement at all :)

Select...Case statement

If you need to perform a series of specific checks, Select...Case is present in VB.Net and is just the ticket for this. The general structure of Select...Case statement is as follows:

```
Select <any implicit data type expression>
       Case expression
         statements
         ' some other case blocks
         ...
         Case Else
         statements
End Select
```

It takes much less time to use the Select...Case than using several If...Then...ElseIf statements. Let's understand it with an example

```
Imports System
' To execute the program write "SwitchCaseExample 2" or
' any other number at command line,
' if the name of .exe file is "SwitchCaseExample.exe"
Module ArithmeticOperators
    ' Demonstrates the use of switch...case statement along with
    ' the use of command line argument
    Sub Main(ByVal userInput() As String)
    ' convert the string input to integer
    ' Will through run-time exception if there is no input
    ' at run-time or input is not castable to integer
      Dim input As Integer = Integer.Parse(userInput(0))
      Select Case input ' what is input?
           Case 1  ' if it is 1
           Console.WriteLine("You typed 1 (one) as first command line argument")
           Case 2 ' if it is 2
               Console.WriteLine("You typed 2 (two) as first command line argument")
           Case 3 To 5  ' if it is 3
               Console.WriteLine("You typed number from 3 (three) to
                                  five (five) as first command line argument")
           Case Else  ' if it is not of the above
               Console.WriteLine("You typed other than 1, 2, 3, 4 and 5")
        End Select
    End Sub
End Module
```

An integer must be supplied for the command line argument. Firstly, compile the program (at command line or in Visual Studio.Net). Suppose we made an exe with name "SelectCaseExample.exe". Now run it at the command line like so:

C:\>VBDotNetSchoolLesson3 2
You typed 2 (two) as command line argument
or,
C:> VBDotNetSchoolLesson3 4
You typed number from 3 (three) to five (five) as first command line argument

or,
C:> VBDotNetSchoolLesson3 7
You typed other than 1, 2, 3, 4 and 5

If you did not enter any command line argument or give a non-integer argument, the program will raise an exception

Unhandled Exception: System.IndexOutOfRangeException: Index was outside the bounds of the array.
at VBDotNetSchoolLesson3.ArithmeticOperators.Main(String[] userInput) in C:\D ocuments and Settings\farazr.YEVOLVE\My Documents\Visual Studio Projects\VBDotNe tSchoolLesson3\Module1.vb:line 14

Lets come to the internal workings. We converted the first command line argument (userInput(0)) into an Integer variable input. For conversion, we used static Parse() method of the Integer data type. This method takes a String and returns an equivalent integer or raises an exception if the conversion can't be completed. Next we checked the value of input variable using switch the statement

```
Select Case input
...
End Select
```

Later on in the basis of the input values, we took specific actions under the respective case statements. Case 1 ' if it is 1 Console.WriteLine("You typed 1 (one) as the first command line argument")

We can also specify a range in the Case Expression to match

```
Case 3 To 5   ' if it is 3
     Console.WriteLine("You typed number from 3 (three)
                        to five (five) as first command line argument")
```

If all the specific checks fail (input is neither 1,2,3,4 or 5), the statements under "Case Else" will execute.

```
Case Else   ' if it is not of the above
     Console.WriteLine("You typed other than 1, 2, 3, 4 and 5")
```

There are some important points to remember when using switch…case statement in VB.Net

- You can use any implicit data type in the Select statement
- You can use multiple statements under a single case statement as follows

```
Case "Pakistan"
     continent = "Asia"
     Console.WriteLine("Pakistan is an Asian Country")
     Case Else
     continent = "Un-recognized"
     Console.WriteLine("Un-recognized country discovered")
```

- Statements under Case Else will only be executed if and only if all Case checks fail.
- You can't have more than one Case Else block in a single Select statement.

Loops In VB.Net

Loops are used for iteration purposes, i.e., performing a task multiple times (usually until a termination condition is met)

For...Next Loop

The most common type of loop in VB.Net is the For...Next loop. The basic structure of the For...Next loop is exactly the same as in VB6 and is like so:

```
For variable = startingValue To lastValue
        statement or block of statements
Next
```

Lets see a For...Next loop that will write integers from 1 to 10 on the console

```
Dim i As Integer
For i = 1 to 10
    Console.WriteLine("In the loop, value of i is " & i)
Next
```

At the start, an integer variable i is initialized with the value of 1, then the statements under the For are executed until the value of i does not equal 10. Each time i is incremented by 1.

The important points about for loop are:

- You can use an Exit For statement in a For...Next loop or any other loop to change the normal execution flow
- An Exit For statement terminates the loop and transfers the execution point outside the for loop as below:

```
For i=1 to 10
    If i>5 Then
    Exit For
End If
    Console.WriteLine("In the loop, value of i is {0}.", i)
Next
```

The loop will terminate once the value of i gets greater than 5. If some statements are present after Exit For, it should be enclosed under some conditions. Otherwise the lines following the break point will not execute

```
For i = 1 To 10
    Exit For
    Console.WriteLine()
Next
```

- You can define the increment/decrement (change) in each iteration of a For...Next Loop using the Step statement. The code below will increment by 2 in the value of i in each cycle of the loop

```
For i = 1 To 10 Step 2
    Console.WriteLine("Value of i is {0}", i)
Next
```

The following is the output on the Console,

Value of i is 1
Value of i is 3
Value of i is 5
Value of i is 7
Value of i is 9
Press any key to continue Note that the increment starts after the first iteration. You can also specify the negative increment (i.e., decrement) in a Step. The following code will output Integers from 10 to 1

Value of i is 10
Value of i is 9
Value of i is 8
Value of i is 7
Value of i is 6
Value of i is 5
Value of i is 4
Value of i is 3
Value of i is 2
Value of i is 1
Press any key to continue

Do While...Loop
The general structure of the Do While...Loop is

```
Do While Boolean expression
     Statement or block of statements
Loop
```

The statements under Do While will run continuously as long as the Boolean expression evaluates to true. The similar code for printing integers 1 to 10 on Console using the Do While...Loop is

```
Dim i As Integer =1
Do While i<=10
     Console.WriteLine("In the loop, value of i is " & i)
i = i + 1
Loop
```

Do...Loop While
A Do...Loop While is similar to a Do While...Loop, except that it does not check the condition before entering the first iteration (execution of code inside the body of loop). The general form of the a Do...Loop While is:

```
Do
statement or block of statements
Loop While Boolean expression
```

The statements under the Do will be executed first and then the Boolean condition is checked. The loop will continue until the condition remains true. The code which prints integers 1 to 10 on console using Do...Loop While is

```
Dim i As Integer = 1
Do
Console.WriteLine("In the loop, value of i is " & i)
i = i + 1
Loop While i<=10
```

The important point is that the statements in a Do...Loop While execute at least once.

Do...Loop Until
A Do...Loop Until is similar to the Do...Loop While, except that it continues to execute the containing statements until the condition against the Until part evaluates to True or the condition against the Until remains False. The general form of the Do...Loop Until is as follows:

```
Do
         statement or block of statements
Loop Until Boolean expression
```

The statements under the Do will execute first and then the condition is checked. The loop will continue until the condition remains false. The following code will print integers from 1 to 10 on console using the Do...Loop Until.

```
Dim i As Integer = 1
Do
     Console.WriteLine("In the loop, value of i is " & i)
i = i + 1
Loop Until i=10
```

Again the statements in Do...Loop Until execute at least once.

Arrays in VB.Net

Declaration

An Array is a collection of values of similar data type. Technically, VB.Net arrays are of reference type. Each array in VB.Net is an object and is inherited from the System.Array class. Arrays are declared as follows:

```
Dim <identifier>(<size of array>) As <data type>
```

Lets define an array of Integer type to hold 10 integers.

```
Dim myIntegers(9) As Integer
```

The above will create an array of 10 integers from the index of 0 to 9. The size of an array is fixed and must be defined before use. You can also use variables to define the size of array like so:

```
Dim size As Integer = 10
Dim myIntegers(10-1) As Integer
```

You can optionally perform declaration and initialization in separate steps like below:

```
Dim myIntegers() As Integer
myIntegers = New Integer() {1, 2, 3, 4, 5}
```

Here we initialized the array myIntegers using the values it holds. Note, we must enclose the values in curly brackets and separate the individual values with commas. This will create an array size of 5, whose successive values will be 1, 2, 3, 4, 5

It is important to note that when an array declaration and initialization are performed separately, you must provide values for each element.

Accessing the values stored in array

To access the values in an Array, we use the indexing operator (Integer index) by passing an Integer to indicate which particular index value we wish to access. It's important to note

that index values in VB.Net starts from 0. So, if an array contains 5 elements, first element would be at index 0, second at index 1 and last (fifth) at index 4. The following code demonstrates how to access the 3rd element of an array

```
Dim myIntArray() As Integer
myIntArray = New Integer() {5, 10, 15, 20}
Dim j As Integer = myIntArray(2)
```

Lets make a program that uses an integral array.

```
' demonstrates the use of arrays in VB.Net
Public Sub Main()
' declaring and initializing an array of type integer
Dim myIntegers() As Integer = New Integer() {3, 7, 2, 14, 65}
' iterating through the array and printing each element
Dim i As Integer
For i = 0 to 4
Console.WriteLine(myIntegers(i))
Next
End Sub
```

Here we used the For...Next loop to iterate through an array and use the Console.WriteLine() method to print each individual element of the array. Note how the indexing operator () is used.

The above program is quite simple and efficient, but we have to hard-code the size of the array in the For...Next loop. As we mentioned earlier, arrays in VB.Net are reference type and are a sub-class of the System.Array Class. This class has a lot of useful properties and methods that can be applied to any instance of an array. Properties are very much like a combination of getter and setter methods in common Object Oriented languages. Properties are context sensitive; meaning the compiler can un-ambiguously identify whether it should call a getter or a setter in certain contexts. We will discuss properties in detail in the coming chapters. System.Array has a very useful read only property "Length" that can be used to find the length or the size of an array programmatically. Using the Length property, the For...Next loop from the previous code example can be written as follows

```
For i = 0 to myIntegers.Length - 1
    Console.WriteLine(myIntegers(i))
Next
```

This type of loop is very flexible and can be applied to an array of any size and of any data-type.

We can also understand the common description of the Main() Sub procedure. Main is also declared as

```
Public Main(ByVal args As String())
```

The command line arguments that we pass when executing our program are available in our programs through an array of String type, which is identified by args string array.

The For Each Loop

There is another type of loop that is very simple and useful to iterate through arrays and other collections. This is a For Each loop. The basic structure of the For Each loop is

```
For Each <identifier> in <array or collection>
        <statements or block of statements>
End For
```

Lets now make our previous code iterate through an array with a For Each loop

```
' demonstrates the use of arrays in VB.Net
Public Sub Main()
' declaring and initializing an array of type integer
Dim myIntegers() As Integer = New Integer() {3, 7, 2, 14, 65}
' iterating through the array and printing each element
Dim i As Integer
For Each i in myIntegers
    Console.WriteLine(i)
Next
End Sub
```

Simple and more readable! Isn't it?

```
For Each i in myIntegers
```

We declared the variable 'i' to hold individual values of array 'myIntegers' in each iteration. It is necessary to specify the type of variable 'i' same as the type of elements in the collection (Integer in our case).

Important points to note here are

- Variables are used to hold individual elements of an array in each iteration (i in the above example) are readonly. You can't change the elements of an array through it, you can only read it. This means For Each only allows you to iterate through the array or collection and not to change the contents of it. If you wish to perform some work on array elements such as to change the individual elements, you should use the For...Next loop.
- For Each can be used to iterate through arrays or collections. By collection, we mean any class, struct or interface that implements an IEnumerable interface. (Just go through this point and come back to it once we complete the lesson describing classes and interfaces)
- The String class is also a collection of characters (implements IEnumerable interface and returns the Char value in the Current property). The following code example demonstrates this and prints all the characters of a string.

```
Public Sub Main()
Dim name As String = "Faraz Rasheed"
Dim ch As Char
For Each ch in name
    Console.WriteLine(ch)
End For
End Sub
```

This will print each character of the name variable on a separate line.

Food for thought: Exercise 3

1. How was we able to call the Parse method on an Integer. The Integer is a data type and not a class?
2. Write a program that asks a user for 5 numbers and then print the average of these numbers.
3. Write a program that asks for 5 names. Find and print the name which has most number of characters in it. .
4. Write a program that processes the following string:

```
Dim sentence As String = "Learning VB.Net is extremely easy & fun, " & _
"specially at Programmers Heaven! VB.Net has a data type Byte, " & _
"which occupies 8 bits and can store integers values from -128 to 127."
```

Iterate through the string and find the number of letters, digits and punctuation characters in it and print these to the console.

5. Write a program that prints all the command line arguments.
6. What is the basic difference between And and AndAlso, Camel and Pascal Notation, Do While...Loop and the Do...Loop While statement?

Solution of Last Issue's Exercise (Exercise 2)

1. I wrote, compiled and executed these programs just like the programs of other languages. Where are all those .Net framework items I keep hearing about: CLR, JITers, MSIL, etc...?

The architecture of .Net applications in Windows is designed to look very similar to traditional Windows applications. When you compile your .Net application, the assembly (executable .exe or library .dll) file is generated. This file contains Win32 header, metadata and manifest and the MSIL code of modules in the assembly. You execute your application like any other Windows program by running the exe file. The Win32 header part of the exe invokes the CLR and passes a pointer to the entry point of your application to CLR. Now, CLR uses the JIT compiler to compile your code into a native executable as the functions are called. Hence, the .Net parts (framework, CLR, JITers, MSIL) do exist, but Microsoft has made the whole process transparent to the programmer.

2. How is it possible to call the methods of the Console Class when I did not create an object from it? (I know some Object Oriented Concepts from other languages)

WriteLine(), Write(), Read() and ReadLine() method of Console class are static, i.e., they belong to the class and not to the object. So, it's possible to call these functions using the name of the class without making an object from the class.

3. When there is more than one class in a program, how would the compiler recognize which class has the Main() method? If more than one class has a valid Main method, what does the compiler do then?

If more than one class is available, you will have to specify which class has Main or which class's Main is to be called using the /main parameter of the command line compiler:

```
vbc MyProg.vb /main:MyMainClass
```

In Visual Studio, right click the project in the solution explorer and select properties. Here (Common Properties - General - Startup), you can select which class in your program has Main() or which class's Main is to be used as the entry point of your program.

4. Is it possible to have more than one Main method in a program? If yes, then how would compiler decide which Main method to call when executing the program?

No, it's not possible to have multiple Main() methods in a single class. The compiler will complain that "Multiple Entry Points Found".

5. Write a program that asks for the name and phone number of the user at runtime. Print that data 3 times on the same line and then 3 times on seperate lines.

```
imports System
Namespace VBDotNetSchool
  Module TakingUserInput
    ' demonstrates how to take user input from console
    ' and displaying data on console window
    Public Sub Main()
        Console.Write("Plz, write your good name   : ")
        Dim name As String = Console.ReadLine()
        Console.Write("Plz, write your phone number: ")
        Dim phone As String = Console.ReadLine()
        Console.WriteLine()
        Console.WriteLine("Name: " & name & " Phone: " & phone)
        Console.WriteLine("Name: " & name & " Phone: " & phone)
        Console.WriteLine("Name: " & name & " Phone: " & phone)
        Console.WriteLine()
        Console.WriteLine("Name : " & name)
        Console.WriteLine("Phone: " & phone)
        Console.WriteLine()
        Console.WriteLine("Name : " & name)
        Console.WriteLine("Phone: " & phone)
        Console.WriteLine()
        Console.WriteLine("Name : " & name)
        Console.WriteLine("Phone: " & phone)
    End Sub
  End Module
End Namespace
```

6. How can you do compilation and execution separately using Visual Studio.Net (VS.Net)?

To compile your program select Build - Build Solution or press Ctrl+Shift+B. To execute your program, select Debug - Start Without Debug or press Ctrl+

What's Next...
Next time, we will be discussing how to deal with classes in objects in C#:

- What are classes, objects, member variables (fields), methods, properties?

- Making a sample class, instantiating and accessing its members
- Default values
- Access Modifiers (private, public, protected, internal, protected internal)
- Constructors, Destructors, Finalize() method
- Instance members, class members (static)
- Properties and its significance
- Overloading methods and constructors
- Value and Reference Types
- readonly and constants

Classes and Objects

Lesson Plan

Today we will start Object Oriented Programming (OOP) in VB.Net. We will start with learning classes, objects and their basics. Then we will move to constructors, access modifiers, properties, method overloading and Shared members of a class.

Concept of Class

A class is simply an abstract model used to define new data types. A class may contain any combination of encapsulated data (fields or member variables), operations that can be performed on the data (methods) and accessors to data (properties). For example, there is a class String in the System namespace of .Net Framework Class Library (FCL). This class contains an array of characters (data) and provide different operations (methods) that can be applied to its data like *ToLowerCase()*, *Trim()*, *Substring()*, etc. It also has some properties like *Length* (used to find the length of the string).

A class in VB.Net is declared using the keyword Class and its members are enclosed with the *End Class* marker

```
Class TestClass
' fields, operations and properties go here
End Class
```

Where *TestClass* is the name of the class or new data type that we are defining here.

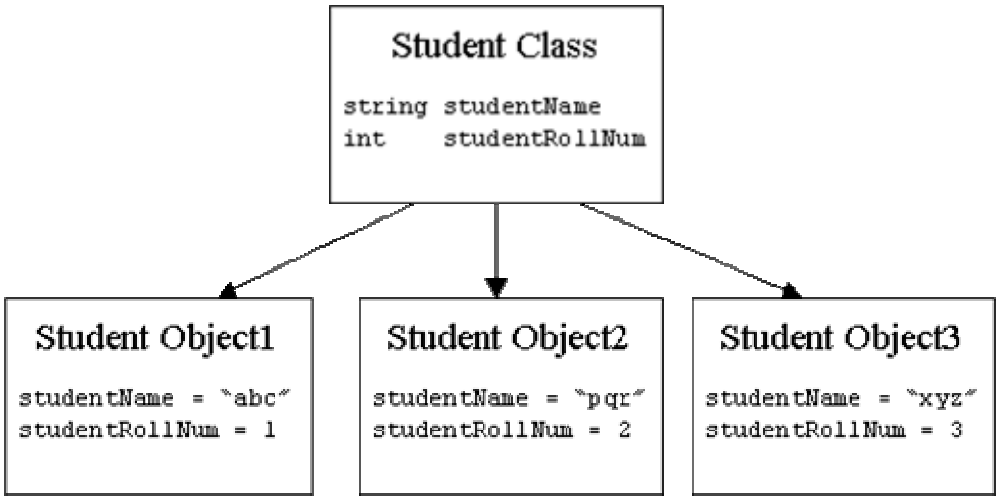
Objects

As mentioned above, a class is an abstract model. An object is the concrete realization or instance build on a model specified by the class. An object is created in memory using the *'New'* keyword and is referenced by an identifier called a "reference".

```
Dim myObjectReference As New TestClass()
```

In the line above, we made an object of type *TestClass* that is referenced by an identifier *myObjectReference*.

The difference between classes and implicit data types is that objects are reference types (passed by reference to methods) while implicit data types are value types (passed to methods by making a copy). Also, objects are created at heap while implicit data types are stored on a stack.



Fields

Fields are the data contained in the class. Fields may be implicit data types, objects of some other class, enumerations, structs or delegates. In the example below, we define a class named Student containing a student's name, age, marks in maths, marks in English, marks in science, total marks, obtained marks and a percentage.

```
Class Student
' fields contained in Student class
Dim name As String
Dim age As Integer
Dim marksInMaths As Integer
Dim marksInEnglish As Integer
Dim marksInScience As Integer
Dim totalMarks As Integer = 300   ' initialization
Dim obtainedMarks As Integer
Dim percentage As Double
End Class
```

You can also initialize the fields with the initial values as we did in the *totalMarks* in the example above. If you don't initialize the members of the class, they will be initialized with their default values. The default values for different data types are shown below

Data Type	Default Value
Implicit data types	
Integer	0
Long	0
Single	0.0
Double	0.0
Boolean	False
Char	'\0' (null character)
String	" " (an empty string)
Objects	Nothing

Methods - Sub Procedures and Functions
Methods are operations that can be performed on data. A method may take some input values through its parameters and may return a value of a particular data type. There are two types of methods in VB.Net: Sub Procedures and Functions.

Sub Procedure
A sub procedure is a method that does not return any values. A sub procedure in VB.Net is defined using the Sub keyword. A sub procedure may or may not accept parameters. A method that does not take any parameters is called a parameterless method. For example, the following is a parameterless sub procedure

```
Sub ShowCurrentTime()
    Console.WriteLine("The current time is: " & DateTime.Now)
End Sub
```

The above sub procedure takes no parameters and returns nothing. It does nothing more than print the Current Date and Time on console using the *DateTime* Class from the *System* namespace. A sub procedure may accept parameters as in the following example

```
Sub ShowName(ByVal myName As String)
    Console.WriteLine("My name is: " & myName)
End Sub
```

Above, the ShowName() sub procedure takes a String parameter *'myName'* and prints this string using the *Console.WriteLine()* method. The keyword *'ByVal'* with the *myName* parameter represents that the String *myName* will be passed to the method by value (i.e., by making a copy of the original string)

Functions
A function is a type of method that returns values. A function, like a sub-procedure may or may not accept parameters. For example, the following function calculates and returns the sum of two integers supplied as parameters.

```
Function FindSum(ByVal num1 As Integer, ByVal num2 As Integer) As Integer
Dim sum As Integer = num1 + num2
Return sum
End Function
```

We defined a function named *FindSum*, which takes two parameters of *Integer* type (num1 and num2) and returns a value of type *Integer* using the keyword *return*.

Instantiating the class
In VB.Net a class is instantiated (making its objects) using the *New* keyword.

```
Dim theStudent As New Student()
```

You can also declare the reference and assign an object to it in different steps. The following two lines are equivalent to the above example

```
Dim theStudent As Student
theStudent = New Student()
```

Note that it is very similar to using implicit data types except that the object is created with the *New* keyword while implicit data types are created using literals

```
Dim i As Integer
i = 4
```

Another important thing to understand is the difference between reference and object. The line

```
Dim theStudent As Student
```

only declares the reference *theStudent* of type *Student* which at this point does not contain an object (and points to the default *Nothing* value). If you try to access the members of the class (*Student*) through it, it will throw a *NullReferenceException* at runtime, which will cause the program to terminate. When we use,

```
theStudent = new Student()
```

a new object of the type Student is created at the heap and its reference (or handle) is given to *theStudent*. Only now, is it legal to access the members of the class through it.

Accessing the members of a class
The members of a class (fields, methods and properties) are accessed using the dot '.' operator against the reference of an object like

```
Dim theStudent As New Student()
theStudent.marksOfMaths = 93
theStudent.CalculateTotal()
Console.WriteLine(theStudent.obtainedMarks)
```

Lets make our Student class with some related fields, methods and then instantiating it in the Main() Sub Procedure.

```
Imports System
Module VBDotNetSchool
```

```
' Main method or entry point of program
Sub Main()
   Dim testStudent As New Student()
   testStudent.PerformOperations()
End Sub
End Module
' Defining a class to store and manipulate students information
Class Student
' Fields
Dim name As String
Dim age As Integer
Dim marksOfMaths As Integer
Dim marksOfEnglish As Integer
Dim marksOfScience As Integer
Dim totalMarks As Integer = 300
Dim obtainedMarks As Integer
Dim percentage As Double
'Methods
Sub CalculateTotalMarks()
obtainedMarks = marksOfMaths + marksOfEnglish + marksOfScience
End Sub
Sub CalculatePercentage()
percentage = obtainedMarks / totalMarks * 100
End Sub
Function GetPercentage() As Double
Return percentage
End Function
'Method to perform various operations on class and its objec
Public Sub PerformOperations()
Dim st1 As New Student()     ' creating new instance of Student
st1.name = "Einstein"        ' setting the values of fields
st1.age = 20
st1.marksOfEnglish = 80
st1.marksOfMaths = 99
st1.marksOfScience = 96
st1.CalculateTotalMarks()    ' calling Sub Procedures
st1.CalculatePercentage()
'calling and retrieving value returned by the function
Dim st1Percentage As Double = st1.GetPercentage()
Dim st2 As New Student()
st2.name = "Newton"
st2.age = 23
st2.marksOfEnglish = 77
st2.marksOfMaths = 100
st2.marksOfScience = 99
st2.CalculateTotalMarks()
st2.CalculatePercentage()
Dim st2Percentage As Double = st2.GetPercentage()
Console.WriteLine("{0} of {1} years age got {2}% marks", st1.name,
                  st1.age, st1.percentage)
Console.WriteLine("{0} of {1} years age got {2}% marks", st2.name,
                  st2.age, st2.percentage)
End Sub
End Class
```

Here, we started by creating an object of the *Student* class (*testStudent*) in the Main() sub procedure of the VBDotNetSchool module. Using the testStudent object, we called its PerformOperation sub procedure to demonstrate object manipulation

In the PerformOperation() method, we started by creating an object of the *Student* class (*st1*). We then assigned name, age and marks of the student. Later, we called methods to calculate the *totalMarks* and *percentage*, then we retrieved and stored the percentage in a variable and finally printed these on the console window. We repeated the same steps again to create another object of the type *Student* and set and printed its attributes. Hence in this way, you can create as many objects of the Student Class as you want.

When you compile and run this program it will display

Einstein of 20 years age got 91.6666666666667% marks
Newton of 23 years age got 92% marks

Access Modifiers or Accessibility Levels
In our Student class, every one has access to each of the fields and methods. So if one wants, he/she can change the totalMarks from 300 to say 200, resulting in the percentages going beyond 100%, which in most cases we like to restrict. VB.Net provides access modifiers or accessibility levels just for this purpose, i.e., restricting the access to particular members. There are 5 access modifiers that can be applied to any member of a class. These ate listed in the order of decreasing restriction along with a short description.

Access Modifier	Description
Private	Private members can only be accessed within the containing class
Friendly	Can only be accessed from the current project
Protected	Can be accessed from containing class and types inherited from the containing class
Public	Public members are accessible to anyone. Anyone who can see them can also access them.
Protected Friend	This type of members can be accessed from the current project **or** from the types inherited from their containing type

In Object Oriented Programming (OOP) it is always advised and recommended to mark all of your fields as private and allow the user of your class to only access certain methods by making them Public. For example, we may change our student class by marking all the fields Private and the three methods in the class as Public.

```
Class Student
' fields
Private name As String
Private age As Integer
Private marksOfMaths As Integer
Private marksOfEnglish As Integer
Private marksOfScience As Integer
Private totalMarks As Integer = 300
Private obtainedMarks As Integer
Private percentage As Double
' methods
Public Sub CalculateTotalMarks()
obtainedMarks = marksOfMaths + marksOfEnglish + marksOfScience
End Sub
Public Sub CalculatePercentage()
percentage = obtainedMarks / totalMarks * 100
End Sub
```

```
Public Function GetPercentage() As Double
Return percentage
End Function
End Class
```

If you don't mark any members of the class with a access modifier, it will be treated as a *Private* member; this means the default access modifier for the members of a class is *Private*.

You can also apply access modifiers to other types in VB.Net like *Class, Interface, Structure, Enum, Delegate* and *Event*. For top-level types (types not bound by any other type except namespace) like *Class, Interface, Structure* and *Enum* you can only use *Public* and *Friend* access modifiers with the same meaning as described above. In fact other access modifiers don't make sense to these types. Finally you cannot apply access modifier to the namespaces.

Properties

You must be wondering if we declare all the fields in our class as private, how can we assign values to them through their reference as we did in the Student class before? The answer is through Properties. VB.Net is the first language to introduce the support of defining properties in the language core. In traditional languages like Java and C++, for accessing the private fields of a class, public methods called getters (to retrieve value) and setters (to assign value) were defined like if we have a private field name

```
Private name As String
```

then, the getters and setters would be like

```
' getter to name field
Public Function GetName() As String
     Return name
End Function
' setter to name field
Public Sub SetName(ByVal theName As String)
     name = theName
End Sub
```

Using these we could restrict the access to a particular member. For example we can only opt to define the getter for the *totalMarks* field to make it read only.

```
Private totalMarks As Integer
Public Function GetTotalMarks() As Integer
     Return totalMarks
End Function
```

Hence outside the class, one can only read the value of *totalMarks* and cannot modify it. You can also decide to check conditions before assigning a value to a field

```
Private marksOfMaths As Integer
Public Sub SetMarksOfMaths(ByVal marks As Integer)
     If marks >= 0 And marks <=100 Then
     marksOfMaths = marks
     Else
     marksOfMaths = 0
     ' or throw some exception informing user marks out of range
End If
End Sub
```

This procedure gives you a lot of control over how fields in your classes should be accessed and dealt with. The problem is this, you need to define two methods and have to prefix the name of your fields with Get or Set. VB.Net provides built in support for these getters and setters in the form of properties

"Properties are context sensitive constructs used to read, write or compute private fields of class and to achieve the control over how the fields can be

accessed"

Using Properties
General Syntax for Properties is

```
<access modifier> Property <name of property> As <data type>
Get
' some optional statements
Return <some private field>
End Get
Set(ByVal <identifier> As <data type>)
' some optional statements
<some private field> = <identifier>
End Set
End Property
```

Didn't understand it? No problem lets clarify it with an example. We have a private field name

```
Private mName As String
```

To define a property for this, providing both getters and setters. Simply type

```
Public Property Name() As String
Get
    Return mName
End Get
Set(ByVal Value As String)
    mName = Value
End Set
End Property
```

We defined a property called *'Name'* and provided both a getter and a setter in the form of a *Get...End Get* and a *Set...End Set* blocks. Note that we called our property *'Name'* which is accessing the private member field *'mName'* (where m in mName denotes the member variable name). It is a convention to name the property similar to the corresponding field but with first letter in uppercase (for *mName->Name*, for *mPercentage->Percentage*). As properties are accessors to certain fields, they are mostly marked as *Public* while the corresponding field is (and should be) mostly marked as *Private*. Finally note in the *Set...End Set* block, we wrote

```
mName = Value
```

Above, a *Value* is the argument that is passed when a property is called.

In our program we will use our property as

```
Dim theStudent As New Student()
theStudent.Name = "Faraz"
Dim myName As String = theStudent.Name
theStudent.mName = "Someone not Faraz"     ' error
```

While defining properties, we said properties are context sensitive. When we write

```
theStudent.Name = "Faraz"
```

The compiler sees that the property *Name* is on the left hand side of assignment operator, so it will call the *Set...End Set* block of the property passing it *"Faraz"* as Value (which is its argument). In the next line...

```
Dim myName As String = theString.Name
```

The compiler now sees that the property *Name* is on the right hand side of the assignment operator, hence it will call the *Get...End* Get block of property Name which will return the contents of the *Private* field *mName* (*"Faraz"* in this case, as we assigned in line 2) which will be stored in the local string variable *myName*. Hence, when the compiler finds the use of a property, it checks in which context it is called and takes the appropriate action with respect to the context. The last line

```
theStudent.mName = "Someone not Faraz"     ' error
```

will generate a compile time error (if called outside the Student class) as the *mName* field is declared *Private* in the declaration of the class.

You can give the definition of either of *Get...End Get* or *Set...End Set* block. If the *Get...End Get* block is missing then the property must be marked as WriteOnly and similarly If the *Set...End Set* block is missing then the property must be marked as ReadOnly.

```
Public WriteOnly Property Password() As String
Set(ByVal Value As String)
    mPassword = Value
End Set
End Property
```

And,

```
Public ReadOnly Property Salary() As String
Get
   Return mSalary
End Get
End Property
```

If you miss one of these, and user tries to call it, he/she will get the compile time error. For example,

```
Dim mySalary As String
Employee.Salary = mySalary
```

The above lines will cause a compile time error (supposing the Salary property above is defined in the Employee class) as we did not provide any *Set...End Set* block of the *Salary* property.

You can also write statements in the *Get...End Get* or *Set...End Set* blocks as you do in methods.

```
Private mMarksOfMaths As Integer
Public WriteOnly Property MarksOfMaths() As Integer
Set(ByVal Value As Integer)
If Value >= 0 And Value <= 100 Then
marksOfMaths = Value
    Else
    marksOfMaths = 0
    ' or throw some exception informing user marks out of range
End If
End Set
End Property
```

Parameterized Property
A VB.Net property may be parameterized, that is, it may accept parameters when called. Consider the following property definition

```
Dim students() As String = {"Gates", "Newton", "Faraz"}
Public Property Student(ByVal rollNumber As Integer) As String
Get
  If rollNumber > 2 Or rollNumber < 0 Then
      Return "No Student"     ' or throw excpetion
  Else
      Return students(rollNumber)
  End If
End Get
Set(ByVal Value As String)
  If rollNumber >= 0 And rollNumber <= 2 Then
      students(rollNumber) = Value
      ' Else throw some exception
  End If
End Set
End Property
```

Here, we have defined a property that retrieves the name of a student based on the roll number supplied by the user. Before getting into details, lets see how the property is accessed

```
Dim s As String = Student(1)
Student(2) = "Fraz"
```

You can see that the Student property requires an integer as its parameter and a get/set of the student name in the property definition using this integer as index in the students array.

Inside the property definition, we have used the string array students. This array stores the name of students and the property considers the index of these names in the array as the student's roll number. In Get and Set blocks, we simply return or assign the names of particular students after checking the array bounds (minimum and maximum indices).

Default Property
A class may contain a default property which can be called using only the reference of a class. The default property is marked with the additional Default keyword and it must be a parameterized. For example, to make the 'Student' property (which we defined earlier) a default property, we only need to mark it with the Default keyword.

```
Class School
Dim students() As String = {"Gates", "Newton", "Faraz"}
Default Public Property Student(ByVal rollNumber As Integer) As String
Get
If rollNumber > 2 Or rollNumber < 0 Then
   Return "No Student"     ' or throw excpetion
Else
   Return students(rollNumber)
End If
End Get
```

```
Set(ByVal Value As String)
If rollNumber >= 0 And rollNumber <= 2 Then
   students(rollNumber) = Value
 ' Else throw some exception
End If
End Set
End Property
End Class
```

Now the Student property can be accessed using any object reference of the School class.

```
Sub Main()
Dim mySchool As New School()
Dim aStudentName As String = mySchool(1)
Console.WriteLine(aStudentName)
mySchool.Student(0) = "Ricky Ponting"
Console.WriteLine(mySchool(0))
End Sub
```

Here we did not use any property name with the reference *'mySchool'* to access the default property *'Student'*. Also note that we can still use the *Student* property as usual.

Important points about the use of properties
Finally some word of precautions while using properties

- *Set, Get, ReadOnly, WriteOnly, Default* and *Property* are keywords in VB.Net
- The default property is also called the Indexer for the containing class.
- The data type of a *Value* must be same as the type of the property you declared when declaring the property
- ALWAYS use correct indentation while using properties.

Constructors
Constructors are a special kind of sub procedure. A Constructor has the following properties

- It always has the name *'New'*
- Being a sub procedure, it does not return any value
- It is automatically called when a new instance or object of a class is created, hence called a constructor.

The constructor contains initialization code for each object like assigning default values to fields. Let us see some examples

```
Imports System
Class Person
' field
Private mName As String
' constructor
Public Sub New()
mNname = "unknown"
Console.WriteLine("Constructor called...")
End Sub
'property
Public Property Name As String
Get
```

```
Return mName
End Get
Set(ByVal Value As String)
mName = Value
End Set
End Property
End Class
```

In the *Person* class above, we have a *private* field *mName*, a *public* constructor which initializes the *mName* field with string value *"unknown"* and prints that when it is called. Then we have a *public* property to read/write private field *mName*. Lets make a module *Test* which contains the *Main()* method and which uses the *Person* class

```
Module Test
Public Sub Main()
Dim thePerson As new Person()
Console.WriteLine("The name of person in object thePerson is " & thePerson.Name)
thePerson.Name = "Faraz"
Console.WriteLine("The name of person in object thePerson is " & thePerson.Name)
End Sub
End Module
```

In our *Test* module, we made an object of the *Person* class and printed the name of the person. We then changed the value of the *Name* and printed the *Name* again. The result of the program is so

```
Constructor called…
The name of person in object thePerson is unknown
The name of person in object thePerson is Faraz
```

Note that the constructor is called just as we created a new instance of *Person* class and initialized the field *name* with the string *"unknown"*.

Infact, when we create a new object, we actually call the constructor of the class, see below

```
Dim thePerson As New Person()
```

That is why a constructor is usually made *public*. If you make your constructor *Private*, no one would be able to make an object of your class outside it. If the *Person* class is defined as

```
Class Person
Public Sub New()

End Sub
End Class
```

it would be an error to write our *Main()* method

```
Module Test
Public Sub Main()
Dim thePerson As new Person()      ' Error
```

```
End Sub
End Module
```

The constructor above is parameterless, i.e., it does not take any parameters. We can define a constructor, which takes values as its parameters we may define

```
Class Person
Private mName As String
Public Sub New(ByVal theName As String)
mNname = theName
Console.WriteLine("Constructor called...")
End Sub
End Class
```

Now, the object of the class *Person* can only be created by passing a string in the constructor.

```
Dim thePerson As New Person("Faraz")
```

If you don't define any constructors for your class, the compiler will generate an empty parameterless constructor for you. That is why we were able to make our *Student* object even when we did not specify any constructors for *Student* class.

Finalize() Method of Object class

Each class in VB.Net is automatically (implicitly) inherited from the *Object* class which contains the method *Finalize()*. This method is guaranteed to be called when your object is garbage collected (removed from memory). You can override this method and put here the code for freeing resources that you reserved when using the object. For example,

```
Protected Overrides Sub Finalize()
Try
Console.WriteLine("Destructing object")
'put some code here
Finally
MyBase.Finalize()
End Try
End Sub
```

Author's Note: I am not going to explain this code for now. If it looks alien to you; read it again when we have covered inheritance, polymorphism and exceptions. We will explain Garbage Collection in the coming chapters.

The *Finalize()* sub procedure is called automatically when the object is about to be destructed (when garbage collector is about to destroy your object to cleanup the memory). In the days of C++, programmers had to manage memory allocation and de-allocation explicitly. Destructors were used there to free the memory allocated by the object dynamically. But with VB.Net, the memory is managed automatically on behalf of your program. Hence, you probably won't encounter destructors or *Finalize()* methods that often.

Method and Constructor Overloading

It is possible to have more than one method (sub procedure and functions) with the same name and return type but with a different number and type of arguments (parameters). This is called the method of overloading. For example it is perfectly legal (code wise) to write

```
Class Checker
    ' 1st overloaded form
    Public Function IsDefaultValue(ByVal val As Boolean) As Boolean
        If val = False Then
            Return True
        Else
            Return False
        End If
    End Function
    ' 2nd overloaded form
    Public Function IsDefaultValue(ByVal val As Integer) As Boolean
        If val = 0 Then
            Return True
        Else
            Return False
        End If
    End Function
    ' 3rd overloaded form
    Public Function IsDefaultValue(ByVal intVal As Integer,
                                   ByVal booleanVal As Boolean) As Boolean
        If intVal = 0 And booleanVal = False Then
            Return True
        Else
            Return False
        End If
    End Function
    ' 4th overloaded form
    Public Sub IsDefaultValue(ByVal val As Double)
        If val = 0 Then
            Console.WriteLine("Zero")
        Else
            Console.WriteLine("Non-Zero")
        End If
    End Sub
End Class
```

In the Checker class above we defined four methods (3 functions and 1 sub procedure) with the name *IsDefaultValue()*. The return type of all the functions is *Boolean* but all differ from each other in the parameter list. The first two and four differ in the data type of parameters while the third one differs in the number of parameters. When the *IsDefaultValue()* is called, the compiler will decide on the basis of parameter which one of these three to actually call. For example, in our *Main()* method

```
Dim ch As New Checker()
        Console.WriteLine(ch.IsDefaultValue(5))         ' will call the second one
        Console.WriteLine(ch.IsDefaultValue(False))     ' will call the first one
        Console.WriteLine(ch.IsDefaultValue(0, True))   ' will call the third one
        ch.IsDefaultValue(0.0)                          ' will call the fourth one
```

But remember, methods are overloaded depending on the parameter list and not on the return type. The return type must be same or one of the methods is a sub procedure(does not have return type). The *WriteLine()* method of the *Console* class in the *System* namespace has as much as 19 different overloaded forms! See .Net Framework Documentation or MSDN for all these.

Overloading Constructors
Since, constructors are a special type of method (a sub procedure), we can overload constructors similarly.

```
Class Person
Private name As String
Public Sub New()
    name = "uknown"
End Sub
Public Sub New(ByVal theName As String)
    name = theName
End Sub
End Class
```

Now, if we create an object like

```
Dim thePerson As New Person()
```

First the constructor will be called initializing the *name* with *"unknown"* and if we create an object like

```
Dim thePerson As New Person("Faraz")
```

The constructor will be called initializing the *name* with *"Faraz"*. As you can see, overloading methods and constructors gives your program a lot of flexibility and reduces a lot of complexity that would otherwise be produced if we had to use different name for these methods (Just consider what would happen to implementers of *WriteLine()*! From where they would arrange 19 names!)

The Me keyword or the Me reference
Each object has a reference *this* which points to itself. Suppose in some method calls, our object needs to pass to itself, what would we do? Suppose in our class Student, we have a method *Store()* that stores the information of *Student* on the disk. In this method, we called another method *Save()* of the *FileSystem* class which takes the object to store as its parameter.

```
Class Student
Dim name As String = "Some Student"
Dim age As Integer
Public Sub Store()
Dim fs As New FileSystem()
fs.save(Me)
End Sub
End Class
```

We passed *Me* as a parameter to the method *Save()* which points to the object itself.

```
Module VBDotNetSchool
Sub Main()
Dim theStudent As New Student()
theStudent.Store()
End Sub
End Module
```

When the *Store()* is called, the reference *theStudent* will be passed as a parameter to *Save()* method in *Store()*. Conventionally, the parameters to the constructors and other methods are named the same as the name of the fields they refer to and are distinguished only using the *Me* reference.

```
Class Student
Private name As String
Private age As Integer
Public Sub New(ByVal name As String, ByVal age As Integer)
   Me.name = name
   Me.age = age
End Sub
End Class
```

In the constructor, when we use *name* or *age*, we actually get the variables passed in the method which overshadowed the instance members (fields) with same name. Hence, to use our fields, we had to use the Me reference to distinguish our instance members (fields) with the members passed through the parameters.

Me is an extremely useful, widely and commonly used construct. I recommend you to practice with *Me* for some time until you feel comfortable with *Me*.

The MyClass keyword or the MyClass reference

The *MyClass* keyword is similar to the *Me* keyword except that its behavior does not change with polymorphism. The *MyClass* reference always refers the members of the containing class while *Me* refers the members of the current object. The following example demonstrates the difference between the *MyClass* and the *Me* references

```
Imports System
Module VBDotNetSchool
Sub Main()
Dim thePerson As Person
thePerson = New Student()
thePerson.UseMe()       ' Prints Student's MyMethod
thePerson.UseMyClass()  ' Prints Person's MyMethod
End Sub
End Module
Class Person
Public Overridable Sub MyMethod()
   Console.WriteLine("Person's MyMethod")
End Sub
Public Sub UseMe()
   Me.MyMethod()   ' Use calling class's version, even if an override.
End Sub
```

```
Public Sub UseMyClass()
   MyClass.MyMethod()    ' Use this version and not any override.
End Sub
End Class
Class Student
Inherits Person
Public Overrides Sub MyMethod()
   Console.WriteLine("Student's MyMethod")
End Sub
End Class
```

The above example shows that the *MyClass* reference always accesses the members of the defining class while the *Me* reference accesses the members of the current object's class.

Author's Note: Inheritance and Polymorphism will be covered in next lessons. So the above code is not meant to be understood right now. But do remember to comeback to it after having the understanding of inheritance and polymorphism in VB.Net

---

Shared Members of the class

All the members of the class that we have seen up until now are instance members. This means they belong to the object being created. For example, if you have an instance field *name* in your *Person* class then each object of our *Person* class will have a separate field *name* of its own. There is another class of members which is called *Shared*. Shared members belong to the whole class rather than to an individual object. For example, if you have a *Shared phoneNumber* field in your *Student* class, then there will be a single instance of this field and all the objects of this class will share this single field. Changes made by one object to *phoneNumber* will be realized by the other object. Shared members are defined using the keyword *Shared*

```
Class Student
Public Shared phoneNumber As Integer
Public rollNumber As Integer
End Class
```

Shared members are accessed with the name of the class rather than object references. Let's make our *Test* Module containing the *Main* method

```
Module Test
Sub Main()
Dim st1 As New Student()
Dim st2 As New Student()
st1.rollNumber = 3
st2.rollNumber = 5
Student.phoneNumber = 4929067
End Sub
End Module
```

Here you can see that the *phoneNumber* is accessed without an object reference but with the name of the class it belongs to. Shared methods are very useful while programming. In fact, the *WriteLine()* and *ReadLine()* methods that we used from the start are *Shared* methods of the *Console* class. That is the reason why we called them with the reference to their class rather than making an object of the *Console* class. Shared variables are useful when you want to cache data that should be available to all objects of a class. You can use *Shared* fields, methods, properties and even constructors which will be called before an instance of a class is created. A Shared constructor is declared like so

```
Shared Sub New()
name = "unknown"
End Sub
```

As *Shared* methods may be called without a reference to the object, you can not use instance members inside shared methods or properties, while you may call a shared member from a non-shared context. The reason for being able to call shared members from non-shared contexts is that shared members belong to the class and are present irrespective of the existence of even a single object. The definition of *MyMethod()* in following code will not compile

```
Class Student
Public Shared phoneNumber As Integer
Public rollNumber As Integer
```

```
Public Sub DoWork()
  MyMethod()  ' legal, static method called in non-static context
End Sub
Public Shared Sub MyMethod()
  phoneNumber = phoneNumber + 1 ' legal,
                                ' static field used in static context
  rollNumber = rollNumber + 1   ' illegal,
                                ' non-static field used in static context
  DoWork()                      ' illegal,
                                ' non-static method called in static context
End Sub
End Class
```

Some precautionary points in the end

- Don't put too many shared methods in your class as it is against the object oriented design principles and makes class less extensible.
- Don't try to make a class with only shared methods and properties unless you have a very good reason for doing this (where good means very very good). For this you can use VB.Net modules (which we will cover in the coming lessons)
- You tend to loose a number of object oriented advantages while using shared methods as shared methods can't be overridden which means it can not be used polymorphically. This is something that is widely used in the Object Oriented Paradigm of programming.

Some More about Methods
We mentioned earlier that there are two kind of 'types' in VB.Net: Value type and Reference type. Value types are like implicit data types and are passed to methods by value and reference types like objects and arrays are passed by reference.

Value types (ByVal, ByRef & Optional Keywords)
When we pass a variable of implicit data type to a method, the compiler generates a copy and passes that copy to a method. It is actually a copy of the variable that is available inside the method. Hence if you modify the value type variable (passed as a parameter) in a method, the actual value of the variable will not be changed outside the method. We have in our test class the *Main()* the method and the *DoWork()* method as follows

```
Module Test
Public Sub Main()
   Dim a As Integer = 3
   DoWork(a)
   Console.WriteLine("The value of a is " & a)
End Sub
Public Sub DoWork(ByVal I As Integer)
   i += 1
End Sub
End Module
```

The program will result

The value of a is 3

Because a copy of the variable *a* is passed to the *DoWork()* method and not the variable *a*.

Also, note that *i* is the local variable in *DoWork()* and *a* is local variable in *Main()*. Hence, they can only be accessed within their containing methods. In fact, we may define *Dim a As Integer* in different methods and each will have its own variable *a* and none will have correspondence with another implicitly.

VB.Net provides a keyword **ByRef**, which means that the value type will be passed by reference instead of the default by value behavior. Hence, changes done inside the method will be reflected back after the method has been called and terminated.

```
Module Test
Public Sub Main()
Dim a As Integer = 3
DoWork(a)
Console.WriteLine("The value of a is " & a)
End Sub
Public Sub DoWork(ByRef i As Integer) ' note ByRef
   i += 1
End Sub
End Module
```

The program will result

The value of a is 4

In the case of the *ByRef* keyword, if the variable is not initialized before passing to the method by reference, the default value of an Integer (i.e., zero) will be passed to the calling method. VB.Net also provides the **ByVal** keyword. This is used to pass variables by making a copy.

The parameters of VB.Net methods (sub procedures and functions) can be marked as 'Optional'. The optional parameters must be the final parameter of a method. A method with an Optional parameter can be used either with the optional parameter or without the optional parameter; hence they are so called. The important point about an Optional parameter is that they must be specified with the default value. For example, look at the following method definition

```
Public Sub DoWork(ByVal i As Integer, Optional ByVal j As Integer = 3)
   Console.WriteLine("i = " & i)
   Console.WriteLine("j = " & j)
End Sub
```

It specifies two parameters; i is a normal Integer parameter while j is an optional parameter with default value of 3. The method can be called with either of these two invocation calls

```
Console.WriteLine("First Call: DoWork(2)")
DoWork(2)        ' i = 2, j = 3
Console.WriteLine()
Console.WriteLine("Second Call: DoWork(4, 1)")
DoWork(4, 1)     ' i = 4, j = 1
```

In the first invocation call (DoWork(2)), the method is called without the optional parameter; hence the default value of (3) will be used as the value of j inside the DoWork()

method. The result of two invocation calls will be

```
First Call: DoWork(2)


i = 2
j = 3

Second Call: DoWork(4, 1)
i = 4
j = 1
```

Reference types
Objects are implicitly passed by reference. It means that only a copy of the reference is passed to the methods during method invocation. Hence, if we initialize an array (which is object in VB.Net) and pass it to a method where the array gets changed, then this changed effect would be visible after the method has been terminated in the actual calling method.

```
Module Test
Public Sub Main()
Dim nums() As Integer
nums = New Integer() {2, 4, 8}
Dim num As Integer
Dim count As Integer = 0
Console.WriteLine("Before Calling DoWork(nums): ")
For Each num In nums
    Console.WriteLine("The value of a({0}) is {1}", count, num)
    count += 1
Next
DoWork(nums)
Console.WriteLine()
Console.WriteLine("After Calling DoWork(nums): ")
count = 0
For Each num In nums
     Console.WriteLine("The value of a({0}) is {1}", count, num)
     count += 1
Next
End Sub
Public Sub DoWork(ByRef numbers() As Integer)
Dim i As Integer
For i = 0 To numbers.Length - 1
    numbers(i) += 1
    Next
End Sub
End Module
```

The program will result

```
Before Calling DoWork(nums):
The value of a(0) is 2
The value of a(1) is 4
The value of a(2) is 8

After Calling DoWork(nums):
The value of a(0) is 3
The value of a(1) is 5
The value of a(2) is 9
Press any key to continue
```

Here, we initialized an *Integer* type array (*nums*) with some values. We passed this array in the *DoWork()* method, which incremented (modified) the contents of the array. Finally, we printed the elements of array. As the output suggests, *DoWork()* method did change the elements in the array (*nums*) and worked on the actual array and not on its copy (as is the case in value types).

Some more about reference and object

Reference is just a pointer or handle to the object in memory. It is not possible in VB.Net to create an object without giving its handle to a reference like

```
New Student()     ' Compile time syntax error
```

The above line is invalid statement. It will try to create an object of the *Student* class without a reference pointing to it. We must use create an object like this

```
Dim theStduent As New Student(87, 94, 79)
theStudent.CalculatePercentage()
theStudent = Nothing
```

Here, we created a new object of the Student class and called one of its methods. Finally we assigned *Nothing to theStudent* so the object above will be destroyed after the method call terminates. When you write

```
Dim student1 As New Student("Faraz")
```

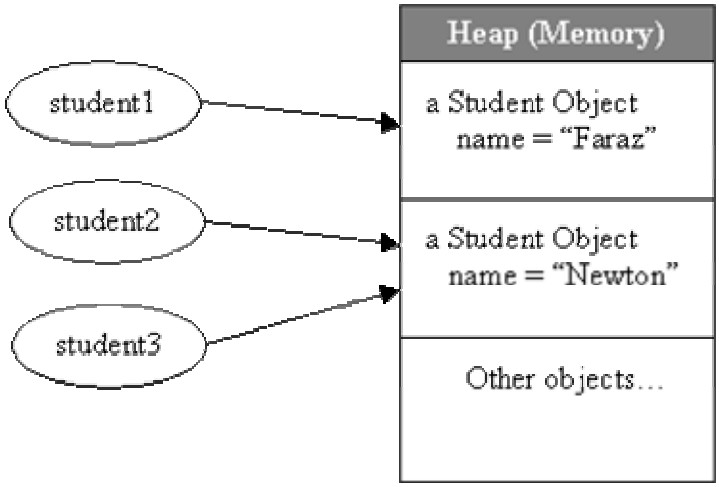
a new object of type *Student* is created at the heap and its handle is given to the reference *student1*. Let us make another object and give its handle to reference *student2*.

```
Dim student2 As New Student("Newton")
```

Now, if we write

```
Dim student3 As Student = student2
```

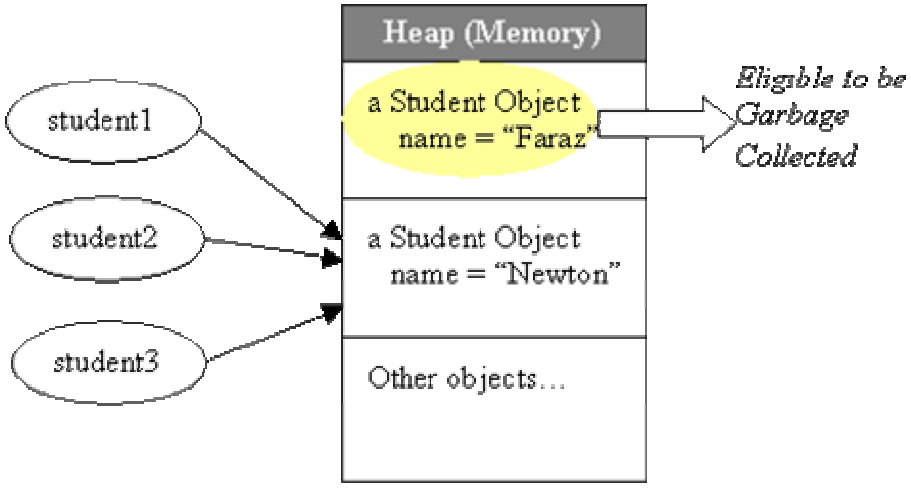
The new reference *student3* will also start pointing at student (Newton) which is already being pointed at by *student2*. Hence both *student2* and *student3* will be pointing to same student and both can make changes to that object.



Now if we write,

```
student1 = student2
```

it will also start pointing to the second object (Newton), leaving the first student (Faraz) unreferenced. It means the first student is now eligible to be garbage collected and can be removed from memory at anytime.



If you want a reference to reference nothing, you can equate it to *Nothing*, which is a keyword in Visual Basic.Net

```
student1 = Nothing
```

Food for thought: Exercise 4

1. What is the difference between a class and an object? Why do we make classes?
2. How are fields and properties different from each other?
3. Isn't it better to make a field *Public* than providing its property with both *Set...End Set* and *Get...End Get* blocks? After all, the property will allow the user to both read and modify the field, so why not use a *Public* field instead? Motivate your answer
4. Write a class to simulate a Circle Shape. Each circle has its radius and center point

- (a) Provide 2 constructors: a parameterless and one which takes a center point and a radius as parameters.
- (b) Provide a mechanism to find the diameter (twice the radius) and the area (PI * square of radius).
- (c) Provide a mechanism to move a circle from its starting location to somewhere else.
- (d) Apply proper access modifiers and use proper fields, method and properties

5. Write a program containing a method that will swap (interchange) the value of two integer type variables.
6. Can we use *Me* in a *Shared* context? Why or Why not? Can *Me* ever be equal to *Nothing*?

Solution of Last Issue's Exercise (Exercise 3)

1. How was we able to call the Parse method on an Integer. The Integer is a data type and not a class?

All the primitive data types in VB.Net (or any other .Net compliant language) are mapped to types defined in the CTS (Common Type System). These CTS types are implemented as Structures (something like classes, we will see the details of structures in coming lessons) and contain different useful methods and properties. An *Integer* in VB.Net is mapped to the CTS type *Int32*, which contains a method called *Parse()*. When we tried to call the *Parse()* method on an *Integer*, the CLR automatically boxed it into a *Int32* and called the *Parse()* method. We will see boxing and un-boxing in the next lesson (Lesson 5).

2. Write a program that asks a user for 5 numbers and then print the average of these numbers.

```
Imports System
' Program that input and calculate the average of 5 numbers
Module NumberAverage
    Sub Main()
        Dim numbers(4) As Integer
        Dim sum As Integer = 0
        Dim average As Double = 0
        Dim i As Integer
        For i = 0 To numbers.Length - 1
            Console.Write("Plz, write the {0}th number: ", i + 1)
            numbers(i) = Integer.Parse(Console.ReadLine())
            sum += numbers(i)
        Next
        average = sum / numbers.Length
        Console.WriteLine(vbCrLf & "The average of given {0} numbers is: {1}",
```

```
                            numbers.Length, average)
    End Sub
End Module
```

3. Write a program that asks for 5 names. Find and print the name which has most number of characters in it.

```
Imports System
' Program that input and find the biggest of the 5 names
Module BiggestName
    Sub Main()
        Dim names(2) As String
        Dim biggestLength As Integer = 0
        Dim biggestName As String = ""
        Dim i As Integer
        For i = 0 To names.Length - 1
            Console.Write("Plz, write the {0}th name: ", i + 1)
            names(i) = Console.ReadLine()
            If names(i).Length > biggestLength Then
                biggestName = names(i)
                biggestLength = names(i).Length
            End If
        Next
        Console.WriteLine(vbCrLf & "The biggest of the given {0} names is:
{1} with {2} characters.", names.Length, biggestName, biggestLength)
    End Sub
End Module
```

4. Write a program that processes the following string:

```
Dim sentence As String = "Learning VB.Net is extremely easy & fun, " & _
"specially at Programmers Heaven! VB.Net has a data type Byte, " & _
"which occupies 8 bits and can store integers values from -128 to 127."
```

Iterate through the string and find the number of letters, digits and punctuation characters in it and print these to the console.

```
Imports System
' Program that finds the number of letters, digits
' and punctuation characters in string
Module DigitLetter
Sub Main()
Dim sentence As String = "Learning VB.Net is extremely easy & fun, " & _
 "specially at Programmers Heaven! VB.Net has a data type Byte, " & _
    "which occupies 8 bits and can store integers values from -128 to 127."
Dim letters As Integer = 0
Dim digits As Integer = 0
Dim punctuators As Integer = 0
Dim ch As Char
For Each ch In sentence
   If Char.IsLetter(ch) Then
   letters += 1
```

```
   ElseIf Char.IsDigit(ch) Then
   digits += 1
   ElseIf Char.IsPunctuation(ch) Then
   punctuators += 1
   End If
Next
Console.WriteLine("Int the sentence: " & vbCrLf & sentence & vbCrLf)
Console.WriteLine("The number of charachters = {0}", sentence.Length)
Console.WriteLine("The number of letters     = {0}", letters)
Console.WriteLine("The number of digits      = {0}", digits)
Console.WriteLine("The number of punctuators = {0}", punctuators)
End Sub
End Module
```

5. Write a program that prints all the command line arguments.

```
Imports System
' program that prints all the command line arguments
Module DigitLetter
Sub Main(ByVal args() As String)
Dim i As Integer
For i = 0 To args.Length - 1
    Console.WriteLine("The {0}th command line argument:
    {1}", i, args(i))
Next
End Sub
End Module
```

6. What is the basic difference between And and AndAlso, Camel and Pascal Notation, Do While...Loop and the Do...Loop While statement?

And and AndAlso operators: AndAlso is used for short circuit evaluation of Boolean values while And is commonly used as a bitwise AND operator. While using AndAlso in comparison, it will terminate evaluating comparisons once it find the false value, hence it is more efficient in a comparison scenario. While And will check each comparison statement before returning the result.

Camel and Pascal Notation: In camel notation, the first character of the identifier appears in lowercase while in Pascal notation, the first character of identifier appears in uppercase. In both notations the successive words start with uppercase letters while the remaining letters appear in lowercase. In VB.Net camel notation is used for naming fields and Pascal notation is used for naming Methods and Types (Class, Interface, Enum, Structure)

Do While...Loop and Do...Loop While: In a *Do Loop...While*, the condition is checked after each iteration whereas in a *Do While...Loop*, the condition is checked before each iteration.

What's Next...

Next time, we will be discussing how to deal Inheritance and Polymorphism in VB.Net. We will see

- What is inheritance? Its significance
- Implementing inheritance in VB.Net and its constraints
- Using Protected access modifier
- Using MyBase keyword
- Object Class - the base of all classes
- Polymorphism and its importance
- Implementing with Overridable, Overrides and Shadows modifiers
- Boxing and Unboxing
- Casting (Up, Down-Casting)

Inheritance & Polymorphism in VB.Net

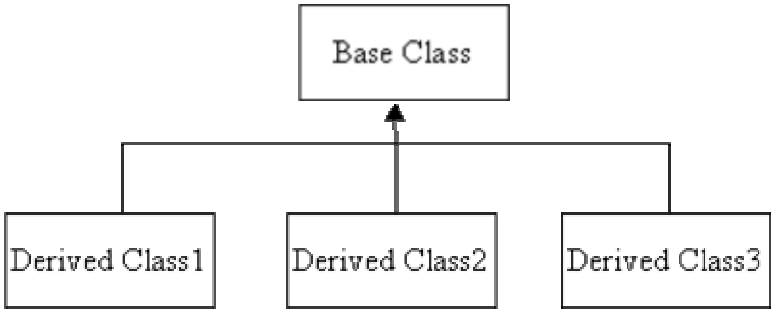
Lesson Plan

We will learn the fundamental object oriented features such as inheritance and polymorphism in VB.Net. We will start with building some understanding of inheritance and then will move towards understanding how VB.Net supports inheritance. We will spend some time exploring the *Object* class and then we will head towards polymorphism and how polymorphism is implemented within VB.Net. We will close this session with the understanding of boxing, un-boxing and how the type-casting mechanism works in VB.Net

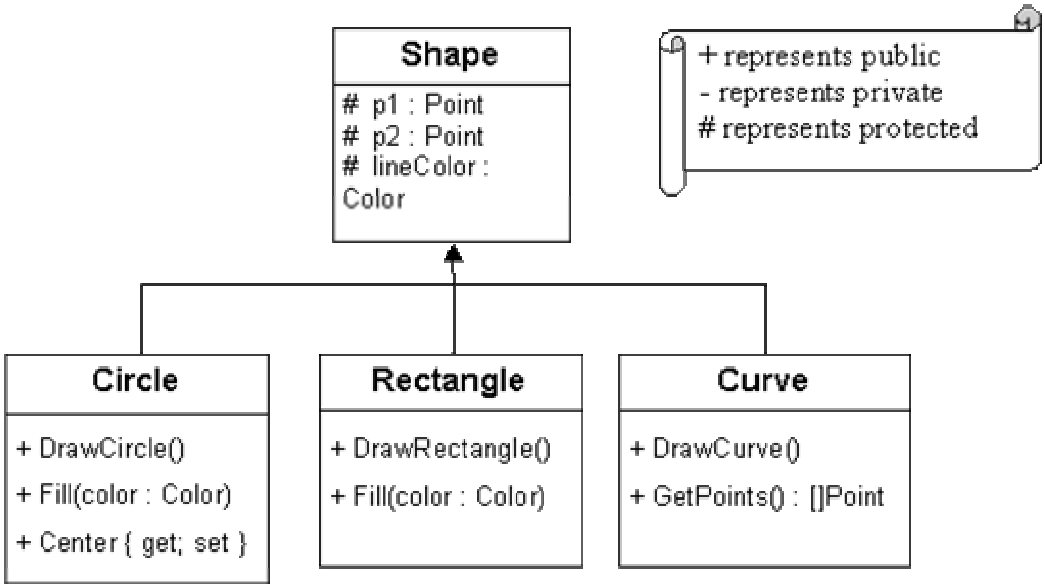
Inheritance

Unless this is your first brush with object oriented programming, you will have heard a lot about Reusability and Extensibility. Reusability is the property of a module (a component, class or even a method) that enables it to be used in different applications without any or little change in its source code. Extensibility of a module is it's potential to be extended (enhanced) as new needs evolve. Reusability in Object Oriented Programming languages is achieved by reducing coupling between different classes while extensibility is achieved by sub-classing. The process of sub-classing a class to extend its functionality is called Inheritance or sub-typing.

The original class (or the class that is sub-typed) is called the base, parent or super class. While the class that inherits the functionality of base class and extends it in its own way is called a sub, child, derived or an inherited class.



An Example



In the figure above, we have used a UML's (Unified Modeling Language) class diagram to show Inheritance. Here, *Shape* is the base class while *Circle, Rectangle* and *Curve* are its sub-classes.

The Base class usually contains general functionality while the sub-classes possess specific functionality. So, when sub-classing or inheriting, we go *'from specialization to generalization'*.

If class B (sub class) inherits class A (base class), then B would have a copy of all the instance members (fields, methods, properties) of class A, and B can access all the members (except for private members) of class A. Private members of the base class **do** get inherited in the sub-class but they can **not** be accessed by the sub-class. Also, inheritance is said to create a **'type of'** relationship among classes which means sub-classes are a type of base class. (If this is not clear, don't worry, things will get more clear when we

look at implementing inheritance in the following sections)

Inheritance in VB.Net

Before we go on to implementation, here are some key-points regarding inheritance in VB.Net

- VB.Net, like C#, Java and contrary to C++, allows only single class inheritance. Multiple inheritance of classes is **not** allowed in VB.Net
- The Object class defined in the *System* namespace is implicitly the ultimate base class of all the classes in VB.Net (and the .Net framework)
- Interfaces, in VB.Net, can inherit more than one interface. So, multiple inheritance of interfaces is allowed in VB.Net (again similar to C# and Java). We will see interfaces in detail in the coming lessons
- Structures in VB.Net, can only inherit (or implement) interfaces and can not be inherited.

Author's Note: Technically, a class inherits another class while it implements an interface. It is technically wrong to say that class A inherits interface B. Rather, it should be 'class A implements interface B'. Although, many write-ups don't follow this path, I would recommend using the word 'inherits' only where it makes sense.

Implementing inheritance in VB.Net

VB.Net uses the *Inherits* keyword to indicate inheritance. Suppose we have a class named *Student* with the following fields; *mRegistrationNumber, mName* and *mDateOfBirth* along with the corresponding properties. The class also has a function called *GetAge()* which calculates and returns the age of a Student.

```
Class Student
    ' private Fields
    Private mRegistrationNumber As Integer
    Private mName As String
    Private mDateOfBirth As DateTime
    ' Student Constructor
    Public Sub New()
        Console.WriteLine("New student created. Parameter less constructor called...")
    End Sub
    Public Sub New(ByVal pRegistrationNumber As Integer, ByVal pName As String,
                   ByVal pDateOfBirth As DateTime)
        mRegistrationNumber = pRegistrationNumber
        mName = pName
        mDateOfBirth = pDateOfBirth
        Console.WriteLine("New Student Created.Parameterized constructor called...")
    End Sub
    ' public Properties
    Public ReadOnly Property RegisterationNumber() As Integer
        Get
            Return mRegistrationNumber
        End Get
    End Property

    Public Property Name() As String
        Get
            Return Name
        End Get
        Set(ByVal Value As String)
            Name = Value
        End Set
    End Property
    Public Property DateOfBirth() As DateTime
        Get
            Return DateOfBirth
        End Get
        Set(ByVal Value As DateTime)
            DateOfBirth = Value
        End Set
    End Property
    ' public Function
    Public Function GetAge() As Integer
        Dim age As Integer = DateTime.Now.Year - DateOfBirth.Year
        Return age
    End Function
End Class
```

The *Student* class above is very simple. We have defined three *Private* fields, their accessor properties and one function to calculate the age of a student. We have defined two constructors: the first one takes no parameters and the other takes the values of three parameters. Note that we have only defined the *Get* property of *mRegistrationNumber* since

we don't want the user of the *Student* class to change the *mRegistrationNumber* once it is assigned through constructor. Also, note that we did not make *GetAge()* a property but a function. The reason for this is that properties are generally supposed to be accessors for getting/setting values of fields and not for calculating/processing data. Hence, it makes sense to declare *GetAge()* as a function.

Let us declare another class named *SchoolStudent* that inherits the *Student* class but with additional members such as marks of different subjects and methods for calculating total marks and percentages.

```
Class SchoolStudent
    Inherits Student
    ' Private Fields
    Private mTotalMarks As Integer
    Private mTotalObtainedMarks As Integer
    Private mPercentage As Double
    ' Public Constructors
    Public Sub New()
        Console.WriteLine("New school student created. Parameter less constructor
called...")
    End Sub
    Public Sub New(ByVal pRegNum As Integer, ByVal pName As String, ByVal pDob As
DateTime, _
                   ByVal pTotalMarks As Integer, ByVal pTotalObtainedMarks As Integer)
        ' call to base class constructor
        MyBase.New(pRegNum, pName, pDob)
        mTotalMarks = pTotalMarks
        mTotalObtainedMarks = pTotalObtainedMarks
        Console.WriteLine("New school student is created. Parameterized constructor
called...")
    End Sub
    ' Public Properties
    Public Property TotalMarks() As Integer
        Get
            Return mTotalMarks
        End Get
        Set(ByVal Value As Integer)
            mTotalMarks = Value
        End Set
    End Property
    Public Property TotalObtainedMarks() As Integer
        Get
            Return mTotalObtainedMarks
        End Get
        Set(ByVal Value As Integer)
            mTotalObtainedMarks = Value
        End Set
    End Property
    ' Public Function
    Public Function GetPercentage() As Double
        mPercentage = TotalObtainedMarks / TotalMarks * 100
        Return mPercentage
    End Function
End Class
```

Our *SchoolStudent* class inherits *Student* class by using the *Inherits* keyword

```
Class SchoolStudent
```

```
    Inherits Student
```

The *SchoolStudent* class inherits all the members of the *Student* class. In addition, it also declares its own members: three private fields (*mTotalMarks, mTotalObtainedMarks* and *mPercentage*) with their corresponding properties, two constructors (a parameter less one and a parameterized one) and one instance the function (*GetPercentage()*). For now, forget about the second (parameterized) constructor of our *SchoolStudent* class. Lets make our *Test* Module and *Main()* method.

```
' program to demonstrate inheritance
Module Test
    Public Sub Main()
        Dim st As New Student(1, "Fraz", New DateTime(1980, 12, 19))
        Console.WriteLine("Age of student, {0}, is {1}" & vbCrLf, _
                          st.Name, st.GetAge())
        Dim schStd = New SchoolStudent()
        schStd.Name = "Newton"
        schStd.DateOfBirth = New DateTime(1981, 4, 1)
        schStd.TotalMarks = 500
        schStd.TotalObtainedMarks = 476
        Console.WriteLine("Age of student, {0}, is {1}. {0} got {2}% marks.", _
   schStd.Name, schStd.GetAge(), schStd.GetPercentage())
    End Sub
End Module
```

In the *Main()* method, first we made an object of the *Student* class (*st*) and printed the *name* and *age* of a Student *st*. Next, we made an object of the *SchoolStudent* class (*schStd*). Since, we used parameter less constructor to instantiate *schStd*, we set the values of its fields through properties and then printed the name, age and percentage of *SchoolStudent (schStd)*. Note that we are able to access the properties *Name* and *DateOfBirth* (defined in *Student* class) because our *SchoolStudent* class is inherited from the *Student* class, thus inheriting the public properties too. When we execute the above program, we get the following output

```
New Student Created. Parameterized constructor called...
Age of student, Fraz, is 24

New student created. Parameter less constructor called...
New school student created. Parameter less constructor called...
Age of student, Newton, is 23. Newton got 95.2% marks.
Press any key to continue
```

The output of the first two lines is as expected. But, notice the output when we create the *SchoolStudent* object. First, the parameter less constructor of *Student* is called and then the constructor of *SchoolStudent* is called.

```
New student created. Parameter less constructor called...
New school student created. Parameter less constructor called...
```

Constructor calls in Inheritance

Infact, when we instantiate the sub-class (*SchoolStudent*), the compiler first instantiates the base-class (*Student*) by calling one of its constructors and then calls the constructor of the sub-class. Suppose now we want to use the second constructor of the *SchoolStudent* class. For this, we need to comment the line with the *MyBase* keyword in *SchoolStudent*'s second constructor declaration and make other changes as below:

```
Public Sub New(ByVal pRegNum As Integer, ByVal pName As String, ByVal pDob As
DateTime,
               ByVal pTotalMarks As Integer, ByVal pTotalObtainedMarks As Integer)
' call to base class constructor
' MyBase.New(pRegNum, pName, pDob)
Me.Name = pName
Me.DateOfBirth = pDob
mTotalMarks = pTotalMarks
mTotalObtainedMarks = pTotalObtainedMarks
Console.WriteLine("New school student is created. Parameterized constructor
called...")
End Sub
```

This constructor takes as parameters the fields defined by *SchoolStudent* and those defined by its base class (*Student*). Also it sets the values accordingly using properties. Now, we need to change the Main method as

```
Public Sub Main()
Dim schStd As New SchoolStudent(2, "Newton", _
New DateTime(1983, 4, 1), 500, 476)
Console.WriteLine("Age of student, {0}, is {1}. {0} got {2}% marks.", _
schStd.Name, schStd.GetAge(), schStd.GetPercentage())
End Sub
```

In the *Main()* method, we made an object of *SchoolStudent* using a parameterized constructor and then printed the *name, age* and *percentage* of the *SchoolStudent*. The output of the program is

```
New student created. Parameter less constructor called...
New school student is created. Parameterized constructor called...
Age of student, Newton, is 21. Newton got 95.2% marks.
Press any key to continue
```

Look at the constructor calls as shown in the output. First, the parameter less constructor of base-class (*Student*) is called and then the parameterized constructor of sub-class (*SchoolStudent*) class is called. It shows that the compiler does create an object of the base class before it instantiates the sub-class.

Now, let us comment the parameterless constructor of the *Student* class

```
Class Student
        ...
' Student Constructor
'Public Sub New()
'    Console.WriteLine("New student created. Parameter less constructor called...")
```

```
'End Sub
...
End Class
```

Now, when we try to compile the program, we get the error

First statement of this 'Sub New' must be a call to 'MyBase.New' or 'MyClass.New'
because base class 'VBDotNetSchoolLesson4.Student' of
'VBDotNetSchoolLesson4.SchoolStudent' does not
have an accessible 'Sub New' that can be called with no arguments.

The compiler is unable to find the zero argument (parameter less) constructor of the base-class. What should be done here?

Keyword MyBase - Calling Constructors of a base-class explicitly
We can explicitly call the constructor of a base-class using the keyword *MyBase*. *MyBase.New* must be the first line of constructor body like

```
Class SubClass
Inherits BaseClass
Public Sub New(ByVal id As Integer)
MyBase.New()' explicit base class constructor call
' some code goes here
End Sub
End Class
```

In the code above, the parameterized constructor of sub-class (*SubClass*) is explicitly calling the parameter less constructor of base class. We can also call the parameterized constructor of base-class through the *MyBase* keyword, like

```
Class SubClass
Inherits BaseClass
Public Sub New(ByVal id As Integer)
MyBase.New(id)' explicit base class constructor call
' some code goes here
End Sub
End Class
```

The constructor of *SubClass-> New(int)* will explicitly call the constructor of the base-class (*BaseClass*) that takes an *Integer* argument.

Lets practice with our *SchoolStudent* class again. Revert the changes we made to its second constructor configuration and make it looks like below

```
Public Sub New(ByVal pRegNum As Integer, ByVal pName As String, ByVal pDob As
DateTime,
               ByVal pTotalMarks As Integer, ByVal pTotalObtainedMarks As Integer)
' call to base class constructor
MyBase.New(pRegNum, pName, pDob)
mTotalMarks = pTotalMarks
mTotalObtainedMarks = pTotalObtainedMarks
Console.WriteLine("New school student is created. Parameterized constructor
called...")
```

```
End Sub
```

The constructor above calls the parameterized constructor of the base-class (*Student*), delegating the initialization of inherited fields to the base-class's parameterized constructor.

```
Public Sub New(ByVal pRegistrationNumber As Integer, ByVal pName As String, _
ByVal pDateOfBirth As DateTime)
mRegistrationNumber = pRegistrationNumber
mName = pName
mDateOfBirth = pDateOfBirth
Console.WriteLine("New Student Created. Parameterized constructor called...")
End Sub
```

This is also important as the field *mRegistrationNumber* is Private and provides the *Get* public property only. So, in no way, we can assign *mRegistrationNumber* of a student except for calling a constructor. *MyBase.New* best fits this scenario.

The *Main()* method would be like

```
Public Sub Main()
Dim schStd As New SchoolStudent(2, "Newton",New DateTime(1983, 4, 1), 500,476)
Console.WriteLine("Age of student, {0}, is {1}. {0} got {2}% marks.",
                  schStd.Name, schStd.GetAge(), schStd.GetPercentage())
End Sub
```

When we run this program, following output is displayed on console

```
New Student Created. Parameterized constructor called...
New school student is created. Parameterized constructor called...
Age of student, Newton, is 20. Newton got 95.2% marks.
Press any key to continue
```

You can see from the output above that first the parameterized constructor of the *Student* (base-class) is called and then the parameterized constructor of the *SchoolStudent* (sub-class) is called and sets the appropriate values. This is exactly what we wanted to do. It then simply prints the name, age and percentage of *SchoolStudent*.

Note that although we write *MyBase* after the sub-class's constructor declaration, the constructor of the base-class is always called before the constructor of the sub-class.

Author's Note: VB.Net's MyBase keyword is similar to C#'s base and Java's super keyword. However C# has adopted the syntax of MyBase from Java's super constructor calls.

Protected Access Modifier

Within the Student class, *mRegistrationNumber* is made Private with only the *Get* property set, so that no user of the *Student* class can modify the *mRegistrationNumber* from outside the class. It may be possible that we need to modify this field (*mRegistrationNumber*) in the *SchoolStudent* class but we can not do this. VB.Net provides the *Protected* access modifier just for this. *Protected* members of a class can be accessed either inside the containing class or inside its sub-class. Users still won't be able to call the *Protected* members through object references and if one tries to do so, the compiler will complain and generate an error.

Suppose we have a class A with a protected method *DoWork()*

```
Class A
    Protected Sub DoWork()
        Console.WriteLine("DoWork called...")
    End Sub
End Class
```

If we try to call *DoWork* in the *Main()* method of the *Test* module, the compiler will generate an error

```
        Module Test
                Sub Main()
                        Dim aObj As new A()
                        aObj.DoWork()
                End Sub
        End Module
```

When we try to compile it, the compiler says

'VBDotNetSchoolLesson4.A.Protected Sub DoWork()' is not accessible in this context because it is 'Protected'.

But, if we declare another class *B* which inherits *A*. This class can call the *DoWork()* method inside its body

```
Class B
    Inherits A
    Public Sub New()
        DoWork()
    End Sub
End Class
```

here, we inherited the class *B* from *A* and called *DoWork()* of its base-class (*A*) in its constructor. Now, when we write

```
  Public Sub Main()
        Dim bObj As New B()
  End Sub
```

It will result as

```
DoWork called...
Press any key to continue
```

This shows that we can access the *Protected* members of a class inside its sub-classes. Note that it is still an error to try and compile the following

```
        Sub Main()
                Dim bObj As new B()
                bObj.DoWork()    ' error
        End Sub
```

We can not access *Protected* members, even with the reference of the sub-class.

Protected Friend Access Modifier
In a similar way, the *Protected Friend* access modifier allows a member (field, property and method) to be accessed

- Inside the containing class, or
- Inside the same project, or
- Inside the sub-class of containing class.

Hence, the *Protected Friend* acts like '*Protected OR Friend*', i.e., either protected or friend.

The NotInheritable class
Finally, if you don't want your class to be inherited by any classes, you can mark it with the *NotInheritable* keyword. No class can inherit from a *NotInheritable* class.

```
NotInheritable Class A
        ...
End Class
```

If one tries to inherit another class B with class A

```
Class B
    Inherits A
        ...
End Class
```

The compiler will generate following error:

```
'B' cannot inherit from class 'A' because 'A' is declared 'NotInheritable'.
```

Author's Note: VB.Net's NotInheritable keyword is identical to C#'s sealed and Java's final keyword when applied to classes.

The Object class - ultimate base of all classes
In VB.NET (and the .NET framework) all the types (classes, structures and interfaces) are implicitly inherited from the Object class defined in the System namespace. This class provides the low-level services and general functionality to each and every class in the .NET framework. The Object class is extremely useful when it comes to polymorphism (which we

are about to see) as the reference of type *Object* can hold any type of object. The *Object* class has following methods

Method Name	Description
Equals(Object)	Compare two objects for equality. The default implementation only supports reference equality, that is, will return true if both references point to the same object. For value types, bitwise checking is performed. Derived classes should override this method to define equality for their objects.
Shared Equals(Object, Object)	Same as above except this method is shared.
GetHashCode()	Returns the hash code for the current object.
GetType()	Returns the Type object that represents the exact run-time type of the current object
Shared ReferenceEquals (Object,Object)	Returns true if both the references passed points to the same object otherwise returns false.
ToString()	Returns the string representation of the object. Derived classes should override this method to provide the string representation of the current object.
Protected Finalize()	This protected method should be overridden by the derived classes to free any resources. This method is called by CLR before the current object is reclaimed by Garbage Collector.
Protected MemberwiseClone()	Provides the shallow copy of current object. The shallow copy contains the copy of all the instance fields (state) of the current objects.

VB.Net also provides an *Object* keyword which maps to this *System.Object* class in the .NET framework class library (FCL)

Polymorphism
It is said that there are three stages when learning an object oriented programming language 1.The first phase is when a programmer uses non-object oriented constructs (like *For...Next, If...Then...Else, Select...Case*) of an object oriented programming language.
2.The second phase is when a programmer writes classes, inherits them and make their objects…
3.The third phase is when a programmer uses polymorphism to achieve late binding.

MSDN (Microsoft Developer Network) explanation,
"Polymorphism is the ability for classes to provide different implementations of methods that are called by the same name. Polymorphism allows a method of a class to be called without regard to what specific implementation it provides."

Using the reference of base types for referencing the objects of child types
Before getting into the details of polymorphism, we need to understand that the reference of a base type can hold the object of a derived type. Class A can be inherited by a class B.

```
Class A
    Public Sub MethodA()
        Console.WriteLine("MethodA called...")
    End Sub
End Class
Class B
```

```
    Inherits A
    Public Sub MethodB()
        Console.WriteLine("MethodB called...")
    End Sub
End Class
```

then it is legal to write

```
        Dim aObj As A   ' type of reference aObj is A
        aObj = New B()  ' aObj refers an Object of B
```

Here a reference of type A. It's holding the object of type B. In this case we are treating the object of type B as an object of type A (which is quite possible as B is a sub-type of A). Now, it is possible to write

```
        aObj.MethodA()
```

but it is in-correct to write

```
        a.MethodB()               ' error
```

Although we have the object of type B (contained in the reference of type A), we can not access any members of type B as the apparent type of the reference is A and not B.

Using the methods of same name in the Base and the Sub-class In our *Shape* class, we can define a method named *Draw()* that draws the shape on the screen

```
Class Shape
    Public Sub Draw()
        Console.WriteLine("Drawing Shape...")
    End Sub
End Class
```

We inherit another class *Circle* from the *Shape* class, which also contains a method called *Draw()*

```
Class Circle
    Inherits Shape
    Public Sub Draw()
        Console.WriteLine("Drawing Circle...")
    End Sub
End Class
```

Here, we have the *Draw()* method with the same signature in both the *Shape* and the *Circle* classes. If we place the following in the *Main()* method

```
    Public Sub Main()
        Dim theCircle As New Circle()
        theCircle.Draw()
    End Sub
```

The Circle's *Draw()* method is called and the program will (expectedly) display

Drawing Circle...

Note that the compiler will give a warning that *Draw()* in *Circle* conflicts the inherited *Draw()* method of the *Shape* class. If we write in our Main() method

```
        Dim theShape As Shape
        theShape = New Circle()
        theShape.Draw()
```

Shape's *Draw()* method is called and the program will display

Drawing Shape...

Overriding the methods - The Overridable and the Overrides keywords
If we want to override the *Draw()* method of the *Shape* class in the *Circle* class, we have to mark the *Draw()* method in the *Shape* (base) class as *Overridable* and the *Draw()* method in the *Circle* (sub) class as *Overrides*.

```
Class Shape
    Public Overridable Sub Draw()
```

```
        Console.WriteLine("Drawing Shape...")
    End Sub
End Class

Class Circle
    Inherits Shape
    Public Overrides Sub Draw()
        Console.WriteLine("Drawing Circle...")
    End Sub
End Class
```

Now, if we write in our *Main()* method

```
    Public Sub Main()
        Dim theShape As Shape
        theShape = New Circle()
        theShape.Draw()
    End Sub
```

We used the reference of the base type (*Shape*) to refer an object of the sub type (*Circle*) and called the *Draw()* method through it.

As we have overridden the *Draw()* method of *Shape* in the *Circle* class and the *Draw()* method is marked Overridable in Shape. The compiler will no longer see the apparent (or reference) type to call the method (static, early or compile time object binding). Rather, it will apply "dynamic, late or runtime object binding" and will see the object type at 'runtime' to decide which *Draw()* method, it should call. This procedure is called polymorphism, where we have different implementations of a method with the same name and signature in the base and sub-classes. When such a method is called using a base-type reference, the Common Language Runtime (CLR) uses the actual object type referenced by the base type reference to decide which of the methods to call. When we compile and execute the above program, it will result are as

```
Drawing Circle...
```

Although, we called the *Draw()* method using the reference of the *Shape* type, the CLR will consider the object held by the *theShape* reference and called the *Draw()* method of *Circle* class.

We mark the method in the base class which we want to achieve polymorphism as *Overridable*. By marking a method as *Overridable* we allow a method to be overridden and be used polymorphically. In the same way we mark the method in a sub-class as *Overrides* when it is overriding the virtual method in the base class.

By marking the method as *Overrides*, we show that we are deliberately overriding the corresponding *Overridable* method in the base class.

Author's Note: All the methods in VB.NET are non-virtual by default like C# and C++ and unlike Java (where all the methods are implicitly virtual). We have to explicitly mark a method as Overridable (like C# and C++).

Unlike C++ and Java, we also have to declare an overriding method in a sub-class as Overrides in order to avoid any unconscious overriding. Infact,

```
VB.NET also introduces the Shadows keyword, to mark the
method as non-overriding.
```

You might be wondering why in the previous example (of *Shape* and *Circle* class), late binding is necessary as the compiler can decide from previous lines which object the reference (*theShape*) holds

```
        Dim theShape As Shape
        theShape = New Circle()
        theShape.Draw()
```

Yes, it is possible for the compiler to conclude it in this particular case. However, usually the compiler is not able to decide the object of which class the reference would be referencing at run-time. Suppose we define two more classes: *Rectangle* and *Curve* which also inherits the *Shape* class and override its *Draw()* method to give their own specific implementation.

```
Class Rectangle
    Inherits Shape
    Public Overrides Sub Draw()
        Console.WriteLine("Drawing Rectangle...")
    End Sub
End Class
Class Curve
    Inherits Shape
    Public Overrides Sub Draw()
        Console.WriteLine("Drawing Curve...")
    End Sub
End Class
```

Now, if we write the *Main()* method as follows

```
    Public Sub Main()
        Dim shapes() As Shape
        shapes = New Shape() {New Circle(), New Rectangle(), New Curve()}
        Dim rand As New Random()
        Dim i As Integer
        For i = 1 To 5
            Dim randNum As Integer = rand.Next(0, 3)
            shapes(randNum).Draw()
        Next
    End Sub
```

We made an array of type *Shape* and stored an object of *Circle*, *Rectangle* and *Curve* classes in it. Later, we used the Random class to generate a random number between 0 and 2 (both 0 and 2 inclusive). We used this randomly generated number as an index in the *shapes* array to call the *Draw()* method. Neither we nor the compiler is sure which particular index of *shapes* will be used and the *Draw()* method of which sub-class of *Shape* will be called at runtime in each iteration of the loop. When we compile and run the above program, we will see different outputs with each run. For example, when I executed the above code, I got the following output

```
Drawing Curve...
Drawing Rectangle...
```

```
Drawing Curve...
Drawing Circle...
Drawing Rectangle...
Press any key to continue
```

The Shadows keyword

Suppose, in our *Circle* class, we don't want to override the *Draw()* method yet we need to have a *Draw()* method, what we can do? In java, we can't do this. In C++, if we have marked the method in base class as *virtual*, it is also impossible. But, VB.Net introduces the keyword *Shadows* to mark a method as a non-overriding method and as one which we don't want to use polymorphically. We have the *Shape* and *Circle* classes as below

```
Class Shape
    Public Overridable Sub Draw()
        Console.WriteLine("Drawing Shape...")
    End Sub
End Class

Class Circle
    Inherits Shape
    Public Shadows Sub Draw()
        Console.WriteLine("Drawing Circle...")
    End Sub
End Class
```

Note, we marked the *Draw()* method in *Circle* with the *Shadows* keyword to avoid polymorphism. If we write our *Main()* method

```
    Public Sub Main()
        Dim theShape As Shape = New Circle()
        theShape.Draw()
    End Sub
```

When we compile and run the code above, we will see the following output.

```
Drawing Shape...
```

Since we marked the *Draw()* method in the *Circle* class with *Shadows*, no polymorphism is applied here and the *Draw()* method of the *Shape* class is called. If we don't mark the *Draw()* method with *Shadows* keyword, we will see the following warning at compile time

```
sub 'Draw' shadows an overridable method in a base class. To override the base method,
this method must be declared 'Overrides'.
```

Type casting the objects - Up-casting and Down-casting
Type Casting means 'making an object behave like' or 'changing the apparent type of object'. In VB.Net, you can cast objects in inheritance hierarchy either from bottom to top (up-casting) or from top to bottom (down-casting).

Up-casting is simple, safe and implicit as we have seen that the reference of the parent type can reference the object of a child type.

```
Dim theParent As Parent = New Child()
```

On the contrary, down-casting is un-safe and explicit (By un-safe, we mean it may throw exception). Review the following line of code

```
Dim shapes() As Shape
shapes = New Shape() {New Circle(), New Rectangle(), New Curve()}
```

Where *Circle, Rectangle* and *Curve* are sub-classes of the *Shape* class. If we want to reference the *Rectangle* object in the *shapes* array with the reference of type *Rectangle*, we can't just write

```
Rectangle rect = shapes(1)
```

Instead, we have to explicitly apply the cast here using the built-in CType() function as

```
Dim rect As Rectangle = CType(shapes(1), Rectangle)
```

Since the cast is changing, the apparent type of the object from parent to child (downward direction in inheritance hierarchy); it is called down-casting.

Note that down-casting can be un-successful and if we attempt to code

```
Dim rect As Rectangle = CType(shapes(2), Rectangle)
```

Since *shapes(2)* contains an object of type *Curve* which can not be cast to the *Rectangle* type. Hence the CLR would raise the following exception at run-time

System.InvalidCastException: Specified cast is not valid.

Checking for the validity of casting - The TypeOf...is a keyword
To check the run-time type of object, you can use the *TypeOf...is* keyword. The *TypeOf...is* compares the type of object with the given type and returns *True* if they are cast-able. Otherwise it returns *False*. For example,

```
Console.WriteLine(TypeOf shapes(1) Is Rectangle)
```

would print *True* on the Console Window, while

```
Console.WriteLine(TypeOf shapes(2) Is Rectangle)
```

would print *False* on the Console window. We might use the *TypeOf...is* operator to check the runtime type of an object before applying down-casting.

```
Dim shapes() As Shape
shapes = New Shape() {New Circle(), New Rectangle(), New Curve()}
Dim rect As Rectangle = Nothing
If TypeOf shapes(1) Is Rectangle Then
   rect = CType(shapes(1), Rectangle)
End If
```

Boxing and Un-boxing

The last topic for this session is boxing and un-boxing. Boxing allows value types to be implicitly treated like objects. Suppose we have an integer variable i as

```
Dim i As Integer = 5
```

when we write

```
i.ToString()
```

The Compiler implicitly creates an instance of the *Object* class and boxes (store) a copy of this *Integer* value (5) in this object. It then calls the *ToString()* method on that instance of the *Object* class which has boxed the copy of the *Integer* value. It is similar to coding

```
Dim i As Integer = 5
Dim obj As Object = i   ' implicit boxing
obj.ToString()
```

You can see in the above code that boxing is implicitly applied in VB.NET and you don't have to write

```
Dim obj As Object = CType(i, Object) ' un-necessary explicit boxing
```

On the other hand, un-boxing is an explicit conversion from object type to value type. The following lines shows how un-boxing is implemented in VB.NET

```
Dim i As Integer = 5
Dim obj As Object = i    ' implicit boxing
Dim j As Integer = CType(obj, Integer)   ' explicit un-boxing
```

Like down-casting, un-boxing can be un-safe and through *InvalidCastException* at runtime.

Author's Note: Although boxing and un-boxing look very similar to up-casting and down-casting, there are some points that differentiate the two.

- Boxing and Un-boxing is the transformation between value type and object type while casting just transforms the apparent (reference) type of objects.
- Value types are stored at the stack and objects are stored at the heap.

Boxing takes a copy of value types from the stack to the heap while un-boxing takes value types back to the stack. On the other hand, casting does not physically move or operate on an object. Casting merely changes the way objects are treated in a program by altering their reference type.

Food for thought: Exercise 5

1. How are inheritance and polymorphism in VB.NET different from that in Java and C++?
2. How can we make

a) a variable so that its value can not be changed once assigned at declaration time?
b) a reference to object so that it can not reference any other object, except for the one it is made to reference at declaration time?
c) a method so that it can not be overridden by a sub-class?
d) a class that can not be inherited by another class?

3. Can we override properties and apply polymorphism on these?
4. We saw that constructors are called in the order from top to bottom (parent to child class) in inheritance hierarchy. In which order are the Finalize() called during inheritance?
5. What are the advantages of using Protected access modifiers in inheritance?
6. How is type casting different from boxing/un-boxing?
7. When I compiled and run the following code

```
Module Test
    Public Sub Main()
        Dim theParent As Parent = New Child()
        Console.WriteLine("i = {0}", theParent.i)
        theParent.MyMethod()
    End Sub
End Module

Class Parent
    Public i As Integer = 5
    Public Overridable Sub MyMethod()
        Console.WriteLine("I am Parent's MyMethod()")
    End Sub
End Class
Class Child
    Inherits Parent
    Public i As Integer = 7
    Public Overrides Sub MyMethod()
        Console.WriteLine("I am Child's MyMethod()")
    End Sub
End Class
```

It outputs as

```
i = 5
I am Child's MyMethod()
Press any key to continue
```

As we called both i and *MyMethod()* using the *Parent* type reference containing the *Child* type object. Why is i of the *Parent* and the *MyMethod()* of *Child* called?

Solution of Last Issue's Exercise (Exercise 4)

1. What is the difference between a class and an object? Why do we make classes?

A class is an abstract model which defines a new data type in a programming language, while an object is the concrete realization or instance of a class. There may be any number of objects of a particular type of class, all having the same behavior and properties defined by the class. We can make classes to introduce new abstract data types in a programming language so that we can use re-use them as many times as we need in development.

2. How are fields and properties different from each other?

Fields are the data encapsulated by a class while properties are the controlled or valid accessors to the fields for which the user who wish to access the fields of the class.

3. Isn't it better to make a field *Public* than providing its property with both *Set...End Set* and *Get...End Get* blocks? After all, the property will allow the user to both read and modify the field, so why not use a *Public* field instead? Motivate your answer

Not always! Properties are not just to provide access to fields; rather, they are supposed to provide controlled access to fields of the class. As the state of the class depends upon the values of its fields. Using properties we can assure that no invalid (or unacceptable) value is assigned to the fields.

4. Write a class to simulate a Circle Shape. Each circle has its radius and center point

- (a) Provide 2 constructors: a parameter less and one which takes the center point and radius as parameters.
- (b) Provide a mechanism to find the diameter (twice of radius) and the area (PI * square of radius).
- (c) Provide a mechanism to move a circle from its starting location to somewhere else.
- (d) Apply correct access modifiers and use proper fields, method and properties

```
Module Test
    Public Sub Main()
        Dim theCircle As New Circle(20, 30, 5)
        Console.WriteLine(theCircle.ToString())
        Console.WriteLine()
        Console.WriteLine("Diameter of Circle is " +
                          theCircle.GetDiameter().ToString())
        Console.WriteLine("Area of Circle is " +
                          theCircle.GetArea().ToString())
        Console.WriteLine()
        Console.WriteLine("Moving center of circle to (50, 50)")
        theCircle.Center = New Point(50, 50)
        Console.WriteLine(theCircle.ToString())
    End Sub
End Module

Class Circle
```

```
    ' private Fields
    Private mCenter As Point
    Private mRadius As Integer
    ' public Constructors
    Public Sub New()
    End Sub
    Public Sub New(ByVal pCenter As Point, ByVal pRadius As Integer)
        mCenter = pCenter
        mRadius = pRadius
    End Sub
    Public Sub New(ByVal centerX As Double, ByVal centerY As Double, _
 ByVal pRadius As Integer)
        mCenter = New Point(centerX, centerY)
        mRadius = pRadius
    End Sub
    'public Properties
    ' move center point using this property
    Public Property Center() As Point
        Get
            Return mCenter
        End Get
        Set(ByVal Value As Point)
            mCenter = Value
        End Set
    End Property
    Public Property Radius() As Integer
        Get
            Return mRadius
        End Get
        Set(ByVal Value As Integer)
            mRadius = Value
        End Set
    End Property
    ' public Functions
    Public Function GetDiameter() As Integer
        Return mRadius * 2
    End Function
    Public Function GetArea() As Double
        Return Math.PI * Math.Pow(mRadius, 2)
    End Function
    Public Overrides Function ToString() As String
        Return "A circle centered at (" + Center.x.ToString() + ", " + _
Center.y.ToString() + ") with radius = " + mRadius.ToString() + " units."
    End Function
End Class
' class to represent a Point
Class Point
    ' public fields
    Public x As Double
    Public y As Double
    ' public constructors
    Public Sub New()
    End Sub
    Public Sub New(ByVal x As Double, ByVal y As Double)
        Me.x = x
        Me.y = y
    End Sub
End Class
```

5. Write a program containing a method that will swap (interchange) the value of

two integer type variables.

```
Module Test
    Public Sub Main()
        Dim i As Integer = 5
        Dim j As Integer = 7
        Console.WriteLine("Before swapping...")
        Console.WriteLine("i = {0}, j = {1}" & vbCrLf, i, j)
        swap(i, j)
        Console.WriteLine("After swapping...")
        Console.WriteLine("i = {0}, j = {1}" & vbCrLf, i, j)
    End Sub
    Public Sub swap(ByRef num1 As Integer, ByRef num2 As Integer)
        Dim temp As Integer = num1
        num1 = num2
        num2 = temp
    End Sub
End Module
```

6. Can we use *Me* in a *Shared* context? Why or Why not? Can *Me* ever be equal to *Nothing*?

No! As Shared members are not accessed with reference to any object but with reference to the class, we can not use Me in the Shared context. Me can never be equal to Nothing as Me always belong to the object using it. Me can not exist without an object hence Me can never be equal to Nothing

What's Next...

Next time, we will be discussing how to use structures and enumerations (enum) in VB.Net (something missing in Java programming language). We will also spend some time discussing Garbage Collector and Nested Classes. We will see

- Structures and Enumerations
- What are structures?
- Advantages, limitations, and specifications of structures
- Implementing/instantiating structures with and without new operator
- What are Enumerations? Their Advantages?
- Defining and using Enumerations
- Passing Enumeration as parameters to methods
- Garbage Collector - How it works?
- Using System.GC class.
- Nested classes in VB.Net

Structure, Enumeration, Garbage Collection and Nested Classes

Lesson Plan

This lesson consists of four major topics: Structures, Enumeration, Garbage Collector and Nested Classes. We will cover these one by one.

Structures

Structures are denoted in VB.Net by the *Structure* keyword. They can be thought of as light-weight objects. Structures are very similar to classes in VB.Net but with the following properties

- Structure are useful for creating lightweight types that are used to hold data such as Point, Rectangle and Color types.
- Structure are of value type, contrary to classes which are of reference type. This means that structures are allocated on the stack and are passed to the methods by value.
- Structure may contain constructors (except for a no-argument constructor), fields, methods and properties just like classes.
- Like all value types, Structures can neither inherit other class nor can they be inherited.
- A Structure can implement interfaces.
- Like every other type in VB.Net, Structures are also implicitly inherited by the System.Object class.
- Instances of a Structure can be created with or without using the New keyword.
- Most of the .Net framework types such as System.Int32 (for an Integer), System.Double (for a Double), System.Boolean (for a Boolean), System.Byte (for a Byte),… are implemented as a Structure.
- When kept to a small size, a Structure can be more efficiently used by the system than a class.

Defining Structure

A Structure is defined just like a *Class* , by use of the *Structure* keyword. Suppose in a drawing application, we need a *Point* data type. Since this is a simple and light weight type; we could implement it as a *Structure*

```
Structure Point
    Public x As Integer
    Public y As Integer
    Public Sub New(ByVal x As Integer, ByVal y As Integer)
        Me.x = x
        Me.y = y
    End Sub
    Public Overrides Function ToString() As String
        Return "(" + x.ToString() + ", " + y.ToString() + ")"
    End Function
End Structure
```

Above, we declared a *Structure* named *Point*. *Point* contains two public fields (x and y) that represent the location of *Point* in a coordinate system. We provided a *Public* constructor to initialize the location of the point and we also overrided the *ToString()* method from the

Object class, so that our point can be printed easily using the *Console.WriteLine()* method from within the *Main()* method.

Instantiating the Structure
A Structure can be instantiated in three ways

- Using the New keyword and calling the default no-argument constructor
- Using the New keyword and calling a custom or user defined constructor
- Without using the New keyword

As mentioned earlier, providing a no-argument constructor in a Structure will generate a compilation error. The compiler implicitly provides a default no-argument constructor for each *Structure* which initializes the fields of a *Structure* with their default values. In the following *Main()* method, we instantiated *Point* using the above mentioned three ways

```
    Public Sub Main()
        Dim pt As New Point()
        Dim pt1 As New Point(15, 20)
        Dim pt2 As Point ' instantiation without new keyword
        pt2.x = 6
        pt2.y = 3
        Console.WriteLine("pt = {0}", pt)
        Console.WriteLine("pt1 = {0}", pt1)
        Console.WriteLine("pt2 = {0}", pt2)
    End Sub
```

The output of this program is

```
pt = (0, 0)
pt1 = (15, 20)
pt2 = (6, 3)
Press any key to continue
```

We have instantiated three *Point* objects. The first one (referenced by pt) has been instantiated using *New* and a default no-argument constructor (implemented by the compiler) which zeroed all the fields thus the first Point (pt) printed (0, 0). The second one (referenced by pt1) is instantiated using the *New* keyword and a custom (*Integer, Integer*) constructor. The point (pt1) is initialized with the specified value and thus printed (15, 20) on the console. The third object (referenced by pt2) was created without using the *New* keyword (like implicit data types). Note that we first initialized all the fields of pt2 before using it (in the *Console.WriteLine()* method). Before using a Structure created without the *New* keyword, all of its fields should be explicitly initialized, otherwise they will take their default values. Hence, the Point (pt2) printed out as (6, 3). Note, that we coded

```
        Console.WriteLine("pt = {0}", pt)
```

instead of

```
Console.WriteLine("pt = {0}", pt.ToString())
```

Although, *Console.WriteLine()* expects a string, but since we overrided the *ToString()*

method in the *Point* Structure, the compiler will implicitly call the *ToString()* method when it expects a string in the *Console.WriteLine()* method.

Let's play with our program to increase our understanding of a Structure. If we don't initialize any of the fields of the Point as so

```
    Public Sub Main()
        Dim pt2 As Point
        'pt2.x = 6
        pt2.y = 3
        Console.WriteLine("pt2 = {0}", pt2)
    End Sub
```

The compiler will use the default value of the field x, which is zero. Hence the program will output

```
pt2 = (0, 3)
Press any key to continue
```

Let's now make the fields of the point *Private* and provide *Public* properties to access them as Structure Point

```
    Public mX As Integer
    Public mY As Integer
    Public Sub New(ByVal pX As Integer, ByVal pY As Integer)
        mX = pX
        mY = pY
    End Sub
    Public Property X() As Integer
        Get
            Return mX
        End Get
        Set(ByVal Value As Integer)
            mX = Value
        End Set
    End Property
    Public Property Y() As Integer
        Get
            Return mY
        End Get
        Set(ByVal Value As Integer)
            mY = Value
        End Set
    End Property
    Public Overrides Function ToString() As String
        Return "(" + mX.ToString() + ", " + mY.ToString() + ")"
    End Function
End Structure
```

Let's try to create an instance of *Point* without using the *New* keyword

```
    Public Sub Main()
        Dim pt2 As Point
        pt2.X = 6
        pt2.Y = 3
        Console.WriteLine("pt2 = {0}", pt2)
```

```
    End Sub
```

We created an instance of Point (pt2) and initialized its fields through their accessor properties and then attempt to use it in a *Console.WriteLine()* method. When we compile and execute the program,the following results are displayed

pt2 = (6, 3)
Press any key to continue

Finally let's try to define a no-argument constructor in a Structure

```
Structure Point
    Public mX As Integer
    Public mY As Integer
    Public Sub New()
        mX = 3
        mY = 4
    End Sub
    Public Sub New(ByVal pX As Integer, ByVal pY As Integer)
        mX = pX
        mY = pY
    End Sub
End Structure
```

When we try to compile the above program, the compiler remarks

Structures cannot declare a non-shared 'Sub New' with no parameters.

Here, the compiler clearly announced it illegal to define a parameter-less (no-argument) constructor within a Structure.

Author's Note: A Structure is not a new concept. In C++, a Structure is another way of defining a class (why?). Although most of the time, C++ developers do use Structures for light weight objects holding just data; but the C++ compiler does not impose this. VB.Net Structures are more restricted than classes because they are value types and can not take part in inheritance. Java does not provide Structure at all.

Enumeration
An enumeration is a very good concept originally found in C++ but is not present in Java. VB.Net supports enumerations using the Enum keyword. Enumeration, like classes and structures, also allow us to define new types. Enumeration types contain a list of named constants. The list is called an enumerator list while its contents are called enumerator identifiers. Enumerations are of integral types, like Integer, Short and Byte,… (except the Char type).

The need for Enumeration
Before we go into details of how to define and use enumerations, let's consider first why we do need enumeration in the first place? Suppose we are developing a Windows Explorer type of program that allows its users to do different tasks with the underlying file system. In "View properties", we allow our user to see the sorted list of files/folders on the basis of name, type, size or modification date. Let us write a function that takes a string whose value represents one of these four criteria.

```
    Public Sub Sort(ByVal criteria As String)
        Select Case criteria
            Case "by name"
                ' code to sort by name
            Case "by type"
                ' code to sort by type
            Case "by size"
                ' code to sort by size
            Case "by date"
                ' code to sort by modification date
            Case Else
                Throw New Exception("Invalid criteria for sorting passed...")
        End Select
```

If the user selects the option to sort files/folders by name, we pass this function a string "by name" and like this.

```
        Dim explorer As new Explorer()
        ' some code goes here
        explorer.Sort("by name")
```

Explorer is the name of the class containing the *Sort()* method. As we pass the string value "by name" to the *Sort()* method, it compares it inside this *Select...Case* block and will take the appropriate action (sort items by name). However, what if someone writes in the following code

```
        explorer.Sort("by extention")
```

or,

```
        explorer.Sort("irrelevant text")
```

The program will still get compiled but it will throw an exception at runtime which if not

caught properly might crash the program (We will see more about exceptions in coming lessons. Right now, just consider that this program will not execute properly and might crash if some irrelevant value is passed as a parameter to the *Sort()* method). There should be some method to check the value at compile-time in order to avoid a run-time collapse. Enumerations provides the solution for this type of problem!

Using Enumeration (enum)

Enumerations are defined in VB.Net using the *Enum* keyword. The enumerator identifiers (named constants) are separated with each other using a new line.

```
Enum SortCriteria
    ByName
    ByType
    BySize
    ByDate
End Enum
```

We have defined an enumeration named *SortCriteria* which contains four constants: *ByName, ByType, BySize* and *ByDate*. We can modify our *Sort()* method to accept a *SortCriteria* type enumeration only as below

```
Class Explorer
    Public Sub Sort(ByVal criteria As SortCriteria)
        Select Case criteria
            Case SortCriteria.ByName
                ' code to sort by name
                Console.WriteLine("Files/Folders sorted by name")
            Case SortCriteria.ByType
                ' code to sort by type
                Console.WriteLine("Files/Folders sorted by type")
            Case SortCriteria.BySize
                ' code to sort by size
                Console.WriteLine("Files/Folders sorted by size")
            Case SortCriteria.ByDate
                ' code to sort by modification date
                Console.WriteLine("Files/Folders sorted by modification date")
        End Select
    End Sub
End Class
```

Note that we did not have an *Case Else* clause in the *Select...Case* block, because it is impossible to pass un-allowed criteria either by mistake or intentionally as the compiler will check the criteria at compile time. OK, in order to call the *Sort()* method, we need to pass an instance of the enumeration of type *SortCriteria* which can only take on form as the allowed four criterion. Hence, providing compile time checks for illegal criteria.

We now call the *Sort()* method as so

```
Dim theExplorer As new Explorer()
theExplorer.Sort(SortCriteria.BySize)
```

or alternatively, we can create an instance of the *SortCriteria* and then pass it as an argument to the *Sort()* as

```
Dim theExplorer As new Explorer()
Dim criteria As SortCriteria = SortCriteria.BySize
theExplorer.Sort(criteria)
```

When we compile and run the above program, we see the following output

```
Files/Folders sorted by size
Press any key to continue
```

In a similar manner, you can define many useful enumerations such as days of the week (Sunday, Monday,…), names of months (January, February,…), connection types (TCPIP, UDP), file opening modes (ReadOnly, WriteOnly, ReadWrite, Append), and many more.

More about Enumerations

Enumerations internally use integral values to represent the differently named constants. The underlying type of an enumeration can be any integral type (except for the Char type). The default type is *Integer*. In fact, when we declare an enumeration, its items are assigned successive integer values starting from zero. Hence, in our *SortCriteria* enumeration, the value of *ByName* is 0, the value of *ByType* is 1, the value of *BySize* is 2 and the value of *ByDate* is 3. We can convert the Enumeration data back to the integral value

```
Public Sub Main()
   Dim criteria As SortCriteria = SortCriteria.BySize
   Dim i As Integer = criteria
   Console.WriteLine("the integral value of {0} is {1}", criteria, i)
End Sub
```

We created an instance of *SortCriteria* (named criteria), and assigned it to an Integer variable and then printed it using the *Console.WriteLine()* method. The program outputs as

```
the integral value of BySize is 2
Press any key to continue
```

You can see from the output that the integral value of *SortCriteria.BySize* is 2. Also note that when we printed the enumeration identifier (*criteria*), it printed the named constant (*BySize*) and not its value (1). We can also define the values for the enumerator identifier explicitly while defining a new enumerations like

```
Enum Temperature
    BoilingPoint = 100
    FreezingPoint = 0
End Enum
```

When we try and print the value of either *Temperature.BolingPoint* or *Temperature.FreezingPoint*, it will display the assigned values and not the default values (starting from zero and incrementing by one each time).

```
Dim i As Integer = Temperature.BoilingPoint
Console.WriteLine("the integral value of {0} is {1}", Temperature.BoilingPoint, i)
```

It will display

```
the integral value of BoilingPoint is 100
Press any key to continue
```

Consider the following enumeration

```
Enum Days As Byte
    Monday = 1
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
    Sunday
End Enum
```

We have defined an enumeration for Days. Note that we assigned the value of 1 to Monday. Hence, the integral values for successive constants would be successive integers after 1 (that is, 2 for Tuesday, 3 for Wednesday and so on). Also, note that we specified the underlying type for the enumeration (Days) as Byte instead of the default Integer type. The compiler will use byte type to represent Days internally.

```
Enum Days As Byte
        ...
End Enum
```

In the *Main()* method, we can print the integral value of Days as

```
Public Sub Main()
   Dim b As Byte = Days.Tuesday
   Console.WriteLine("the integral value of {0} is {1}", Days.Tuesday, b)
End Sub
```

that outputs as

```
the integral value of Tuesday is 2
Press any key to continue
```

Finally, enumerations are value types. They are created and stored on the stack and passed to methods by making a copy of the value (by value).

Garbage Collection in .Net

Memory management is no longer the responsibility of the programmer when writing managed code in the .Net environment. .Net provides its own Garbage Collector to manage memory. The Garbage Collector is a program that runs in the background as a low-priority thread and keeps track of un-referenced objects (objects that are no more referenced by any reference) by marking them as 'dirty objects'. The Garbage collector is invoked by the .Net Runtime at regular intervals and removes these dirty objects from memory.

Before re-claiming memory from an object (removing the object from memory), the garbage collector calls the *Finalize()* method (discussed in lesson 4) of the object so as to allow the object to free its resources. Hence, you should override the Object class' *Finalize()* method in your classes, if you want to free some un-managed resources held by your objects. Since the *Finalize()* method is called by the Garbage Collector before re-claiming the object, we are never to know exactly when it will be called. It may be called instantly after an object is un-referenced or sometime later when the CLR needs to re-claim some memory. You may optionally implement the IDisposable interface and override its *Dispose()* method if you want to allow the user of your class to specify (by calling the *Dispose()* method) when the resources occupied by the object should be freed. Although we haven't yet discussed interfaces, but I am giving an example here of implementing the IDisposable interface as you will see this method quite often in your future programs. I recommend you to comeback and read this again when we have focused on interfaces later.

```
Imports System
Module Test
    Public Sub Main()
        Dim garCol As New GarbageCollection()
        garCol.DoSomething()
        garCol.Dispose()
    End Sub
End Module
Class GarbageCollection
    Implements IDisposable
    Public Sub DoSomething()
        Console.WriteLine("Performing usual tasks...")
    End Sub
    Public Sub Dispose() Implements IDisposable.Dispose
        GC.SuppressFinalize(Me)
        Console.WriteLine("Disposing object...")
        Console.WriteLine("Freeing resources captured by this object...")
    End Sub
    Protected Overrides Sub Finalize()
        Console.WriteLine("Destructing object...")
        Console.WriteLine("Freeing resources captured by this object...")
    End Sub
End Class
```

In the above example we have declared a class named *GarbageCollection* which implements the *IDisposable* interface and overrides its *Dispose()* method. Now, the user or client of the *GarbageCollection* class can decide when to free-up the resources held by the object by calling the *Dispose()* method. The common usage of this class is demonstrated in the Test Module's *Main()* method. Note that we called the *SuppressFinalize()* method of the *System.GC* class in the *Dispose()* method.

```
    Public Sub Dispose() Implements IDisposable.Dispose
        GC.SuppressFinalize(Me)
        Console.WriteLine("Disposing object...")
        Console.WriteLine("Freeing resources captured by this object...")
    End Sub
```

The *System.GC* class can be used to control the Garbage Collector. As a result of passing the current object in the *GC.SuppressFinalize()* method, the *Finalize()* method of the current object won't be called. If we execute the above program, we will get the following output

```
Performing usual tasks...
Disposing object...
Freeing resources captured by this object...
Press any key to continue
```

But if we comment out the call to the *Dispose()* method in our *Main()* method like

```
    Public Sub Main()
        Dim garCol As New GarbageCollection()
        garCol.DoSomething()
        'garCol.Dispose()
    End Sub
```

The runtime will call the *Finalize()* method when the object is no longer referenced. The output would be like

```
Performing usual tasks...
Destructing object...
Freeing resources captured by this object...
Press any key to continue
```

Finalize() and Performance Overhead
Having a *Finalize()* method in your class creates some performance overhead, therefore it is not recommended to declare these in your classes unless your class is holding un-managed resources such as file handles, database resources or internet connection.

System.GC.Collect() method
The *System.GC* class provides a *Shared* method named *Collect()* which enforces the Garbage Collection to commence at your command. It is usually better to leave the invocation of the Garbage Collector to the .Net Runtime. It is not wise to start a Garbage Collector process just for freeing some objects in memory. Although you may use the method when a large number of objects in your program are un-referenced (E.g., an array or a collection) and you want the memory to be re-claimed instantly.

Nested Classes in VB.Net
So far we have only seen the top-level classes (classes only bound by the namespace) in our VB.Net programs. We can also define nested classes in VB.Net. Nested classes (also called inner classes) are defined inside another class as below

```
Imports System
Namespace CSharpSchool
    Module Test
```

```
        Public Sub Main()
            Dim inner As New Outer.Inner()
            inner.InnerFun()
        End Sub
    End Module
    Class Outer
        Private Shared name As String = "Faraz"
        Public Sub New()
            Console.WriteLine("Constructor of Outer called...")
        End Sub
        Public Class Inner
            Public Sub New()
                Console.WriteLine("Constructor of Inner called...")
            End Sub
            Public Sub InnerFun()
                Console.WriteLine("InnerFun() called...")
                Console.WriteLine("Name in Outer is {0}", name)
            End Sub
        End Class
    End Class
End Namespace
```

We defined a nested class named *Inner* which is contained inside the top-level class *Outer*. Both the *Outer* and *Inner* classes have a no-argument constructor. While the top-level classes can only be marked as *Public* or *Friend*, nested classes can be marked with all the access modifiers: *Private, Public, Protected, Friend* OR *Protected* with the same meaning as described in lesson 5. Like other class members, the default access modifier for a nested class is *Private* which means that a nested class can not be referenced and accessed outside a containing class. In order to reference and instantiate ane Inner class in the *Main()* method of the *Test* Module, we marked it as *Public* in our program. It is also worth-noting that nested classes are always accessed and instantiated with reference to their container or enclosing class. That is the reason why we instantiated the Inner class in our *Main()* method as

```
     Public Sub Main()
         Dim inner As New Outer.Inner()
         inner.InnerFun()
     End Sub
```

Another important point about a nested classes is that they CAN access all the (*Private, Friend, Protected, Public*) *Shared* members of the enclosing class. We have demonstrated this in the *InnerFun()* method of the *Inner* class where it accesses the *Private* member (*name*) of the enclosing (*Outer*) class.

```
Public Sub InnerFun()
        Console.WriteLine("InnerFun() called...")
        Console.WriteLine("Name in Outer is {0}", name)
End Sub
```

When we compile and run the above code, we get the following output.

```
Constructor of Inner called...
InnerFun() called...
```

```
Name in Outer is Faraz
Press any key to continue
```

Author's Note: Nested classes in VB.Net are similar to Java's static inner classes. There is no concept of Java's non-static inner classes in VB.Net

Since that nested classes are instantiated with reference to the enclosing class, their access protection level always remains less than or equal to that of the enclosing class. Hence, if the access level of the enclosing class is *Friend* and the access level of nested class is *Public*, the nested class will only be accessible in the current assembly (project).

Food for thought: Exercise 6

1. How are Structures different from classes?
2. What is the difference between instantiating structures with and without using the New keyword?
3. How can we achieve the functionality provided by an enumeration using only a class or a structure? (Java developers use this technique quite often as there is no support for enumeration in Java).
4. What is the Finalize() method? When does the Finalize() method get called?
5. How can we enforce the garbage collector to be invoked? How can we enforce the garbage collector not to call the Finalize() method in our class?

Solution of Last Issue's Exercise (Exercise 5)

1. How are inheritance and polymorphism in VB.NET different from that in Java and C++?

VB.Net, like C# and Java, only allows single class inheritance contrary to C++ which allows multiple inheritance of classes. VB.Net provides more access modifiers for the class than Java and C++. VB.Net, like C++ and unlike Java, requires methods to be marked *Overridable* if we want to achieve polymorphism on them. VB.Net also requires a method to be marked as *Overrides* if it is overriding a overridable method in the base class. In VB.Net you can mark a method with the Shadows keyword if it is not overriding the same method in the base class and is providing a new implementation.

2. How can we make

a) a variable so that its value can not be changed once assigned at declaration time?

b) a reference to an object so that it can not reference any other object, except for the one it is made to reference at declaration time?

c) a method so that it can not be overridden by any of its sub-classes?

d) a class that can not be inherited by any other class?

a) by marking a variable with the Const keyword
b) the is no way :)
c) by not marking it with the Overridable keyword.
d) by marking it with the NotInheritable keyword

3. Can we override properties and apply polymorphism to these?

Yes we can override properties and apply polymorphism to them as they are similar to Get/Set methods.

4. We saw that constructors are called in the order from top to bottom (parent to child class) in inheritance hierarchy. In which order are the Finalize() called during inheritance?

The Finalize() are called in the reverse order, i.e., from the bottom to the top (child to parent class) in inheritance hierarchy.

5. What are the advantages of using protected access modifiers in inheritance?
Protected modifiers makes sure that the member is accessible only within the containing class or the class that has inherited the containing class.

6. How is type casting different from boxing/un-boxing?
Type casting does not change the physical location of objects but just changes the appearance of the object while boxing makes a copy of the value type from the stack and places it inside an object of type System.Object in the heap.

7. When I compiled and run the following program

```
Module Test
    Public Sub Main()
        Dim theParent As Parent = New Child()
        Console.WriteLine("i = {0}", theParent.i)
        theParent.MyMethod()
    End Sub
End Module

Class Parent
    Public i As Integer = 5
    Public Overridable Sub MyMethod()
        Console.WriteLine("I am Parent's MyMethod()")
    End Sub
End Class
Class Child
    Inherits Parent
    Public i As Integer = 7
    Public Overrides Sub MyMethod()
        Console.WriteLine("I am Child's MyMethod()")
    End Sub
End Class
```

It outputs as

```
i = 5
I am Child's MyMethod()
Press any key to continue
```

As we called both i and MyMethod() using the Parent type reference containing the Child type object, why is i of the Parent and MyMethod() of the Child called?
Fields and properties are always called with reference to the reference type while methods are called with reference to the object type when applying polymorphism

What's Next...
Next time, we will be discussing Abstract (*MustInherit*) classes and Interfaces, their purpose, advantage and limitations for using these and how these are implemented in VB.Net. We will see

- Abstract (*MustOverride*) methods and abstract (*MustInherit*) classes
- Inheriting/practicing abstract (*MustInherit*) classes.
- Interfaces, the concept behind interfaces
- Implementing interface
- Casting to and from Interfaces, *TypeOf...is* and *CType()* operators

Abstract classes and Interfaces

Lesson Plan

In this lesson we will explore abstract (*MustInherit*) classes and interfaces. We will look at the idea behind the abstract (*MustOverride*) methods, abstract classes (*MustInherit*), interfaces and how they are implemented in VB.Net. Later we will see how to cast to and from the interface reference by using the *TypeOf...is* and *CType()* operators

Abstract (MustInherit) Classes

Abstract classes can simply defined as incomplete classes.

- Abstract (*MustInherit*) classes contain one or more incomplete methods called abstract (*MustOverride*) methods.
- The abstract (*MustInherit*) class only provides the signature or declaration of the abstract (*MustOverride*) methods and leaves the implementation of these methods to the derived or sub-classes.
- Abstract classes are marked with *MustInherit* and abstract methods are marked with the *MustOverride* keyword
- Since abstract classes are incomplete; they **can not** be instantiated. They must be sub-classed in order to use their functionality. This is the reason why an abstract class can't be *NotInheritable*
- A class inheriting an abstract class must implement all the abstract methods in the abstract class or it must also be declared as an *MustInherit* class.
- A class inheriting an abstract class and one that implements all its abstract methods is called a concrete class of that abstract class.
- We can declare a reference of the type of abstract class and it can point to the objects of classes that have inherited the abstract class.

Let us declare an abstract class with two concrete properties and an incomplete (*MustOverride*) method.

```
MustInherit Class TaxCalculator
    Protected mItemPrice As Double
    Protected mTax As Double
    ' an abstract (MustOverride) function
    Public MustOverride Function CalculateTax() As Double
    ' Two concrete properties
    Public ReadOnly Property Tax() As Double
        Get
            Return Tax
        End Get
    End Property
    Public ReadOnly Property ItemPrice() As Double
        Get
            Return ItemPrice
        End Get
    End Property
End Class
```

The abstract (*MustInherit*) *TaxCalculator* class contains two fields: *mItemPrice* and applied tax (*mTax*). It contains an abstract *MustOverride* function *CalculateTax()* which calculates the tax applied on the *mItemPrice* and stores it in the field *mTax*. The *CalculateTax()* function is made abstract so the concrete sub-classes can provide their own criteria for applying the *tax* on the *itemPrice*. The class also contains two public read only properties to access the two private fields. If we try to instantiate this abstract class in the *Main()* method

```
    Public Sub Main()
        Dim taxCalc As New TaxCalculator()
    End Sub
```

The compiler will complain as so

'New' cannot be used on class 'TaxCalculator' because it contains a 'MustOverride' member that has not been overridden.

In order to create an instance of the *TaxCalculator* class, we need to sub-class it. Let us now inherit a class from the MustInherit *TaxCalculator* class and call it *SalesTaxCalculator*

```
Class SalesTaxCalculator
    Inherits TaxCalculator
    Public Sub New(ByVal pItemPrice As Double)
        Me.mItemPrice = pItemPrice
    End Sub
    Public Overrides Function CalculateTax() As Double
        mTax = 0.3 * mItemPrice
        Return mItemPrice + mTax
    End Function
End Class
```

The *SalesTaxCalculator* class inherits *TaxCalculator* and overrides its *CalculateTax()* functions. It applies 30% tax on the price of item (a bit harsh!) and returns the new price of the item. The *SalesTaxCalculator* class also defines a constructor that takes the *itemPrice* as its parameter. If we don't provide the implementation of the *CalculateTax()* method in *SalesTaxCalculator*

```
Class SalesTaxCalculator
    Inherits TaxCalculator
    Public Sub New(ByVal pItemPrice As Double)
        Me.mItemPrice = pItemPrice
    End Sub
    'Public Overrides Function CalculateTax() As Double
    '    mTax = 0.3 * mItemPrice
    '    Return mItemPrice + mTax
    'End Function
End Class
```

We will get a compile time error as

Class 'SalesTaxCalculator' must either be declared 'MustInherit' or override the following inherited 'MustOverride' member(s): Public MustOverride Function CalculateTax() As Double.

OK, un-comment the overridden *CalculateTax()* method in *SalesTaxCalculator*. Since we have overridden the *CaculateTax()* method of *TaxCalculator* in the *SalesTaxCalculator* class, we can create its instance in the *Main()* method

```
    Public Sub Main()
        Dim salesTaxCalc As New SalesTaxCalculator(225)
        Dim newPrice As Double = salesTaxCalc.CalculateTax()
        Console.WriteLine("The item price changed because of sales _
                          tax from {0} $ to {1} $", _
                          salesTaxCalc.ItemPrice, newPrice)
        Console.WriteLine("Tax applied = {0} $", salesTaxCalc.Tax)
    End Sub
```

Here we instantiated the *SalesTaxCalculator* class just like a regular class and accessed its members. The output of the above program will be

The item price changed because of sales tax from 225 $ to 292.5 $
Tax applied = 67.5 $
Press any key to continue

We can also use the abstract class type (*TaxCalculator*) reference to handle the object of its concrete class (*SalesTaxCalculator*) in our *Main()* method

```
    Public Sub Main()
        Dim salesTaxCalc As TaxCalculator = New SalesTaxCalculator(225)
        Dim newPrice As Double = salesTaxCalc.CalculateTax()
        Console.WriteLine("The item price changed because of _
                          sales tax from {0} $ to {1} $", _
                          salesTaxCalc.ItemPrice, newPrice)
        Console.WriteLine("Tax applied = {0} $", salesTaxCalc.Tax)
    End Sub
```

We can derive as much concrete classes as we want from the abstract *TaxCalculator* class as long as they provide the definition of the abstract (*MustOverride*) methods of it. Here is another concrete class (*WarSurchargeCalculator*) of the abstract *TaxCalculator* class.

```
Class WarSurchargeCalculator
    Inherits TaxCalculator
    Public Sub New(ByVal pItemPrice As Double)
        Me.mItemPrice = pItemPrice
    End Sub
    Public Overrides Function CalculateTax() As Double
        mTax = 0.5 * mItemPrice
        Return mItemPrice + mTax
    End Function
```

```
End Class
```

The *WarSurchargeCalculator* can be used similarly in the *Main()* method.

Interfaces
Interfaces are a special kind of type in VB.Net which are used to define the specifications (in terms of method signatures) that must be followed by sub-types.

- An interface is declared using the *Interface* keyword.
- An interface, like the abstract (*MustInherit*) classes, can not be instantiated
- An interface may contain signatures of methods, properties and indexers
- An interface is a type with members that are all Public. *MustOverride* is the default.
- An interface is implemented by a class. A class implementing the interface must provide the body for all the members of an interface.
- To implement an interface, a class uses the *Implements* keyword to show that it is implementing a particular interface.
- A class can implement more than one interface contrary to class-inheritance where you can only inherit one class.
- An interface itself can inherit other interfaces.
- We can declare the reference of the interface type and it can point to any class implementing the interface.
- It is a convention in VB.Net to prefix the name of interfaces with uppercase '**I**' like *IDisposable, ISerializable, ICloneable, IEnumerator*, etc

Let's define an interface named *IWindow* as

```
Interface IWindow
    Property Position() As Point
    Property Title() As String
    Sub Draw()
    Sub Minimize()
End Interface
```

The interface above defines a new type *IWindow*. The interface contains two properties with *Get* and *Set* blocks for the *Position* and *Title* of the Window. It also contains the signature of two methods to *Draw()* and *Minimize()* the window. All the classes that wish to implement (or realize) this interface must have to provide the body for these members. The property *Position* is of type *Point*, which is a *Structure* type and defined in the program like so

```
Structure Point
    Public X As Integer
    Public Y As Integer
    Public Sub New(ByVal pX As Integer, ByVal pY As Integer)
        X = pX
        Y = pY
    End Sub
End Structure
```

It is important to understand that an interface declares abstract types and provides a specification which must be followed by all implementing classes. The compiler enforces this specification, and does not compile any concrete classes which inherits the interface but does not implement (provide the body of) all the members of the interface.

Let's now define a *RectangularWindow* class that implements our *IWindow* interface.

```
Class RectangularWindow
    Implements IWindow
    Private mTitle As String
    Private mPosition As Point
    Public Sub New(ByVal pTitle As String, ByVal pPosition As Point)
        Me.mTitle = pTitle
        Me.mPosition = pPosition
    End Sub
    Public Property Position() As Point Implements IWindow.Position
        Get
            Return mPosition
        End Get
        Set(ByVal Value As Point)
            mPosition = Value
        End Set
    End Property
    Public Property Title() As String Implements IWindow.Title
        Get
            Return mTitle
        End Get
        Set(ByVal Value As String)
            mTitle = Value
        End Set
    End Property
    Public Sub Draw() Implements IWindow.Draw
        Console.WriteLine("Drawing Rectangular Window")
    End Sub
    Public Sub Minimize() Implements IWindow.Minimize
          Console.WriteLine("Minimizing Rectangular Window")
    End Sub
End Class
```

The *RectangularWindow* class implements the *IWindow* interface using the *Implements* keyword

```
Class RectangularWindow
    Implements IWindow
```

Since the *RectangularWindow* is implementing the *IWindow* interface and it is not declared as an abstract (*MustInherit*) class, it provides the body for all the members of the *IWindow* interface. It contains two private fields: *mTitle* and *mPosition*. The properties *Title* and *Position* specified in the *IWindow* interface are implemented as accessors to the *mTitle* and the *mPosition* fields respectively. The two methods, *Draw()* and *Minimize()* are implemented to print a simple message.

Note that all the implementing members in *RectangularWindow* are declared with the *Implements* keyword followed by the actual interface member they are implementing. This specification with each implementing member ensures that a class does not implement (override) the interface member accidentally or without the developer's intention.

```
    Public Sub Minimize() Implements IWindow.Minimize
          Console.WriteLine("Minimizing Rectangular Window")
    End Sub
```

If we don't mark these members with *Implements* keyword like

```
    Public Sub Minimize()
          Console.WriteLine("Minimizing Rectangular Window");
    End Sub
```

We will get the compile-time error as

'VBDotNetSchoolLesson7.RectangularWindow' must implement 'Sub Minimize()' for interface 'VBDotNetSchoolLesson7.IWindow'.

The compiler is quite supportive here and suggests some possible reasons for the error.

Also note that there is no *Overrides* keyword when overriding the abstract methods of the interface (*IWindow*) in the implementing class (*RectangularWindow*), like we used to have in VB.Net during polymorphism. If you write *Overrides* before a member, you will get the compile time error below.

sub 'Draw' cannot be declared 'Overrides' because it does not override a sub in a base class.

The reason for not applying the *Overrides* keyword here is that we do not actually override the default implementation but provide our own specific implementation for the members which we assert with the *Implements* keyword.

Implementing More Than One Interfaces

A class can implement more than one interface. In such a case, a class has to provide the implementation for all the members of each of the implementing interfaces. Suppose we have two interfaces *ILoggable* and *IFile* as

```
Interface ILoggable
    Sub Log(ByVal filename As String)
End Interface
Interface IFile
    Function ReadLine() As String
    Sub Write(ByVal s As String)
End Interface
```

The *ILoggable* interface contains a single method *Log(string)* which logs the activities in a specified file while the interface *IFile* contains two methods: *ReadLine()* and *Write(string)*. The method *ReadLine()* reads a line from the specified file and *Write(string)* writes the supplied string to the file. Let's define a class (*MyFile*) which implements these interfaces.

```
Class MyFile
    Implements ILoggable, IFile
    Private filename As String
    Public Sub New(ByVal filename As String)
        Me.filename = filename
    End Sub
    Public Sub Log(ByVal filename As String) Implements ILoggable.Log
        Console.WriteLine("Logging activities in file: {0}", filename)
    End Sub
    Public Function ReadLine() As String Implements IFile.ReadLine
        Return "A line from MyFile"
    End Function
    Public Sub Write(ByVal s As String) Implements IFile.Write
        Console.WriteLine("Writing '{0}' in the file", s)
    End Sub
End Class
```

The *MyFile* implements both the *ILoggable* and the *IFile* interfaces and uses a comma ',' to separate the list of interfaces as

```
Class MyFile
    Implements ILoggable, IFile
```

The class provides the implementation of the methods specified in the two implemented interfaces (*ILoggable* and *IFile*). *MyFile* also contains a constructor that accepts the name of the file. The *ReadLine()* and *Write()* methods operates on the file supplied in the constructor. The *Log()* method records the activities of this process to the specified method. The *Test* module containing the *Main()* method is implemented as

```
Module Test
    Public Sub Main()
        Dim aFile As New MyFile("myfile.txt")
        aFile.Log("c:\VBDotNet.txt")
        aFile.Write("My name is Faraz")
```

```
        Console.WriteLine(aFile.ReadLine())
    End Sub
End Module
```

Here, we created an instance of *MyFile* named *aFile*. We then called different methods of the class *MyFile* using its object. The sample output of the program is

```
Logging activities in file: c:\VBDotNet.txt
Writing 'My name is Faraz' in the file
Reading a line from MyFile
Press any key to continue
```

Up to this point there should be no confusion about implementing multiple interfaces. But what if the name of the method in *ILoggable* was *Write(string)* rather *Log(string)*? The class *MyFile* would then be implementing two interfaces (*ILoggable* and *IFile*) and both with a method *Write(string)* with similar signatures.

```
Interface ILoggable
    Sub Write(ByVal filename As String)
End Interface
Interface IFile
    Function ReadLine() As String
    Sub Write(ByVal s As String)
End Interface
```

Explicit implementation of methods

VB.Net uses explicit implementation of all the members of implementing interfaces using the explicit Implements keyword after the member definition. Hence, even if a class is implementing more than one interface and at least two of them have methods with similar signatures, then there should be no problem and ambiguity. Consider the case defined above, where we have two interfaces (*ILoggable* and *IFile*) where both contain a *Write(string)* method with an identical signature. Our class *MyFile* accordingly provides explicit implementations of both *Write()* methods.

```
Class MyFile
    Implements ILoggable, IFile
    Private filename As String
    Public Sub New(ByVal filename As String)
        Me.filename = filename
    End Sub
    Public Sub Log(ByVal filename As String) Implements ILoggable.Write
        Console.WriteLine("Logging activities in file: {0}", filename)
    End Sub
    Public Function ReadLine() As String Implements IFile.ReadLine
        Return "Reading a line from MyFile"
    End Function
    Public Sub Write(ByVal s As String) Implements IFile.Write
        Console.WriteLine("Writing '{0}' in the file", s)
    End Sub
End Class
```

Here we defined the *Write()* method of *ILoggable* explicitly by marking it with the interface name as

```
    Public Sub Log(ByVal filename As String) Implements
ILoggable.Write
```

The two *Write()* methods are distinguishable in the class definition for the compiler. If we write our *Main()* method as

```
    Public Sub Main()
        Dim aFile As New MyFile("myfile.txt")
        aFile.Log("c:\VBDotNet.txt")
        aFile.Write("My name is Faraz")
        Console.WriteLine(aFile.ReadLine())
    End Sub
```

We would get the following output

```
Logging activities in file: c:\VBDotNet.txt
Writing 'My name is Faraz' in the file
Reading a line from MyFile
Press any key to continue
```

In fact, we can define only a single method to implement both of these methods (*ILoggable.Write* and *IFile.Write*) as

```
Class MyFile
    Implements ILoggable, IFile
    Private filename As String
    Public Sub New(ByVal filename As String)
        Me.filename = filename
    End Sub
    'Public Sub Log(ByVal filename As String) Implements ILoggable.Write
    '    Console.WriteLine("Logging activities in file: {0}", filename)
    'End Sub
    Public Function ReadLine() As String Implements IFile.ReadLine
        Return "Reading a line from MyFile"
    End Function
    Public Sub Write(ByVal s As String) Implements IFile.Write, ILoggable.Write
        Console.WriteLine("Writing '{0}' in the file", s)
    End Sub
End Class
```

Here we have used a single *Write()* method in the *MyFile* class to implement the two *Write()* methods in *ILoggable* and *IFile* and have separated the interface members specification with a comma

```
    Public Sub Write(ByVal s As String) Implements IFile.Write, ILoggable.Write
```

Note that we have commented the separate implementation of the *ILoggable.Write* in *MyFile.Log* method as a class can not implement a member of an interface more than once

```
    'Public Sub Log(ByVal filename As String) Implements ILoggable.Write
    '    Console.WriteLine("Logging activities in file: {0}", filename)
    'End Sub
```

Note that in both the calls to the *Write()* method, the implicit version (*IFile.Write(string)*) is executed. To call the *ILoggable.Write()* method

Casting to and from interfaces and using TypeOf...is and CType operators
As an interface is a parent or base type of its implementing classea; a reference of an interface type can refer to the object of the implementing class.

```
        Dim aFile As New MyFile("myfile.txt")
        Dim fileRef As IFile = aFile
        fileRef.Write("This is a test message")
```

Here we defined a reference of interface type '*fileRef*' and made it point to the object of its implementing type. Note that we can only access members specified in the *IFile* interface through the reference '*fileRef*' on the *aFile* object.

In the above example, we knew that the object (*aFile*) is castable to the *IFile* interface. So we casted it straightforward. However, in such a case when we are not sure whether a particular object is really castable to an interface type at run-time, we can make use of the *TypeOf...is* operator (The *TypeOf...is* operators are discussed in more detail in lesson 5 - Inheritance and Polymorphism).

The *TypeOf...is* operator is used to check the type of particular object and it returns a *Boolean* result. For example,

```
      Dim aFile As New MyFile("myfile.txt")
      Console.WriteLine(TypeOf aFile Is ILoggable)
```

will output True as *MyFile* is a sub-type of *ILoggable* while

```
      Dim s As String = "faraz"
      Console.WriteLine(TypeOf s Is ILoggable)
```

will output as *False*.

To cast an object referenced by the interface reference to a specific class reference, we can again use the *CType()* built-in method

```
    Public Sub Main()
        Dim aFile As New MyFile("myfile.txt")
        Dim fileRef As IFile = aFile
        fileRef.Write("This is a test message")
        aFile = CType(fileRef, MyFile)
        aFile.Log("abc.log")
    End Sub
```

We created an object of the type *MyFile*, casted it to the interface type reference *fileRef*, called a member method of *IFile* and converted its type back to the *MyFile* class and called the *Log()* method defined in the *ILoggable* interface

Finally, there is a point that should be remembered when using interfaces.

- It is generally preferred to access the members of interfaces using the interface reference.

An interface inheriting one or more interfaces
An interface can inherit from more than one interface. Suppose the *IFile* interface inherits two other interfaces *IReadable* and *IWritable*

```
Interface IWritable
    Sub Write(ByVal s As String)
End Interface
Interface IReadable
    Function ReadLine() As String
end interface
Interface IFile
    Inherits IWritable, IReadable
    Sub Open(ByVal filename As String)
    Sub Close()
End Interface
```

The interface *IWritable* contains a method *Write(string)* that writes the supplied string to the file. While the interface *IReadable* contains a method *ReadLine()* that returns a string from the file. The interface *IFile* inherits the two interfaces (*IWritable* and *IReadable*) and also contains two methods (*Open(string)* and *Close()*) to open and close the file. Let's now define a class named *MyFile* that implements the interface *IFile*. The class has to provide the body of all the methods present (directly or inherited) in the *IFile* interface.

```
Class MyFile
    Implements IFile
    Private mFilename As String
    Public Sub Open(ByVal pFilename As String) Implements IFile.Open
        mFilename = pFilename
        Console.WriteLine("Opening file: {0}", mFilename)
    End Sub
    Public Function ReadLine() As String Implements IFile.ReadLine
        Return "Reading a line from MyFile: " + mFilename
    End Function
    Public Sub Write(ByVal s As String) Implements IFile.Write
        Console.WriteLine("Writing `{0}' in the file: {1}", s, mFilename)
    End Sub
    Public Sub Close() Implements IFile.Close
        Console.WriteLine("Closing the file : {0}", mFilename)
    End Sub
End Class
```

The class provides a simple implementation of all the methods by printing a message in the body of each method. The *Test* Module containing the *Main()* method is

```
Module Test
    Public Sub Main()
        Dim aFile As New MyFile()
        aFile.Open("c:\VBDotNet.txt")
        aFile.Write("My name is Faraz")
        Console.WriteLine(aFile.ReadLine())
        aFile.Close()
    End Sub
End Module
```

Here we created an instance of *MyFile* and named the reference as *aFile*. We later called different methods in the object using the reference. The output of the program is

```
Opening file: c:\VBDotNet.txt
Writing `My name is Faraz' in the file: c:\VBDotNet.txt
Reading a line from MyFile: c:\VBDotNet.txt
Closing the file : c:\VBDotNet.txt
Press any key to continue
```

The output shows the result of the methods as they are called in order.

Food for thought: Exercise

1. How are abstract (*MustInherit*) classes different from normal classes and interfaces?
2. Write an abstract class that declares a read only abstract property.
3. Abstract classes may contain constructors. When we can't instantiate the abstract classes, what is the use of these constructors?
4. Interfaces are also called specification. Why?
5. We have seen in our programs that sub-classes can call the methods defined in the base-class. Can you make a program where a base-class calls the method defined in a sub-class? (Tip: Try using abstract class)

Solution of Last Issue's Exercise (Exercise 6)

1. How are Structures different from a class?

Structures are light weight data-types used to represent simple types. They are much more restricted than classes. A structure can neither inherit nor can be inherited by any other class. A structure can't have a no-argument constructor. A structure is created on the stack (hence value-type) while a class is created on the heap (hence a reference-type). A structure can be created with or without using the *New* keyword.

2. What is the difference between instantiating structures with and without using the New keyword?

When a structure is instantiated using the *New* keyword, a constructor (no-argument or custom if provided) is called which initializes the fields in the structure. When a structure is instantiated without using the *New* keyword, no constructor is called. Hence, one should provide the initialization values for its members.

3. How can we achieve the functionality provided by enumeration using only a class or a structure? (Java developers use this technique quite often as there is no support for enumeration in Java).

The following program achieves the same functionality provided by the enumeration through a class with a private constructor having static object type. The description of a class is provided in between the code in the form of code comments

```
' class abstracting the functionality of enumeration
Class SortingCriteria
    Private meName As String     ' name of the enumeration
    Private meValue As Integer   ' value of named constant
    ' constructor is made private, so no one can make the object
    ' outside the class
    Private Sub New(ByVal eName As String, ByVal eValue As Integer)
        Me.meName = eName
        Me.meValue = eValue
    End Sub
    ' overriden ToString() method, so the object can be printed
    ' using the reference
    Public Overrides Function ToString() As String
        Return meName
    End Function
    Public ReadOnly Property Value() As Integer
        Get
            Return meValue
        End Get
    End Property
    Public ReadOnly Property Name() As String
        Get
            Return meName
        End Get
    End Property
    ' named constants made public to be accessed outside the class
    Public Shared ByName As New SortingCriteria("ByName", 0)
```

```
    Public Shared ByType As New SortingCriteria("ByType", 1)
    Public Shared BySize As New SortingCriteria("BySize", 2)
    Public Shared ByDate As New SortingCriteria("ByDate", 3)
End Class
```

4. What is the Finalize() method? When does the Finalize() method get called?
The *Finalize()* method is a member of the *System.Object* class. *Finalize()* contains the code for freeing resources when the object is about to be garbage collected. It is called by the .Net Runtime and we can not predict when it will be called. It is guaranteed to be called when there is no reference pointing to the object and the object is about to be garbage collected.

5. How can we enforce the garbage collector to be invoked? How can we enforce the garbage collector not to call the Finalize() method of our class?
We can enforce the garbage collector to be invoked by calling the *Shared* method *System.GC.Collect()*. We can enforce the garbage collector not to call the *Finalize()* method on particular objects by making a call to the *Shared* method *System.GC.SuppressFinalize(obj)*, where *obj* is the object reference.

What's Next...
Next time, we will be discussing Multi-dimensional arrays, collections and string handling in VB.Net. We will explore

- Arrays Revisited (rectangular, jagged, multi-dimensional)
- The For Each loop
- ArrayList, Stack, Queue, Stacks
- Dictionaries
- Operations on string class
- String Builder class and its operations

Arrays, collections and string Manipulation

Lesson Plan

In this lesson we will explore arrays, collections and string manipulation in VB.Net. First of all, we will explore multidimensional (rectangular and jagged) arrays. We will also explore how *For Each* iterates through a collection. Then we will move on to collections and see how they are implemented. Later, we will explore different collections like *ArrayList, Stack, Queue* and *Dictionaries*. Finally, we will see how strings are manipulated in VB.Net. We will explore both the *String* and *StringBuilder* types.

Arrays Revisited

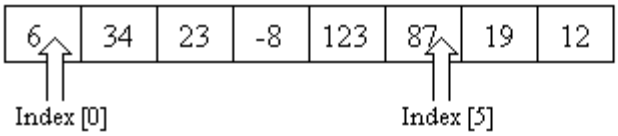
As we have seen earlier, an array is a sequential collection of elements of a similar data type. In VB.Net, an array is an object and thus a reference type, and therefore they are stored on the heap. We have only covered the single dimensional arrays in the previous lessons, now we will explore multidimensional arrays.

Multidimensional Arrays

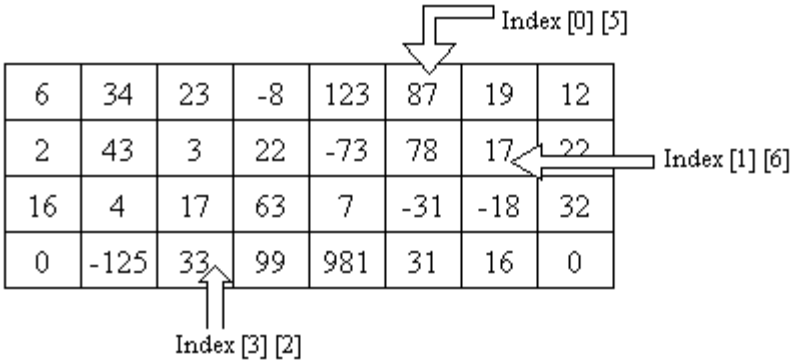
A multidimensional array is an 'array of arrays'. A multidimensional array is the one in which each element of the array is an array itself. It is similar to tables of a database where each primary element (row) is the collection of other secondary elements (columns). If the secondary elements do not contain a collection of other elements, it is called a 2-dimensional array (the most common type of multidimensional array), otherwise it is called an **n**-dimensional array where **n** is the depth of the chain of arrays. There are two types of multidimensional arrays in VB.Net 1.A Rectangular array (the one in which each row contains an equal number of columns)
2.A Jagged array (the one in which each row does not necessarily contain an equal number of columns)

The images below show how the different kinds of array looks. This figure also shows the indexes of the different elements in the arrays. Remember, the first element of an array is always zero (0).

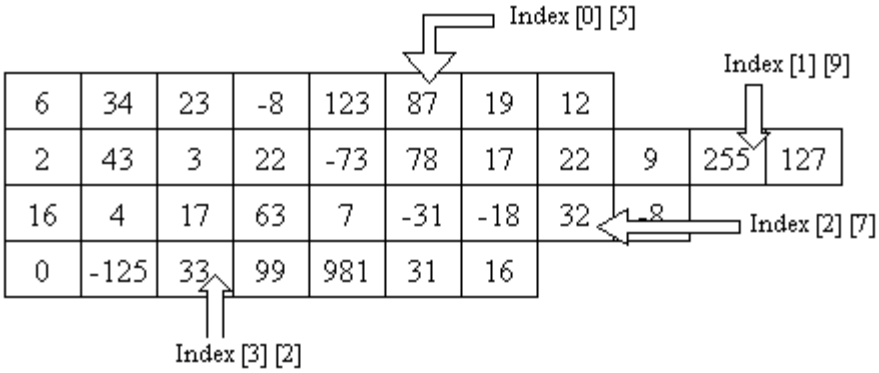
A Single Dimensional Array



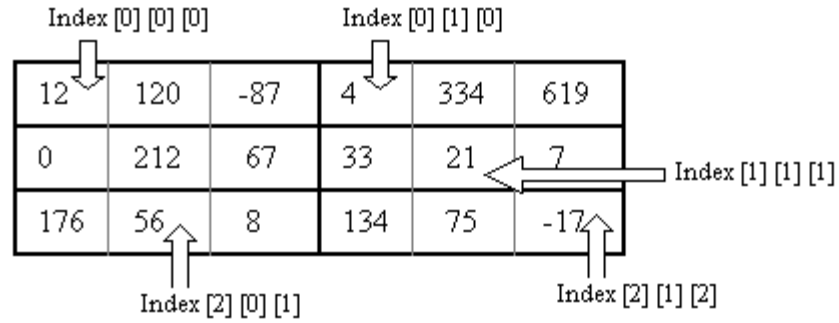
A two-dimensional (rectangular) Array (4 by 4)



A two-dimensional (jagged) Array



A three-dimensional Array (3 by 2 by 3)



Instantiation and accessing the elements of multidimensional arrays
Recall that we instantiate our single dimensional arrays like this:

```
        Dim intArray(4) As Integer
```

The above line would instantiate (create) a one dimensional array (*intArray*) of type Integer whose length would be 5. We can access the elements of the array like this:

```
        intArray(0) = 45 ' set the first element to 45
        intArray(2) = 21 ' set the third element to 21
        intArray(4) = 9          ' set the fifth and last element to 9
```

Instantiating a multidimensional array is almost identical to the above procedure as long as you keep the most basic definition of the multidimensional array in mind which is 'a multidimensional array is an array of arrays'. Suppose we wish to create a two dimensional rectangular array with 2 rows and 3 columns. We can instantiate the array as follows

```
        Dim myTable(1, 2) As Integer
```

All the elements of the array are auto-initialized to their default values; hence all the elements of our *myTable* array would be initialized with zero. We can iterate through this array using either a *For Each* or a *For...Next* loop.

```
        Dim intVal As Integer
        For Each intVal In myTable
            Console.WriteLine(intVal)
        Next
```

When it is compiled and executed, it will print six (2 x 3) zeros at the console.

```
0
0
0
0
0
0
```

Let's change the values of the individual elements of the array. To change the value of the first element of the first row to 32, we can use the following code

```
        myTable(0,0) = 32
```

In the same way, we can change the values of other elements in the array

```
        myTable(0,1) = 2
        myTable(0,2) = 12
        myTable(1,0) = 18
```

```
        myTable(1,1) = 74
        myTable(1,2) = -13
```

We can use a for loop to iterate this array

```
        Dim row As Integer
        Dim col As Integer
        For row = 0 To myTable.GetLength(0) - 1
            For col = 0 To myTable.GetLength(1) - 1
                Console.WriteLine("Element at ({0},{1}) is {2}", row, col,
myTable(row, col))
            Next
        Next
```

Above, we have used two *For...Next* loops to iterate through each of the two dimensions of the array. We have used the *GetLength()* method of the *System.Array* class (the underlying class for arrays in .Net) to find the length of a particular dimension of the array. Note that the *Length* property will give a total number of elements within this two dimensional array, i.e., 6. The output of the above program will be

```
Element at (0,0) is 32
Element at (0,1) is 2
Element at (0,2) is 12
Element at (1,0) is 18
Element at (1,1) is 74
Element at (1,2) is -13
Press any key to continue
```

Instantiating and accessing a Jagged Array

A jagged array is an array in which the length of each row is not the same. For example we may wish to create a table with 3 rows where the length of first row is 3, the second row is 5 and the third row is 2. We can instantiate this jagged array as

```
        Dim myTable(2)() As Integer
        myTable(0) = New Integer(2) {}
        myTable(1) = New Integer(4) {}
        myTable(2) = New Integer(1) {}
```

Then we can fill the array as

```
        myTable(0)(0) = 3
        myTable(0)(1) = -2
        myTable(0)(2) = 16
        myTable(1)(0) = 1
        myTable(1)(1) = 9
        myTable(1)(2) = 5
        myTable(1)(3) = 6
        myTable(1)(4) = 98
        myTable(2)(0) = 19
        myTable(2)(1) = 6
```

We will show you how to use the *For Each* loop to access the elements of the array

```
        Dim row() As Integer
```

```
        Dim col As Integer
        For Each row In myTable
            For Each col In row
                Console.WriteLine(col)
            Next
            Console.WriteLine()
        Next
```

The code above is very simple and easily understandable. We picked up each *row* (which is an *Integer* array) and then iterated through the *row* while printing each of its columns. The output of the above code will be

```
3
-2
16

1
9
5
6
98

19
6

Press any key to continue
```

In the same way, we can use a three-dimensional array as

```
        Dim myTable(2, 1, 3) As Integer
        myTable(0, 0, 0) = 3
        myTable(1, 1, 1) = 6
```

Or in the jagged array fashion as

```
        Dim myTable(1)()() As Integer
        myTable(0) = New Integer(1)() {}
        myTable(0)(0) = New Integer(2) {}
        myTable(0)(1) = New Integer(3) {}
        myTable(1) = New Integer(2)() {}
        myTable(1)(0) = New Integer(1) {}
        myTable(1)(1) = New Integer(3) {}
        myTable(1)(2) = New Integer(2) {}
        myTable(0)(0)(0) = 34
        myTable(0)(1)(1) = 43
        myTable(1)(2)(2) = 76
```

We have created a three dimensional jagged array. It is an array of two 2-dimensional arrays. The first of the 2-dimensional arrays contain 2 rows. The length of the first row is 3 while the length of the second row is 4. In the similar fashion, the second two dimensional array is also initialized. In the end, we accessed some of the elements of the array and assigned them different values. Although the higher dimensional jagged arrays are quite difficult to perceive; they may be very useful in certain complex problems. Again, the key to avoid confusion with regards multidimensional arrays is to perceive them as 'an array of arrays'.

Some other important points about multidimensional arrays

- In the examples above, we have only used the multidimensional arrays of the integer type. However, you can define the arrays of any data type. For example, you may define an array of a string or even the objects of your own class.

```
Dim names(3) As String
```

- You can initialize the elements of an array on the fly in the following way

```
Dim names() As String = {"Faraz", "Gates", "Hejlsberg", "Gosling"}
```

- You can also separate the declaration and initialization of an array reference and an array as

```
Dim names() As String
   names = New String() {"Faraz", "Gates", "Hejlsberg", "Gosling"}
```

- You can also initialize two-dimensional arrays on the fly (along with declaration)

```
Dim names()() As String = {New String() {"Faraz", "Gates"}, _
                           New String() {"Hejlsberg", "Gosling"}}
```

- Some of the more important properties & methods that can be applied to arrays are

Length gives the number of elements in all dimensions of an array
GetLength(Integer) gives the number of elements in a particular dimension of an array
GetUpperBound() gives the upper bound of the specified dimension of an array
GetLowerBound() gives the lower bound of the specified dimension of an array

The For Each Loop

We have been using the *For Each* loop for quite a long time now. Let us see how it does work and how can we enable our class to be iterated through by the *For Each* loop. For this we need to implement the *IEnumerable* interface which only contains a single method *GetEnumerator()* that returns an object of type *IEnumerator*. The *IEnumerator* interface contains one public property (*Current*) and two public methods *MoveNext()* and *Reset()*. The *Current* property is declared in the *IEnumerator* interface as

```
ReadOnly Property Current As Object
```

It returns the current element of the collection which is of Object data type. The *MoveNext()* method is declared as

```
Function MoveNext() As Boolean
```

It advances the current selection to the next element of the collection and returns *True* if the advancement is successful and returns *False* if the collection has ended. When *MoveNext()* is called for the first time, it sets the selection to the first element in the collection. This means that the Current property is not valid until the *MoveNext()* has been executed for the first time.

Finally the *Reset()* method is declared as

```
        Sub Reset()
```

This method resets the enumerator and sets it to its initial state. After the reset, *MoveNext()* will again advance the selection to the first element in the collection.

We will show you how to make a class that can be iterated using a *For Each* loop by implementing the *IEnumerable* and *IEnumerator* interfaces.

```
Imports System
Imports System.Collections
Module Test
    Public Sub Main()
        Dim list As New MyList()
        Dim name As String
        For Each name In list
            Console.WriteLine(name)
        Next
    End Sub
End Module
Class MyList
    Implements IEnumerable
    Shared names() As String = {"Faraz", "Gates", "Hejlsberg", "Gosling", "Bjarne"}
    Public Function GetEnumerator() As IEnumerator Implements
IEnumerable.GetEnumerator
        Return New MyEnumerator()
    End Function
    Private Class MyEnumerator
        Implements IEnumerator
        Dim index As Integer = -1
        Public ReadOnly Property Current() As Object Implements IEnumerator.Current
            Get
                Return names(index)
            End Get
        End Property
        Public Function MoveNext() As Boolean Implements IEnumerator.MoveNext
            index += 1
            If index >= names.Length Then
                Return False
            Else
                Return True
            End If
        End Function
        Public Sub Reset() Implements IEnumerator.Reset
            index = -1
        End Sub
    End Class
End Class
```

Above, we have declared a class called *MyList* which contains a private nested class named *MyEnumerator*. The class *MyEnumerator* implements the *IEnumerator* by providing the implementation of its public properties and methods. The class *MyList* implements the *IEnumerable* interfaces and returns an object of *MyEnumerator* in its *GetEnumerator()* method. The class *MyList* contains a shared array of string type called *names*. The *MyEnumerator* class iterates through this array. It uses the integer variable *index* to keep track of the current element of the collection. The variable *index* is initialized with -1. Each time the *MoveNext()* method is called, it increments it by 1 and returns whether the collection has been finished or not. The *Current* property returns the *index* element of the collection while the *Reset()* method resets the *index* variable to -1 again.

In the *Test* class we have instantiated the *MyList* class. We iterate through it using a *For Each* loop because we have implemented *IEnumerable* interface on the *MyList* class. The output of the above program is like

```
Faraz
Gates
Hejlsberg
Gosling
Bjarne
Press any key to continue
```

Collections

Although we can make collections of related objects using arrays, there are some limitations while using arrays for collections. The size of an array is always fixed and must be defined at the time of array instantiation. Secondly, an array can only contain objects of the same data type. We also need to define this at the time of its instantiation. Moreover, an array does not impose any particular mechanism for inserting and retrieving the elements of collections. For this purpose, the creators of VB.Net and the .Net Framework Class Library (FCL) have provided a number of classes to serve as a collection for different types. These classes are present in the *System.Collections* namespace. Some of the most common classes from this namespace are

Class	Description
ArrayList	Provides a collection similar to an array but which can grow dynamically as the number of elements change.
Stack	A collection that works on the Last In and First Out (LIFO) principle, i.e., the last item inserted is the first item removed from the collection.
Queue	A collection that works on First In First Out (FIFO) principle, i.e., the first item inserted is the first item removed from the collection.
Hashtable	Provides a collection of key-value pairs that are organized based on the hash code of the key.
SortedList	Provides a collection of key-value pairs where the items are sorted according to the key. The items are accessible by both the keys and the index.

All of the above classes implement the *ICollection* interface which contains three properties and one method.

- The *Count* property returns the number of elements in a collection (similar to the *Length* property of an *Array*)
- The *IsSynchronized* property returns a boolean value representing whether the access to the collection is thread-safe or not
- The SyncRoot property returns an object that can be used to synchronize access to the collection.
- The *CopyTo*(*array As Array, index as Integer*) method copies the elements of the collection to the array starting from a specified index.

All of the collection classes also implement the *IEnumerable* interface so they can be iterated through the *For Each* loop.

The ArrayList class

The *System.Collections.ArrayList* class is similar to arrays but can store elements of any data type. We don't need to specify the size of the collection when using an *ArrayList* (as we used to do in the case of simple arrays). The size of the *ArrayList* grows dynamically as the number of elements it contains, changes. An ArrayList uses an array internally and initializes its size with a default value called Capacity. As the number of elements increase or decrease, *ArrayList* adjusts the capacity of the array accordingly by making a new array and copying the old values to it. The Size of the *ArrayList* is the total number of elements that are actually present in it while the Capacity is the number of elements, the *ArrayList* can hold without instantiating a new array. An *ArrayList* can be constructed as

```
        Dim list As New ArrayList()
```

We can also specify the initial Capacity of the *ArrayList* by passing an integer value in the constructor as

```
        Dim list As New ArrayList(20)
```

We can also create an *ArrayList* with some other collection by passing the collection in the constructor

```
        Dim list As New ArrayList(someCollection)
```

Now, we can add elements to the *ArrayList* by using its *Add()* method. The *Add()* method takes an object of type object as its parameter

```
        list.Add(45)
        list.Add(87)
        list.Add(12)
```

This will add the three numbers to the *ArrayList*. Now, we can iterate through the items in the *ArrayList (list)* using the *For Each* loop as

```
    Public Sub Main()
        Dim list As New ArrayList()
        list.Add(45)
        list.Add(87)
        list.Add(12)
        Dim num As Integer
        For Each num In list
            Console.WriteLine(num)
        Next
    End Sub
```

which will print out the elements in ArrayList as

45

```
87
12
Press any key to continue
```

Author's Note: Java developers take note that we did not cast the integers (implicit data type) to its wrapper before passing to Add() method which expects an instance of type object. The reason for this is that in VB.Net boxing is performed automatically and compiler boxes the value types to object implicitly.

The *ArrayList* class has also implemented the indexer property (or default property) which allow its elements to be accessed using the () operators just like a simple array (We have presented how to implement default properties in the 3rd lesson). The following code is similar to the above code but uses the default property to access the elements of *ArrayList*

```
Public Sub Main()
    Dim list As New ArrayList()
    list.Add(45)
    list.Add(87)
    list.Add(12)
    Dim i As Integer
    For i = 0 To list.Count - 1
        Console.WriteLine(list(i))
    Next
End Sub
```

The output of the code will be similar to the one presented previously. The above code uses the property *Count* to find the current number of elements in the *ArrayList*. Recall that *ArrayList* inherits this property (*Count*) from its parent interface *ICollection*.

A list of some other important properties and methods of the *ArrayList* class is presented in the following table

Property or Method	Description
Capacity	Gets or sets the number of elements the ArrayList can contain
Count	Gets the exact number of elements in the ArrayList
Add(Object)	Adds an element at the end of an ArrayList
Remove(Object)	Removes an element from the ArrayList
RemoveAt(Integer)	Removes an element at the specified index from the ArrayList
Insert(Integer, object)	Inserts an object in the ArrayList at the specified index.
Clear()	Removes all the elements from the ArrayList
Contains(Object)	Returns a boolean value indicating whether the ArrayList contains the supplied element or not
CopyTo()	Copies the elements of the ArrayList to the array supplied as a parameter. The method is overloaded and one can specify the range to be copied and also from which index of the array copy should be started.
IndexOf(Object)	Returns the zero based index of first occurrence of the object in the ArrayList. If the object is not found in the ArrayList, it returns -1
LastIndexOf(Object)	Rerturns the zero based index of the last occurrence of the object in the ArrayList

ToArray()	Returns an array of type object that contains all the elements of the ArrayList
TrimToSize()	Sets the capacity to the actual number of elements in the ArrayList

The Stack class

The *System.Collections.Stack* class is a kind of collections that provides controlled access to its elements. A Stack works on the principle of Last In First Out (LIFO), which means that the last item inserted into the stack will be the first item to be removed from it. Stacks and Queues are very common data structures in computer science and they are implemented in both hardware and software. The insertion of an item into the stack is termed as 'Push' while removing an item from the stack is called a 'Pop'. If the item is not removed but only read from the top of the stack, then this is called 'Peek' operation. The *System.Collections.Stack* class provides the functionality of a Stack in the .Net environment. The Stack class can be instantiated in the similar manner we used for *ArrayList*. The three constructors for *Stack* class are

```
        Dim stack As new Stack()
```

The above (default) constructor will initialize a new empty stack. The following constructor call will initialize the stack with the supplied initial capacity.

```
        Dim stack As new Stack(12)
```

While the following constructor will initialize the Stack with the supplied collection

```
        Dim stack As new Stack(myCollection)
```

Now, we can push elements onto the Stack using the *Push()* method as

```
        stack.Push(2)
        stack.Push(4)
        stack.Push(6)
```

In the similar manner, we can retrieve elements from the stack using the *Pop()* method. The complete program that pushes 3 elements onto the stack and then pops them one by one is presented below

```
    Public Sub Main()
        Dim stack As New Stack()
        stack.Push(2)
        stack.Push(4)
        stack.Push(6)
        While stack.Count <> 0
            Console.WriteLine(stack.Pop())
        End While
    End Sub
```

Note that we have used a *While* loop here to iterate through the elements of the stack. One

thing to remember in the case of stack is that the *Pop()* operation does not only return the element at the top of stack but also removes the top element so the *Count* value will decrease with each *Pop()* operation. The output of the above program is like

```
6
4
2
Press any key to continue
```

The other methods in the *Stack* class are very similar to those of an *ArrayList* except for the *Peek()* method. The *Peek()* method returns the top element of the stack without removing it. The following program demonstrates the use of *Peek()* operation.

```
    Public Sub Main()
        Dim stack As New Stack()
        stack.Push(2)
        stack.Push(4)
        stack.Push(6)
        Console.WriteLine("The total number of elements in stack before Peek() = {0}",
stack.Count)
        Console.WriteLine("The top element of stack is {0}", stack.Peek())
        Console.WriteLine("The total number of elements in stack after Peek() = {0}",
stack.Count)
    End Sub
```

The above program pushes three elements onto the stack and then peek the top element on the stack. The program prints the number of elements in the stack before and after the *Peek()* operation. The result of the program is

```
The total number of elements in stack before Peek() = 3
The top element of stack is 6
The total number of elements in stack after Peek() = 3
Press any key to continue
```

The output of the program shows that *Peek()* does not effect the number of elements in the stack and does not removes the top element contrary to *Pop()* operation.

The Queue class
A Queue works on the principle of First In First Out (FIFO), which means that the first item inserted into a queue will be the first item removed from it. The insertion of an item to a queue is termed as 'Enqueue' while removing an item from a queue is called 'Dequeue'. If an item is not removed but only read from the beginning of the queue, then this is called a 'Peek' operation. The *System.Collections.Queue* class provides the functionality of queues in the .Net environment. A Queue's constructors are similar to those of an *ArrayList* and a *Stack*.

```
        Dim queue As New Queue()  ' an empty queue
        Dim queue As new Queue(16)        ' a queue with initial capacity 16
        Dim queue As new Queue(myCollection) ' a queue containing elements from
myCollection
```

The following program demonstrates the use of Queues in VB.Net

```
    Public Sub Main()
        Dim queue As New Queue()
        queue.Enqueue(2)
        queue.Enqueue(4)
        queue.Enqueue(6)
        While queue.Count <> 0
            Console.WriteLine(queue.Dequeue())
        End While
    End Sub
```

The program enqueues three elements into the Queue and then dequeues them using a while loop. The output of the program is

```
2
4
6
Press any key to continue
```

The output shows that the queue removes the items in the order they were inserted. The other methods of the Queue are very similar to the *ArrayList* and the *Stack* class.

Dictionaries
Dictionaries are a kind of collection that stores items in a key-value pair fashion. Each value in the collection is identified by its key. All the keys in the collection are unique and there can not be more than one key with the same name. This is similar to the English language dictionary like the Oxford Dictionary where each word (key) has its corresponding meaning (value). The two most common types of Dictionaries in the *System.Collections* namespace are the *Hashtable* and the *SortedList* type.

The Hashtable class
The *Hashtable* stores items as key-value pairs. Each item (or value) in the hashtable is uniquely identified by its key. The *Hashtable* stores the key and its value as an object type. Mostly the string class is used as the key in the hashtable. But you can use any other class

as a key. But before selecting a class for the key, make sure to override the *Equals()* and the *GetHashCode()* methods that it inherits from the object class such that

- *Equals()* checks for instance equality rather than the default reference equality.
- *GetHashCode()* returns the same integer for similar instances of the class
- The values returned by *GetHashCode()* are evenly distributed between the *MinValue* and the *MaxValue* of the Integer type

Constructing a Hashtable

The *String* and some of the other classes provided in the Base Class Library do consider these issues and they are very suitable to be used in the hashtable or other dictionaries as a key. There are many constructors available to instantiate a hashtable. Some are

```
        Dim ht As New Hashtable()
```

Which is a default no argument constructor. A hashtable can also be constructed by passing in the initial capacity as

```
        Dim ht As New Hashtable(20)
```

The *Hashtable* class also contains some other constructors which allow you to initialize the hashtable with another collection or dictionary.

Adding items in a Hashtable

Once you have instantiated a hashtable object, you can add items to it using its *Add()* method

```
        ht.Add("st01", "Faraz")
        ht.Add("sci01", "Newton")
        ht.Add("sci02", "Einstein")
```

Retrieving items from the Hashtable

Here we have inserted three items into a hashtable along with their keys. Any particular item can be retrieved using its key

```
        Console.WriteLine("Size of Hashtable is {0}", ht.Count)
        Console.WriteLine("Element with key = st01 is {0}", ht("st01"))
        Console.WriteLine("Size of Hashtable is {0}", ht.Count)
```

We used the default property to retrieve the values from the hashtable. This way of retrieval does not remove the element from the hashtable but just returns the object with the specified key. Therefore, the size before and after the retrieval operation is always same (that is 3 in our case). The output of the code above is

```
Size of Hashtable is 3
Element with key = st01 is Faraz
Size of Hashtable is 3
Press any key to continue
```

Removing a particular item

The elements of the hashtable can be removed by using the *Remove()* method which takes the key of the item to be removed as its argument

```
Public Sub Main()
    Dim ht As New Hashtable()
    ht.Add("st01", "Faraz")
    ht.Add("sci01", "Newton")
    ht.Add("sci02", "Einstein")
    Console.WriteLine("Size of Hashtable is {0}", ht.Count)
    Console.WriteLine("Element with key = st01 is {0}", ht("st01"))
    ht.Remove("st01")
    Console.WriteLine("Size of Hashtable is {0}", ht.Count)
End Sub
```

The output of the program is

```
Size of Hashtable is 3
Removing element with key = st01
Size of Hashtable is 2
Press any key to continue
```

Getting the collection of keys and values

The collection of all the keys and values in a hashtable can be retrieved using the *Keys* and *Values* properties which returns an *ICollection* containing all the keys and values respectively. The following program iterates through all the keys and values and prints these in a *For Each* loop

```
Public Sub Main()
    Dim ht As New Hashtable()
    ht.Add("st01", "Faraz")
    ht.Add("sci01", "Newton")
    ht.Add("sci02", "Einstein")
    Console.WriteLine("Printing Keys...")
    Dim key As String
    For Each key In ht.Keys
        Console.WriteLine(key)
    Next
    Console.WriteLine(vbCrLf & "Printing Values...")
    Dim value As String
    For Each value In ht.Values
        Console.WriteLine(value)
    Next
End Sub
```

The output of the program will be

```
Printing Keys...
st01
sci02
sci01

Printing Values...
Faraz
Einstein
```

Newton
Press any key to continue

Checking the existence of a particular item in the hashtable
You can use the *ContainsKey()* and the *ContainsValue()* methods to find out whether a particular item with the specified key and value exists in the hashtable or not. Both of the methods return a Boolean value.

```
    Public Sub Main()
        Dim ht As New Hashtable()
        ht.Add("st01", "Faraz")
        ht.Add("sci01", "Newton")
        ht.Add("sci02", "Einstein")
        Console.WriteLine(ht.ContainsKey("sci01"))
        Console.WriteLine(ht.ContainsKey("st00"))
        Console.WriteLine(ht.ContainsValue("Einstein"))
    End Sub
```

The program outputs as

True
False
True

Indicating whether the elements in question exist in the dictionary (hashtable) or not.

The SortedList class
The sorted list class is similar to the *Hashtable*, the difference being that the items are sorted according to the key. One of the advantages of using a *SortedList* is that you can get the items in the collection using an integer index just like an array. In the case of a *SortedList*, if you want to use your own class as a key than in addition to the considerations described in the *Hashtable*, you also need to make sure that your class implements the *IComparable* interface.

The *IComparable* interface has only one method

```
    Function CompareTo(ByVal obj as Object) As Integer
```

This method takes the *Object* type argument and returns an integer representing whether the supplied object is equal to, greater than or less than the current object.

- The return value of 0 (zero) indicates that the object is equal to the supplied obj.
- The return value greater than 0 (zero) indicates that this object is greater than the supplied obj.
- While the return value is less than 0 (zero), this indicates that the object is less than the supplied obj.

The *String* class and other primitive data types provide the implementation of this interface and hence can be directly used as a key in the *SortedList*.

The *SortedList* provides similar constructors as provided by the *Hashtable*. The simplest one is a zero argument constructor.

```
        Dim sl As new SortedList()
```

The following table lists some useful properties and methods of the *SortedList* class

Property or Method	Description
Count	Gets the number of elements that the SortedList contains
Keys	Returns an ICollection of all the keys in the SortedList
Values	Returns an ICollection of all the values in the SortedList
Add(key As Object, value As Object)	Adds an element (key-value pair) to the SortedList
GetKey(index As Integer)	Returns the key at the specified index
GetByIndex(index As Integer)	Returns the value at the specified index
IndexOfKey(key As Object)	Returns a zero based index of the specified key
IndexOfValue(value As Object)	Returns a zero based index of the specified value
Remove(key As Object)	Removes an element with the specified key from the SortedList
RemoveAt(Integer)	Removes an element at the specified index from the SortedList
Clear()	Removes all the elements from the SortedList
ContainsKey(key As Object)	Returns a boolean value indicating whether the SortedList contains an element with the supplied key
ContainsValue(value As Object)	Returns a boolean value indicating whether the SortedList contains an element with the supplied value

The following code demonstrates the use of *SortedList*

```
    Public Sub Main()
        Dim sl As New SortedList()
        sl.Add(32, "Java")
        sl.Add(21, "C#")
        sl.Add(7, "VB.Net")
        sl.Add(49, "C++")
        Console.WriteLine("The items in the sorted order are...")
        Console.WriteLine("   Key     Value")
        Console.WriteLine("   ===     =====")
        Dim i As Integer
        For i = 0 To sl.Count - 1
            Console.WriteLine("   {0}       {1}", sl.GetKey(i), sl.GetByIndex(i))
        Next
    End Sub
```

This code stores the names of different programming languages (in string form) using integer keys. Then the *For* loop is used to retrieve the keys and values contained in the *SortedList (sl)*. Since this is a sorted list, the items are internally stored in a sorted order and when we retrieve these names by the *GetKey()* or the *GetByIndex()* method, we get a sorted list of items. The output of the executed code will be

```
The items in the sorted order are...
Key Value
=== =====
```

7 VB.Net
21 C#
32 Java
49 C++
Press any key to continue

String Handling in VB.Net The next topic of today's lesson is about handling strings in the VB.Net programming language. In VB.Net, a *String* is a built and a primitive data type. The primitive data type *string* maps to the *System.String* class. The objects of the *String* class are immutable by nature. By immutable, this means that the state of the object can not be changed by any operation. This is the reason why when we call the *ToUpper()* method on *String*. It doesn't change the original string but creates and returns a new string object that is the upper case representation of the original object. The mutable version of a string in the .Net Platform is the *System.Text.StringBuilder* class. The objects of this class are mutable, that is, their state can be changed by their operations. Hence, if you call the *Append()* method on the *StringBuilder* class object, the new string will be appended (added in the end of) the original object. Let's now discuss the two classes one by one.

String class and its members

We have been using the *String* class since our first lesson in the VB.Net school. We have also seen some of its properties (like *Length*) and methods (like *Equals()*) in the previous lessons. Now we will describe some of the common properties and methods of the *String* class and then demonstrate their use in code.

Property or Method	Description
Length	Gets the number of characters the String object contains
CompareTo(s As String)	Compares this instance of the string with the supplied string s and returns an integer on the pattern of the IComparable interface
Compare(s1 As String, s2 As String)	This is a shared method and compares the two supplied strings on the pattern of the IComparable interface
Equals(s As String)	Returns true if the supplied string is exactly the same to this string else returns false
Concat(s As String)	Returns a new string that is the concatenation (addition) of this string and the supplied string s.
Insert(index As Integer, s As String)	Returns a new string by inserting the supplied string s at the specified index of this string
Copy(s As String)	This shared method returns the a new string that is a copy of the supplied string
Intern(s As String)	This shared method returns the system's reference to the supplied string
StartsWith(s As String)	Returns true if this string starts with the supplied string s
EndsWith(s As String)	Returns true if this string ends with the supplied string s
IndexOf(s As String)IndexOf(ch As Char)	Returns the zero based index of the first occurrence of the supplied string s or supplied character ch. The method is overloaded and more versions are available.
LastIndexOf(s As String)LastIndexOf(ch As Char)	Returns the zero based index of the last occurrence of the supplied string s or supplied character ch. The method is overloaded and more versions are available.
Replace(char, char)Replace(string, string)	Returns a new string by replacing all occurrences of the first char with the second char (or first string with the second string)
Split(params As Char())	Identifies the substrings in this instance that are delimited by one or more characters specified in an array, then places the substrings into a String array and returns it.
Substring(i1 As Integer)Substring(i2 As Integer, i3 As Integer)	Retrieves a substring of this string starting from the index position i1 till end in the first overloaded form. In the second overloaded form, it retrieves the substring starting from the index i2 and has a length of i3 characters.
ToCharArray()	Returns an array of characters of this string
ToUpper()	Returns a copy of this string in uppercase
ToLower()	Returns a copy of this string in lowercase
Trim()	Returns a new string by removing all occurrences of a set of specified characters from the beginning and at the end of this instance.
TrimStart()	Returns a new string by removing all occurrences of a set of specified characters from the beginning of this instance.
TrimEnd()	Returns a new string by removing all occurrences of a set of specified characters from the end of this instance.

In the following code we have demonstrated the use of most of the above methods of the string class. The code is quite self-explanatory and only includes the method calls and their results.

```
    Public Sub Main()
        Dim s1 As String = "faraz"
        Dim s2 As String = "fraz"
        Dim s3 As String = "Faraz"
        Dim s4 As String = "VB.Net is a great programming language!"
        Dim s5 As String = "    This is the target text   "
        Console.WriteLine("Length of {0} is {1}", s1, s1.Length)
        Console.WriteLine("Comparision result for {0} with {1} is {2}", s1, s2,
s1.CompareTo(s2))
        Console.WriteLine("Equality checking of {0} with {1} returns {2}", s1, s3,
s1.Equals(s3))
        Console.WriteLine("Equality checking of {0} with lowercase {1} ({2}) returns
{3}", _
                s1, s3, s3.ToLower(), s1.Equals(s3.ToLower()))
        Console.WriteLine("The index of a in {0} is {1}", s3, s3.IndexOf("a"))
        Console.WriteLine("The last index of a in {0} is {1}", s3,
s3.LastIndexOf("a"))
        Console.WriteLine("The individual words of '{0}' are", s4)
        Dim words() As String = s4.Split(" "c)
        Dim word As String
        For Each word In words
            Console.WriteLine("   {0}", word)
        Next
        Console.WriteLine(vbCrLf & "The substring of " & vbCrLf & "  '{0}' " & _
                vbCrLf & "from index 3 of length 10 is " & vbCrLf & "  '{1}'", _
                s4, s4.Substring(3, 10))
        Console.WriteLine(vbCrLf & "The string " & vbCrLf & "   '{0}'" & vbCrLf & _
                "after trimming is " & vbCrLf & "    '{1}'", s5, s5.Trim())
    End Sub
```

The output of the program will certainly help you understand the behavior of each member.

```
Length of faraz is 5
Comparision result for faraz with fraz is -1
Equality checking of faraz with Faraz returns False
Equality checking of faraz with lowercase Faraz (faraz) returns True
The index of a in Faraz is 1
The last index of a in Faraz is 3
The individual words of 'VB.Net is a great programming language!' are
VB.Net
is
a
great
programming
language!

The substring of
'VB.Net is a great programming language!'
from index 3 of length 10 is
```

```
'Net is a g'

The string
' This is the target text '
after trimming is
'This is the target text'
Press any key to continue
```

The StringBuilder class

The *System.Text.StringBuilder* class is very similar to the *System.String* class with the difference being that it is mutable, that is, the internal state of its objects can be modified by its operations. Unlike the string class, you must first call the constructor of *StringBuilder* to instantiate its object

```
        Dim s As String = "This is held by string"
        Dim sb As New StringBuilder("This is held by StringBuilder")
```

StringBuilder is somewhat similar to the *ArrayList* and other collections in the way that it grows automatically as the size of the string it contains changes. Hence, the Capacity of the *StringBuilder* may be different from its *Length*. Some of the more common properties and methods of the *StringBuilder* class are listed in the following table

Property or Method	Description
Length	Gets the number of characters that the StringBuilder object contains
Capacity	Gets the current capacity of the StringBuilder object
Append()	Appends the string representation of the specified object at the end of this StringBuilder instance. The method has a number of overloaded forms
Insert()	Inserts the string representation of the specified object at the specified index of this StringBuilder object
Replace(char, char)Replace(string, string)	Replaces all occurrences of the first supplied character (or string) with the second supplied character (or string) in this StringBuilder object
Remove(st as Integer, length as Integer)	Removes all characters from the index position st of the specified length in the current StringBuilder object
Equals(StringBuilder)	Checks the supplied StringBuilder object with this instance and returns true if both are identical (or same) else returns false

The following code demonstrates the use of some of these methods

```
Imports System
Imports System.Text
Module Test
    Public Sub Main()
        Dim sb As New StringBuilder("The text")
        Dim s As String = " is complete"
        Console.WriteLine("Length of StringBuilder '{0}' is {1}", sb, sb.Length)
        Console.WriteLine("Capacity of StringBuilder '{0}' is {1}", sb, sb.Capacity)
        Console.WriteLine(vbCrLf & "StringBuilder before appending is `{0}'", sb)
        Console.WriteLine("StringBuilder after appending `{0}' is `{1}'", s,
sb.Append(s))
        Console.WriteLine(vbCrLf & "StringBuilder after inserting `now' is `{0}'",
sb.Insert(11, " now"))
        Console.WriteLine(vbCrLf & "StringBuilder after removing 'is ' is `{0}'",
sb.Remove(8, 3))
        Console.WriteLine(vbCrLf & "StringBuilder after replacing all `e' with `x' is
{0}", sb.Replace("e"c, "x"c))
    End Sub
End Module
```

And the output of the executed code is

```
Length of StringBuilder `The text' is 8
Capacity of StringBuilder `The text' is 16

StringBuilder before appending is `The text'
StringBuilder after appending ` is complete' is `The text is complete'

StringBuilder after inserting `now' is `The text is now complete'

StringBuilder after removing 'is ' is `The text now complete'

StringBuilder after replacing all `e' with `x' is Thx txxt now complxtx
Press any key to continue
```

Note that in all cases, the original object of the *StringBuilder* is being modified hence the *StringBuilder* objects are mutable compared to the immutable *String* objects.

Food for thought: Exercise 8

1. What is the difference between a rectangular and a jagged array?
2. Can we create a rectangular array without using (,) notation and using the jagged array notation only? Write some code to demonstrate this.
3. What is the difference between arrays and collections?
4. Write your own class for a *Stack* using the object array internally. (You don't need to worry about thread safety and other issues. Your class should only contain *Push(), Pop() and Peek()* methods and the *Count* property)
5. How are Dictionaries different from simple collections like *ArrayList, Stack and a Queue*?
6. In this text, we did not describe which algorithm the Hashtable uses to maintain the collection internally. Search for the text on the internet or in data structure's books and write a short description of this algorithm.
7. Why it is advised to properly override the *Equals()* and the *GetHashCode()* methods of a class that is to be used as a key in Hashtable?
8. What is the basic difference between the *String* and the *StringBuilder* class? What is the difference between the *Intern()* and the *Copy()* methods of the String class?

Solution of Last Issue's Exercise (Exercise 7)

1. How are abstract (MustInherit) classes different from normal classes and interfaces?

Abstract (*MustInherit*) classes are incomplete classes. They contain methods that are to be implemented by subclasses. Unlike normal classes, abstract classes can not be instantiated. A class inheriting an abstract class must provide the body of the abstract methods declared in the abstract class. Since the abstract class is made to be inherited, it can not be declared NotInheritable. An abstract class, unlike an interface, may contain the body of some of the methods and may also contain constructors.

2. Write an abstract class that declares a read only abstract property.

```
MustInherit Class MyAbstract
    Public MustOverride ReadOnly Property PI() As Double
End Class
```

Here, PI is a read-only abstract (*MustOverride*) property defined in the abstract class *MyAbstract*. The data type of the property is a *Double* and we have made it read-only by using the *ReadOnly* keyword. We made the property abstract by marking it with the *MustOverride* keyword. Any class that wishes to subclass the *MyAbstract* class must provide the definition of property PI with only the *Get* block.

3. Abstract classes may contain constructors. When we can't instantiate the abstract classes, what is the use of these constructors?

A constructor of a class is not only called when its object is instantiated; it also gets called when an object of its subclass is created. An abstract class may contain a constructor to be called when it's subclass is instantiated.

4. Interfaces are also called specification. Why?

This is because interfaces define the contract to be followed by all implementing classes. They are also called specification. The Interface specifies which services its subclasses must provide and what the format of these services should be.

5. We have seen in our programs that sub-classes can call the methods defined in a base-class. Can you make a program where a base-class calls a method defined in a sub-class? (Tip: Try using abstract class)

Consider the following code

```
Imports System
Module Test
    Public Sub Main()
        ' line 1 and line 2 will both do the same work
        'Dim cal As new Callee()  ' line 1
        Dim cal As Caller = New Callee() ' line 2
        cal.CallMethod()
    End Sub
End Module
MustInherit Class Caller
    Public Sub CallMethod()
        Console.WriteLine("In CallMethod() of Caller class, about to call DoWork()
defined in sub-class")
        DoWork()
    End Sub
    Public MustOverride Sub DoWork()
End Class
Class Callee
    Inherits Caller
    Public Overrides Sub DoWork()
        Console.WriteLine("DoWork() of class Callee called...")
    End Sub
End Class
```

Above, the CallMethod() method of the Caller class is calling the method DoWork() declared in the Caller class but defined in the Callee class. Hence, a method (CallMethod()) in the base-class (Caller) is calling a method (DoWork()) defined in its subclass (Callee).

What's Next...

Next time, we will be discussing exception handling mechanism in VB.Net. We will explore

- Exceptions basics
- Exceptions in .Net Framework
- Try…Catch...Finally blocks
- Throwing Exceptions
- Your Own Custom Exceptions

Exception Handling in VB.Net

Lesson Plan

Shortly we will explore the exception handling mechanism in VB.Net. We will start by looking at the idea behind exceptions and how different exceptions are caught. Later we will explore the different features provided and the constraints applied by the VB.Net programming language in exception handling. Finally we will learn how to define our own custom exceptions.

Exceptions Basics

Exception handling is a mechanism to manage run-time errors in .NET that would otherwise cause your software to terminate abruptly.

The need for Exceptions

Consider the following simple code

```
Imports System
Module Test
    Public Sub Main()
        Dim p As Integer = ConvertToInteger("34")
        Console.WriteLine(p)
    End Sub
    Public Function ConvertToInteger(ByVal str As String) As Integer
        Dim i As Integer = Integer.Parse(str)
        Return i
    End Function
End Module
```

The *ConvertToInteger()* method returns the integer present in the string type. The output of this code will be

```
34
```

But what if the method *ConvertToInteger()* is called in the *Main()* method as

```
        Dim p As Integer = ConvertToInteger("My name")
```

There won't be a compile time error as the parameter 'My name' is in the required form of a string. However when the code is executed, the processor will attempt to convert the string 'My name' to an Integer which is offcourse not possible. The result, the program will crash! What should we do now?

Exceptions in VB.Net and .Net Framework

People have worked on the problem of managing errors and have developed a solution in

the form of 'Exceptions'. Programmers may define and throw an exception in the case of unexpected events. Below are some important points about exceptions and the exception handling mechanism in VB.Net and the .Net Framework:

- All exceptions in .Net are objects.
- The *System.Exception* class is the base class for all the exceptions in .Net
- Any method can raise or throw an exception in the case of some unexpected event during its execution using the throw keyword in VB.Net
- The thrown exception can be caught or dealt within the calling method using the *Try...Catch* block.
- The code that may throw an exception which we want to handle is put in the *Try* block. This is called an attempt to catch an exception.
- The code to handle the thrown exception is placed in the Catch block just after the *Try* block. This is called catching an exception. We can define which particular class of exception we want to deal in this *Catch* block by mentioning the name of the exception with the *Catch* keyword
- Multiple Catch blocks can be defined for a single *Try* block where each *Catch* block will Catch a particular class of exception.
- The code that must always be executed after the *Try* or *Try...Catch* block is placed in the *Finally* block. This is just after the *Try* or the *Try...Catch* block. This code is guaranteed to always be executed whether the exception occurs or not.
- When an exception is raised during the execution of code inside the *Try* block, the remaining code in the *Try* block is neglected and the control of execution is transferred to the respective *Catch* or *Finally* block.
- Since exceptions are present in .Net as classes and objects they follow the inheritance hierarchy. This means that if you write a *Catch* block to handle a base class exception, it will automatically handle all of its sub-class exceptions. Attempting to Catch any of the sub-class exceptions explicitly after the parent class exception will make render your code unreachable or useless.
- The *Finally* block is optional. Exception handling requires any combination of the *Try...Catch* or *Try...Catch...Finally* or *Try...Finally* blocks.
- If you do not Catch an exception the runtime environment (Common Language Runtime or CLR) will Catch it on your behalf and may cause your program to be terminated.

Author's Note: For VB6 developers, Try...Catch is .Net's replacement for On Error and is much more structured as you will see in this lesson. For Java developers, there is no concept of checked exceptions in VB.Net. All the exceptions are implicitly unchecked. Hence there is no throws keyword present in VB.Net. The absence of checked exceptions in VB.Net is quite a hot topic of debate since the birth of .Net. I personally feel that the idea of checked exceptions is extremely good and it should have been implemented in VB.Net. See the Microsoft web site regards the argument of Hejlsberg and other designers of C# and why they haven't included checked exceptions in .Net
http://msdn.microsoft.com/chats/vstudio/vstudio_032103.asp

Handling Exceptions using the Try...Catch...Finally blocks

Use of the Try...Catch block

A simple demonstration for the use of *Try...Catch* block is given below

```
Public Sub Main()
    Dim s As String = Nothing
    Try
        Console.WriteLine("In Try block... before calling s.ToLower()")
        Console.WriteLine(s.ToLower())
        Console.WriteLine("In Try block... after calling s.ToLower()")
    Catch e As NullReferenceException
        Console.WriteLine("In Catch block...")
        Console.WriteLine("NullReferenceException Caught")
    End Try
    Console.WriteLine("After Try...Catch block")
End Sub
```

The string 's' in the *Main()* method is assigned a Nothing value. If we attempt to call the *ToLower()* method with this null reference in the *Console.WriteLine()* method, the CLR (Common Language Runtime) will raise the *NullReferenceException*. Since we have enclosed the call to the *ToLower()* method in a *Try* block, the Runtime will search for a *Catch* block which can Catch this exception and, if one is found, the execution will jump to this *Catch* block. The syntax of the *Catch* block is important to understand. After the *Catch*, a reference ('e' in our case) of our target exception class is declared (*NullReferenceException* in our case). When the above program is executed, the outputs is

```
In Try block... before calling s.ToLower()
In Catch block...
NullReferenceException Caught
After Try...Catch block
Press any key to continue
```

Carefully look at the output of the program and compare it with the source code. The call to *s.ToLower()* raised the *NullReferenceException*. As a result the execution of the remaining part of the *Try* block is ignored and the program execution is transferred to the *Catch* block. Remember that the *NullReferenceException* is raised when we attempt to access the members of a class using a null reference (*Nothing*). Lets change the code above a little and assign an object to the reference 's'

```
Public Sub Main()
    Dim s As String = "Faraz"
    Try
        Console.WriteLine("In Try block... before calling s.ToLower()")
        Console.WriteLine(s.ToLower())
        Console.WriteLine("In Try block... after calling s.ToLower()")
    Catch e As NullReferenceException
        Console.WriteLine("In Catch block...")
        Console.WriteLine("NullReferenceException Caught")
    End Try
    Console.WriteLine("After Try...Catch block")
End Sub
```

Since this code does not cause any exceptions to be raised, the execution of program will output as

```
In Try block... before calling s.ToLower()
faraz
In Try block... after calling s.ToLower()
After Try...Catch block
Press any key to continue
```

Exception class' Message and StackTrace Property
Note that the code under the *Catch* block didn't get executed because of the absence of an *NullReferenceException*. Now lets remove the *Try...Catch* block and see what happens

```
Public Sub Main()
    Dim s As String = Nothing
    Console.WriteLine("Before printing lower case string...")
    Console.WriteLine(s.ToLower())
    Console.WriteLine("After printing lower case string...")
End Sub
```

When we compile and execute the above code, we see the following output

```
Before printing lower case string...

Unhandled Exception: System.NullReferenceException: Object reference not set to
an instance of an object.
at VBDotNetSchoolLesson9.Test.Main() in C:\Documents and Settings\Administrat
or\My Documents\Visual Studio Projects\VBDotNetSchoolLesson4\Module1.vb:line 7
Press any key to continue
```

Since we did not Catch the *NullReferenceException*, our program got terminated prematurely with the runtime (CLR) reporting two things 1. An Exception Message: which describes the exception
2. A Stack Trace of the cause of execution: This is the hierarchy of function calls which caused the exception. In our case it shows that the exception is caused by the *Main()* method in the Test module which is contained in the *VBDotNetSchoolLesson9* namespace (*VBDotNetSchoolLesson9.Test.Main()*). The Stack trace also points out the file name along with its complete path and the line number which contained the cause of the exception.

Because we don't want our program to crash when an exception occurs, we attempt to catch all the exceptions that can be caused by our code. Let's move to our previous code segment where we caught the *NullReferenceException* in the *Catch* block. We can also print the *Message* and *Stack Trace* of the exception using the *Message* and the *StackTrace* properties of the *Exception* class. Examine the following code

```
Public Sub Main()
    Dim s As String = Nothing
    Try
        Console.WriteLine("In Try block... before calling s.ToLower()")
        Console.WriteLine(s.ToLower())
        Console.WriteLine("In Try block... after calling s.ToLower()")
    Catch e As NullReferenceException
        Console.WriteLine(vbCrLf & "In Catch block...")
        Console.WriteLine("NullReferenceException Caught")
        Console.WriteLine(vbCrLf & "Exception Message")
        Console.WriteLine("=================" & vbCrLf)
        Console.WriteLine(e.Message)
        Console.WriteLine(vbCrLf & "Exception Stack Trace")
        Console.WriteLine("====================" & vbCrLf)
        Console.WriteLine(e.StackTrace)
    End Try
    Console.WriteLine(vbCrLf & "After Try...Catch block")
```

```
    End Sub
```

The difference between the above and the previous code segment is that here we have printed the explanatory message and stack trace of the exception explicitly by use of the exception reference. The output of the program will be

```
In Try block... before calling s.ToLower()

In Catch block...
NullReferenceException Caught

Exception Message
==================

Object reference not set to an instance of an object.

Exception Stack Trace
=====================

at VBDotNetSchoolLesson9.Test.Main() in C:\Documents and Settings\Administrat
or\My Documents\Visual Studio Projects\VBDotNetSchoolLesson4\Module1.vb:line 9

After Try...Catch block
Press any key to continue
```

The Finally block

The optional *Finally* block comes just after the *Try* or *Catch* block. The code in the *Finally* block is guaranteed to always be executed whether an exception occurs or not in the *Try* block. Usually the *Finally* block is used to free any resources acquired within the *Try* block and hence could not be closed because of the exception. For example the *Finally* block can be used to close a file, database, socket connection or other important resources opened in the *Try* block. Let's add the *Finally* block to our previous code example.

```
    Public Sub Main()
        Dim s As String = "Faraz"
        Try
            Console.WriteLine("In Try block... before calling s.ToLower()")
            Console.WriteLine(s.ToLower())
            Console.WriteLine("In Try block... after calling s.ToLower()")
        Catch e As NullReferenceException
            Console.WriteLine(vbCrLf & "In Catch block...")
            Console.WriteLine("NullReferenceException Caught")
        Finally
            Console.WriteLine(vbCrLf & "In Finally block...")
        End Try
    End Sub
```

When we execute the program we see the following output

```
In Try block... before calling s.ToLower()
Faraz
In Try block... after calling s.ToLower()
```

```
In Finally block...
Press any key to continue
```

Since no exception is raised, the code in the *Finally* block is executed just after the code in the *Try* block. Let's cause an exception to occur by setting the string 's' to *Nothing* in the *Main()* method

```
    Public Sub Main()
        Dim s As String = Nothing
           ...
    End Sub
```

Now the output will be

```
In Try block... before calling s.ToLower()

In Catch block...
NullReferenceException Caught

In Finally block...
Press any key to continue
```

The output shows that the code in the *Finally* block is always executed after the execution of the *Try* and *Catch* block. This is regardless if an exception occurred or not.

It is possible to write the *Try...Finally* block without using a *Catch* block

```
    Public Sub Main()
        Dim s As String = "Faraz"
        Try
            Console.WriteLine("In Try block... before calling s.ToLower()")
            Console.WriteLine(s.ToLower())
            Console.WriteLine("In Try block... after calling s.ToLower()")
        Finally
            Console.WriteLine(vbCrLf & "In Finally block...")
        End Try
    End Sub
```

The output of the program is

```
In Try block... before calling s.ToLower()
Faraz
In Try block... after calling s.ToLower()

In Finally block...
Press any key to continue
```

The output of the program shows that the *Finally* block is always executed after the *Try* block.

Catching Multiple Exceptions using multiple Catch blocks

It is possible to catch multiple (different) exceptions that may be raised in a *Try* block using multiple (or a series of) *catch* blocks. For example we can write three *Catch* blocks; one for catching a *NullReferenceException*, the second for catching an *IndexOutOfRangeException* and the third is for any other exception (*Exception*). Remember that the *IndexOutOfRangeException* is raised when an element of an array whose index is out of the range of the array is accessed. An out of range index can be either less than zero or greater than or equal to the size of the array. The code below demonstrates the use of multiple *Catch* blocks

```
Public Sub Main()
    Dim s As String = "Faraz"
    Dim i(2) As Integer
    Try
        Console.WriteLine("Entering the Try block..." & vbCrLf)
        ' can cause NullReferenceException
        Console.WriteLine("Lower case name is: " + s.ToLower())
        ' can cause NullReferenceException or IndexOutOfRangeException
        Console.WriteLine("First element of array is: " + i(0).ToString())
        ' can cause DivideByZeroException
        i(0) = 3
        i(1) = CType(4 / i(0), Integer)
        Console.WriteLine(vbCrLf & "Leaving the Try block...")
    Catch e As NullReferenceException
        Console.WriteLine(vbCrLf & "In Catch block...")
        Console.WriteLine("NullReferenceException Caught")
    Catch e As IndexOutOfRangeException
        Console.WriteLine(vbCrLf & "In Catch block...")
        Console.WriteLine("IndexOutOfRangeException Caught")
    Catch e As Exception
        Console.WriteLine(vbCrLf & "In Catch block...")
        Console.WriteLine("Exception Caught")
        Console.WriteLine(e.Message)
    End Try
End Sub
```

Here we have used the string 's' and an integer array 'i'. The size of the Integer array is declared as 3. There are three places in the program where we will introduce the occurrence of the exceptions.

- First 's' and 'i' can be *Nothing* or *Null* causing the *NullReferenceException*.
- Secondly, the access to the array 'i' may cause an *IndexOutOfRangeException*
- Finally the division of 4 by i(0) may cause an *DivideByZeroException* if the value of i(0) is zero.

We have declared three *Catch* blocks in the code for each of these types of exception. Note that the last *Catch* block is designed to catch any other exception except *NullReferenceException* and *IndexOutOfRangeException* which have already been caught above. Only one of these exceptions can be raised, which will terminate the execution of the *Try* block and will transfer the execution to the respective *Catch* block.

When the above code is executed we will see the following output

```
Entering the Try block...

Lower case name is: Faraz
First element of array is: 0

Leaving the Try block...
Press any key to continue
```

So far so good, no exception occurred. Let's first make the string reference 's' pointing to *Nothing* and see the effect

```
    Public Sub Main()
        Dim s As String = Nothing
          ...
    End Sub
```

The output will be

```
Entering the Try block...


In Catch block...
NullReferenceException Caught
Press any key to continue
```

It looks similar and very much as expected. Now change the array index to an out of bounds value (either less than zero or greater than or equal to 3). Also change the string 's' to point to another string to avoid the *NullReferenceException*

```
        …
        Console.WriteLine("Sixth element of array is: " + i(5).ToString())
        …
```

The output will expectedly be

```
Entering the Try block...

Lower case name is: Faraz

In Catch block...
IndexOutOfRangeException Caught
Press any key to continue
```

Finally correct the access to the array using a valid index and make the value of i(0) equal to zero to cause the *DivideByZeroException*

```
        ...
        i(0) = 0
        ...
```

The output of the program will be

```
Entering the Try block...

Lower case name is: faraz
First element of array is: 0

In Catch block...
Exception Caught
Attempted to divide by zero.
Press any key to continue
```

The execution of 3/i(0) will cause the *DivideByZeroException*. The runtime checked for the presence of the *Catch* block. It found in the third *Catch* block the Exception which is the super type of *DivideByZeroException* (and any other exception in .Net). The execution is then transferred to the corresponding *Catch* block.

An important point to remember when using multiple Catch blocks
Since exceptions are present in .Net as classes and objects, they follow the inheritance hierarchy. This means that if you write a *Catch* block to handle a base class exception, it will automatically handle all of its sub-class exceptions. Attempting to catch any of the sub-class exceptions explicitly after the parent class exception, will cause this code to become unreachable, dead, useless code. For example, review the following code

```
Public Sub Main()
    Dim s As String = Nothing
    Try
        Console.WriteLine("Entering the Try block..." & vbCrLf)
        ' can cause NullReferenceException
        Console.WriteLine("Lower case name is: " + s.ToLower())
        Console.WriteLine(vbCrLf & "Leaving the Try block...")
    Catch e As Exception
        Console.WriteLine(vbCrLf & "In Catch block...")
        Console.WriteLine("Exception Caught")
        Console.WriteLine(e.Message)
    Catch e As NullReferenceException
        ''''''' Unreacable code
        Console.WriteLine(vbCrLf & "In Catch block...")
        Console.WriteLine("NullReferenceException Caught")
    End Try
End Sub
```

Above, the *Catch* block, which is to handle the Exception type exception is placed prior to its sub-class exception (*NullReferenceException*). No matter what exception is raised in the *Try* block, it will always be caught by the first *Catch* block, making the second *Catch* block "a dead piece of code". Hence, one must consider the inheritance hierarchy of different exceptions while using multiple *Catch* blocks.

Author's Note: Unfortunately the VB.Net compiler does not detect the unreachable or dead code and does not even produce a warning. While defining unreachable code concludes a compile time error in C#.

Other important points about Exception Handling in VB.Net

- It is permissible to only write the *Catch* keyword without defining an identifier and exception. This is similar to catching an *Exception* type exception or catching any exception as

```
Catch
        ...
```

We would strongly suggest not to use it, rather use the *Exception* class to catch all the exceptions as

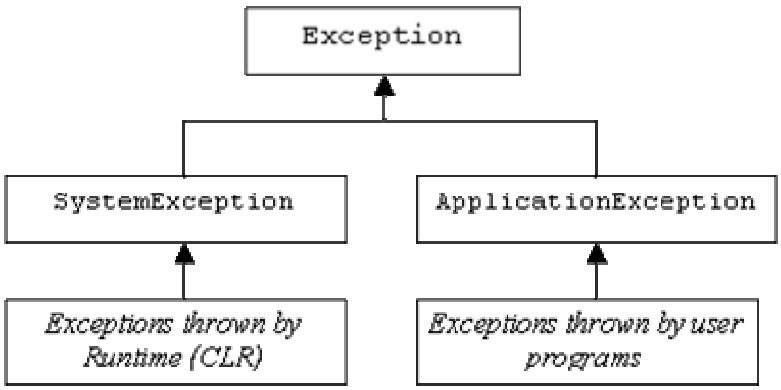
```
Catch e As Exception
        ...
```

Defining your own custom exceptions
It is possible in VB.Net to define your own custom exceptions to identify the occurrence of unexpected events in your code. As stated earlier, exceptions are implemented in VB.Net as classes and objects. So in order to define a new custom exception, one must define a new class for this. But before understanding how we can define our own exceptions, it is important to understand the inheritance hierarchy of basic exceptions in the .Net framework.

Exception Hierarchy in .Net Framework
There are two types of exception in .Net.

- Exceptions generated by the runtime (Common Language Runtime) are called System Exceptions
- Exceptions generated by user programs are called Application Exceptions.

The simple hierarchy of exceptions in the .NET framework is shown in the following diagram.



Exceptions Hierarchy in .net Framework

All user defined exceptions should be derived from the ApplicationException class. In the following code we will demonstrate how to define our own custom exception named *InvalidArgumentException*. We will use this exception in our *Divide()* method when the second argument is zero. The code to define a custom exception *InvalidArgumentException* is

```
Class InvalidArgumentException
    Inherits ApplicationException
    Public Sub New()
        MyBase.New("Divide By Zero Error")
    End Sub
    Public Sub New(ByVal message As String)
        MyBase.New(message)
```

```
    End Sub
End Class
```

Our custom exception class above *InvalidArgumentException* has been derived from *ApplicationException* (the base of all user defined exceptions). In the body of the class we have defined only two constructors. One takes no arguments while the other takes a string 'message' as an argument. Both of the constructors pass this message to the base class constructor which initializes the *Message* property (originally present in *Exception* class) with the supplied string.

We have defined our own custom exception, it's now the time to use it.

Throwing an exception : the Throw keyword
A method can throw an exception using the *Throw* keyword. We will now demonstrate how the *Divide()* method can throw the *InvalidArgumentException* when the second argument is zero.

```
    Function Divide(ByVal a As Double, ByVal b As Double) As Double
        If b = 0 Then
            Throw New InvalidArgumentException()
        End If
        Dim c As Double = a / b
        Return c
    End Function
```

The *Divide()* method will Throw the *InvalidArgumentException* when the second argument is found to be zero. Note that the *Divide()* method creates and throws a new object of type *InvalidArgumentException*. Alternatively, it can also define its own message when throwing an exception like below

```
Throw New InvalidArgumentException("Error: The second argument of Divide is found
zero")
```

The *Main()* method will use this method and catch the exception using the *Try...Catch* block.

```
    Public Sub Main()
        Try
            Console.WriteLine("In Try block...")
            Dim d As Double = Divide(3, 0)
            Console.WriteLine("\tResult of division: {0}", d)
        Catch e As InvalidArgumentException
            Console.WriteLine(vbCrLf & "In Catch block...")
            Console.WriteLine("   System.InvalidArgumentException caught...")
            Console.WriteLine("   Message: " + e.Message)
        End Try
```

This code is very similar to the earlier examples except that now we are catching our specific pre defined exception. When this code is executed, the following output is generated on the console

```
In Try block...
```

```
In Catch block...
System.InvalidArgumentException caught...
Message: Divide By Zero Error
Press any key to continue
```

Note that the *Message* Property prints our own custom message.

Some important points about defining your own exceptions

- It is a convention in the .NET framework that the names of all exceptions end with the word '*Exception*' like *SystemException*, *NullReferenceException*, *IndexOutOfRangeException* etc. We would strongly recommend following this naming convention
- Always derive your custom exceptions from the *ApplicationException* class
- Catching and throwing exceptions has some performance overhead, so it is not good programming practice to throw unnecessary exceptions.

Food for thought: Exercise 9

1. What is the difference between an error and an exception?
2. Why do we need exceptions?
3. The Finally block is generally placed after the Try...Catch block. Why it is used? What is the difference between the regular code after the Try...Catch block and code in the Finally block

```
    Public Sub Main()
        Try
            ' some code
        Catch e As InvalidArgumentException
            ' some code
        Finally
            ' code in Finally block
        End Try
        ' some regular code
    End Sub
```

4. Why should we define our own custom exceptions? What will be the difference if instead of throwing a custom InvalidArgumentException we throw an Exception class object in the Divide() method like

```
Throw New Exception("The second argument of Divide is found zero")
```

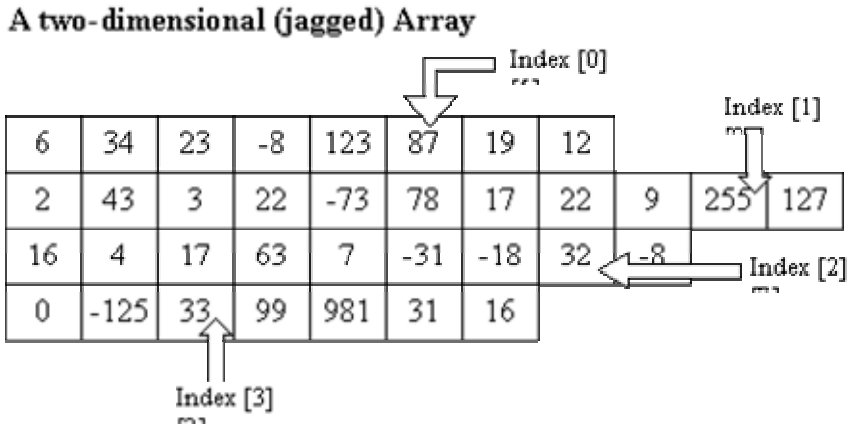
5. Why do we need multiple Catch blocks when we can Catch all the exception by the base

class exception (Exception)?

Solution of Last Issue's Exercise (Exercise 8)

1. What is the difference between a rectangular and a jagged array?

In a rectangular array, each dimension contains an equal number of elements while in a jagged array, not all dimensions of the array contain an equal number of elements. A Rectangular array can be accessed with the new VB.Net syntax (,) like myArray(2, 3) while accessing jagged arrays, you have to use the traditional multiple () pairs. Such as myArray(1)(4)



2. Can we create a rectangular array without using (,) notation and using the jagged array notation only? Write code to demonstrate this.

Yes, rectangular arrays can be created and accessed with jagged array notation. The following code demonstrates this

```
Imports System
Module Test
    Public Sub Main()
        ' Declaring and Defining Array
        Dim myArray(3)() As Integer
        Dim i, j As Integer
        For i = 0 To myArray.Length - 1
            myArray(i) = New Integer(3) {}
        Next
        ' Filling the array
        Dim rand As New Random()
        For i = 0 To myArray.Length - 1
            For j = 0 To myArray(i).Length - 1
                myArray(i)(j) = rand.Next(30)
            Next
        Next
        ' Printing the elements of array
        For i = 0 To myArray.Length - 1
            For j = 0 To myArray(i).Length - 1
```

```
                Console.WriteLine("The value at ({0}, {1}) is {2}",
i, j, myArray(i)(j))
            Next
        Next
    End Sub
End Module
```

3. What is the difference between an array and a collection?

The size of an array is fixed and must be specified at the time of its creation, while the size of a collection is not fixed and increases and decreases as the number of elements in the collection changes. You can define the type of elements that the array can hold but you can not specify the type of elements stored in the collection. In a collection, all the elements are stored as object type. In case you need a collection that should store only a particular type of element, subclass the collection and override its Add() and Get() methods to store and retrieve the elements of the specific type.

4. Write your own class for a Stack using the object array internally. (You don't need to worry about thread safety and other issues. Your class should only contain the Push(), Pop() and Peek() methods and the Count property)

```
Imports System
Module Test
    Public Sub Main()
        Dim st As New Stack()
        Console.WriteLine("After intialization, (size, capacity) = " +
st.size.ToString() + ", " + st.capacity.ToString())
        Console.WriteLine(vbCrLf & "Performing 5 Push() operations...")
        Dim i As Integer
        For i = 1 To 5
            Console.WriteLine("Pushing " + st.Push(i).ToString())
        Next
        Console.WriteLine(vbCrLf & "After 5 Push() operations, (size,
capacity) = " + st.size.ToString() + ", " + st.capacity.ToString())
        Console.WriteLine(vbCrLf & "Peek() operation gives " + st.Peek
().ToString())
        Console.WriteLine("After Peek() operation, (size, capacity) = " +
st.size.ToString() + ", " + st.capacity.ToString())
        Console.WriteLine(vbCrLf & "Performing 5 Pop() operations...")
        Console.WriteLine(st.Pop())
        Console.WriteLine(st.Pop())
        Console.WriteLine(st.Pop())
        Console.WriteLine(st.Pop())
        Console.WriteLine(st.Pop())
        Console.WriteLine(vbCrLf & "After 5 Pop() operations, (size,
capacity) = " + st.size.ToString() + ", " + st.capacity.ToString())
    End Sub
End Module
Class Stack
    Dim elements() As Object
    Public size As Integer
    Public capacity As Integer
```

```
    Public Sub New()
        size = 0
        capacity = 3
        ReDim elements(capacity - 1)
    End Sub
    Public Function Push(ByVal element As Object) As Object
        If size >= capacity Then
            Expand()
        End If
        elements(size) = element
        size += 1
        Return element
    End Function
    Public Function Pop() As Object
        size -= 1
        Return elements(size)
    End Function
    Public Function Peek() As Object
        Return elements(size - 1)
    End Function
    Private Sub Expand()
        Dim temp(capacity * 2) As Object
        Dim i As Integer
        For i = 0 To elements.Length - 1
            temp(i) = elements(i)
        Next
        capacity = capacity * 2
        elements = temp
    End Sub
End Class
```

5. How are Dictionaries different from simple collections such as ArrayList, Stack and Queue?

In dictionaries, elements are stored in a key-value pair fashion and each key maps to only one value. On the other hand, An ArrayList and other collections allow access to their elements in a predefined order like First In First Out (FIFO), Last In First Out (LIFO). Both the dictionary and simple collections keep track of the size of the collection on behalf of the programmer.

6. In this text we did not describe which algorithm the Hashtable uses to maintain the collection internally. Search for the text on rgw internet or in data structure's books and write a short description of this algorithm.

A hashtable consists of sub-groups (called buckets) of elements belonging to the collection. Elements are stored in the hashtable as key-value pairs. An object which is used as a key should be able to produce a hash-code; an integer, through its GetHashCode() method. Elements with the same hash-code are placed in the same bucket. When a value is being searched for within the Hashtable, the hash code is generated for that value and the bucket associated with that hash code is searched using the Equals() method.

We can see from the short description above that an object which is used as a key should override the System.Object class' GetHashCode() and Equals() methods properly. The GetHashCode() should use a quick algorithm to generate the hash-codes. It is also important that the GetHashCode() method produces the same hash-code every time it is called for the same object, which is the reason why immutable objects are more often used as keys in hashtables. The hashing algorithm may generate the same hash-code for two different keys but the hashing function that produces a unique hash-code for each key result has better performance. Finally, the Equals() method should check for the object equality rather than the default reference equality as it is used to search for elements in buckets.

We still recommend that the reader search for more text on this very interesting topic in MSDN and on the Internet

7. Why is it advised to properly override the Equals() and the GetHashCode() methods of a class that is to be used as a key in Hashtable?

The Hashtable class uses the GetHashCode() method to find the hashcode for a particular object. This hashcode is used to select the storing bucket for an element. The Equals() method is used when searching for an element in a bucket selected on the basis of its hashcode.

8. What is the basic difference between the String and the StringBuilder class? What is the difference between the Intern() and Copy() method of the String class?

The String objects are immutable, that is the internal state of the object can not be changed by any of its operations. The StringBuilder object, on the other hand, are mutable which means that the operations in this class may change the internal state of the object. The Copy() method creates and returns a copy of the supplied string while the Intern() method returns the reference to the supplied string.

What's Next...

Next time, we will be discussing delegates and events in VB.Net. We will explore

- Thorough Introduction (coverage) of concept behind delegates
- Declaring and using delegates in your program
- Multi-cast delegates
- Events basics
- Implementing events through delegates
- Multicasting events

Delegates and Events

Lesson Plan

In this lesson we will explore the concept of delegates and events. We will start out by looking at the idea behind delegates and will see how delegates are used in VB.Net. Later we will explore multicast delegates and their significance. Finally we will learn about events and the event handling mechanism through delegates.

Delegates Basics

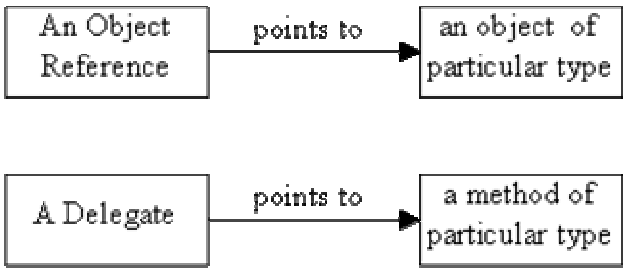
Delegates are referenced to methods. So far we have used references for objects such as

```
        Dim st As New Stack()
```

In this senario *st* is a reference to an object of the *Stack* class type. Each reference has two properties 1. The type of object (class), the reference can point to
2. The actual object referenced (or pointed) by the reference.

Delegates are similar to object references but are used to reference methods instead of objects. The type of a delegates is the type or signature of a method rather than the class. Hence a delegate has three properties 1. The type or signature of the method the delegate can point to
2. A delegate reference which can be used to reference a method
3. The actual method referenced by the delegate

Author's Note: The concept of delegates is similar to the function pointers used in C++.



1. The type or signature of the method the delegate can point to

Before using a delegate, we need to specify the type or signature of method the delegate can reference to. The signature of a method includes its return type and the type of parameter which it requires to be passed. For example,

```
Sub someMethod(ByVal args() As Integer)
```

is one of the common signatures of *Main()* method of VB.Net programs defined as

```
Sub Main(ByVal args() As Integer)
...
End Sub
```

and for the following *Add()* method

```
    Function Add(ByVal a As Integer, ByVal b As Integer) As Integer
        Return a + b
    End Function
```

The signature will be

```
    Function aMethod(ByVal p As Integer, ByVal q As Integer) As Integer
```

which is also the signature of the following *Subtract()* method

```
    Function Subtract(ByVal a As Integer, ByVal b As Integer) As Integer
        Return a - b
    End Function
```

It should be noticed from the above examples that the name of a method is not part of its signature but only its return type and parameters.

In case of delegates, we define the type of a delegate using the *Delegate* keyword as coded below

```
    Delegate Function MyDelegate(ByVal p As Integer, ByVal q As Integer) As Integer
```

Here we have defined a delegate type with the name '*MyDelegate*'. The reference of this delegate type can be used to point any method which takes two integers as parameters and returns an integer value.

2. The delegate reference which can be used to reference a method

Once we have defined a delegate type, we can set it to references actual methods with the

matching signature. A delegate reference can be declared just like an object reference. For example, a reference of type *MyDelegate* (defined above) can be declared as

```
        Dim arithMethod As MyDelegate
```

The delegate reference *arithMethod* can now reference any method whose signature is similar to the signature of *MyDelegate*

3. The actual method referenced by the delegate - the AddressOf keyword
The delegate reference can be made to reference any method with a matching signature by passing its name in the parameter of delegate type with the AddressOf keyword like below

```
        arithMethod = New MyDelegate(AddressOf Add)
```

We can see the *arithMethod* delegate reference has been made to point out the *Add()* method by passing its name as a parameter to the delegate type (*MyDelegate*). The *AddressOf* keyword here justifies that we are actually passing the address of the method '*Add*' to the delegate. Internally the delegate will use this address to call the '*Add*' method.

The last two steps can be merged together in a single statement as

```
        Dim arithMethod As New MyDelegate(AddressOf Add)
```

Calling the actual method through its delegate
Once the delegate reference '*arithMethod*' has been made to point out the *Add()* method, it can be used to call the actual method using the snippet code below

```
        Dim r As Integer = arithMethod(3, 4)
```

The complete source code of the program is is here for you to review

```
Imports System
Module Test
    Delegate Function MyDelegate(ByVal p As Integer, ByVal q As Integer) As Integer
    Public Sub Main()
        Dim arithMethod As New MyDelegate(AddressOf Add)
        Dim r As Integer = arithMethod(3, 4)
        Console.WriteLine("The resutl of arthmetic operation `+' on 3 and 4 is: {0}",
r)
    End Sub
    Function Add(ByVal a As Integer, ByVal b As Integer) As Integer
        Return a + b
    End Function
End Module
```

The result of the above code will be

```
The result of arthmetic operation `+' on 3 and 4 is: 7
Press any key to continue
```

The above code can be changed so that the arithmetic operation can be selected by the user as

```
Imports System
Module Test
    Delegate Function MyDelegate(ByVal p As Integer, ByVal q As Integer) As Integer
    Public Sub Main()
        Dim arithMethod As MyDelegate = Nothing
        Console.WriteLine("Which arithmetic operation you like to perform on 3 and
4?")
        Console.WriteLine("Press + for Add        ")
        Console.WriteLine("Press - for Subtract   ")
        Console.Write("Press m for Maximum Number ")
        Dim choice As Char = Convert.ToChar(Console.Read())
        Select Case choice
            Case "+"c
                arithMethod = New MyDelegate(AddressOf Add)
            Case "-"c
                arithMethod = New MyDelegate(AddressOf Subtract)
            Case "m"c
                arithMethod = New MyDelegate(AddressOf Max)
        End Select
        Dim r As Integer = arithMethod(3, 4)
        Console.WriteLine(vbCrLf & "The result of arithmetic operation {0} on 3 and 4
is: {1}", choice, r)
    End Sub
    Function Add(ByVal a As Integer, ByVal b As Integer) As Integer
```

```
        Return a + b
    End Function
    Function Subtract(ByVal a As Integer, ByVal b As Integer) As Integer
        Return a - b
    End Function
    Function Max(ByVal c As Integer, ByVal d As Integer) As Integer
        If c > d Then
            Return c
        Else
            Return d
        End If
    End Function
End Module
```

Here we have defined three methods of the same signature; *Add()*, *Subtract()* and *Max()*. A delegate type called *MyDelegate* is defined so that its reference *arithDelegate* can point to any method with a similar signature. The delegate reference '*arithDelegate*' is used to point out the particular method based on the user input at runtime. The sample output of the code is

```
Which arithmetic operation you like to perform on 3 and 4?
Press + for Add
Press - for Subtract
Press m for Maximum Number -

The result of arithmetic operation - on 3 and 4 is: -1
Press any key to continue
```

In the output above the users input was '-', the delegate reference is made to reference and call the *Subtract()* method. The above program shows that the same delegate reference can be used to point various methods as long as their signature is the same as the signature specified by the delegate type.

Confusion in terminology

Unfortunately the same term 'delegate' is used for both 'delegate type' and 'delegate reference' which could easily create confusion in the mind of a .NET learner. For the sake of clarity we are going to continuously use the term 'delegate type' and 'delegate reference' and will recommend you as the reader to also use these.

Delegates in .Net Framework

Although VB.Net presents delegates as a keyword and a first class language construct. In .Net delegates are present as reference type and all delegates inherit from the *System.Delegate* type. Technically our prior definition that 'a delegate is a reference to methods' is not quite appropriate. A delegate is a reference type derived from *System.Delegate* and its instances can be used to call methods of a matching signature. Another important thing to note here is that since defining a delegate means creating a new sub-type of *System.Delegate*, the delegates can not be defined within a method (which is also true for ordinary types). This is the reason why we have defined the delegate *MyDelegate* outside the *Main()* method in the example code in this lesson

```
Module Test
    Delegate Function MyDelegate(ByVal p As Integer, ByVal q As Integer) As Integer
    Public Sub Main()
```

```
        Dim arithMethod As MyDelegate = Nothing
           ...
    End Sub
```

Passing delegates to methods

Just like a reference to an object can be passed to other objects, the delegate reference of one method can be passed to another method. For example, lets make a method *'PerformArithOperatoin()'* which takes two integers and a delegate reference of type *MyDelegate*. Call the encapsulated method using the two integers.

```
    Sub PerformArithOperation(ByVal a As Integer, ByVal b As Integer, ByVal
arithOperation As MyDelegate)
        Dim r As Integer = arithOperation(a, b)
        Console.WriteLine(vbCrLf & "The result of arithmetic operation on 3 and 4 is:
{0}", r)
    End Sub
```

Now in *Main()* method we will call this method as

```
        PerformArithOperation(3, 4, arithMethod)
```

The task of collecting and printing the result has been delegated (or transferred) to the *PerformArithOperation()* method. The complete source code of this opperation is printed below

```
Imports System
Module Test
    Delegate Function MyDelegate(ByVal p As Integer, ByVal q As Integer) As Integer
    Public Sub Main()
        Dim arithMethod As MyDelegate = Nothing
        Console.WriteLine("Which arithmetic operation you like to perform on 3 and
4?")
        Console.WriteLine("Press + for Add        ")
        Console.WriteLine("Press - for Subtract   ")
        Console.Write("Press m for Maximum Number ")
        Dim choice As Char = Convert.ToChar(Console.Read())
        Select Case choice
            Case "+"c
                arithMethod = New MyDelegate(AddressOf Add)
            Case "-"c
                arithMethod = New MyDelegate(AddressOf Subtract)
            Case "m"c
                arithMethod = New MyDelegate(AddressOf Max)
        End Select
        PerformArithOperation(3, 4, arithMethod)
    End Sub
    Sub PerformArithOperation(ByVal a As Integer, ByVal b As Integer, ByVal
arithOperation As MyDelegate)
        Dim r As Integer = arithOperation(a, b)
        Console.WriteLine(vbCrLf & "The result of arithmetic operation on 3 and 4 is:
{0}", r)
    End Sub
    Function Add(ByVal a As Integer, ByVal b As Integer) As Integer
        Return a + b
    End Function
```

```
    Function Subtract(ByVal a As Integer, ByVal b As Integer) As
Integer
        Return a - b
    End Function
    Function Max(ByVal c As Integer, ByVal d As Integer) As Integer
        If c > d Then
            Return c
        Else
            Return d
        End If
    End Function
End Module
```

Multicast Delegates
A special feature of delegates is that a single delegate can encapsulate more than one method of a matching signature. These kind of delegates are called 'Multicast Delegates'. Internally, multicast delegates are the sub-types of *System.MulticastDelegate* which itself is a subclass of *System.Delegate*. The most important point to remember about multicast delegates is that "The multicast delegate type must be a Sub procedure or a sub routine". The reason for this limitation is that a multicast delegate may have multiple methods in its invocation list. Since a single delegate (or method) invocation can return only a single value. Hence a multicast delegate type must have no return type.

In VB.Net, multicast delegates are used with events.

Events and Event Handling
Events are certain actions that happen during the execution of a program that the application wishes to be notified about, so it can respond. An event can be a mouse click, a keystroke or an exact time (alarm). An event is basically a message which is said to be fired or triggered when a respective action occurs. A class that raises an event is called 'an event sender', a class that receives an event is called 'an event consumer' and the method which is used to handle a particular event is called 'an event handler'.

Author's Note: Event handling in .Net follows the Publisher-Subscriber and Observer Design Patterns. Truly speaking, if you are using Visual Studio.Net for developing your VB.Net applications (which most of us do), you don't need to learn or at least remember the event handling mechanism as it is provided to you automatically by Visual Studio.Net's IDE. But as TanenBaum, the famous writer of many computer science books once wrote "…Finally, like eating spinach and learning Latin in high school, some things are considered good for you in some abstract way!"

Event Handling in VB.Net
In .Net events are implemented as multicast delegates. In VB.Net events are a first class (basic) language construct and are defined using the Event keyword. The steps for implementing events and event handling are 1. Define a public delegate for the event outside a class boundary. The conventional signature of delegate is

```
Public Delegate Sub EventDelegate(ByVal sender As Object, ByVal e As EventArgs)
```

2. Define a class to generate or raise an event

- Define a public event in a class using the Event keyword and the Public delegate like

```
Public Event MyEvent As EventDelegate
```

- Write some logic to raise the event using the *RaiseEvent* keyword. While raising an event, the first argument is the sender or originator of the event. Usually the *Me* reference that is passed as the first argument. The second argument is a sub-type

of *System.EventArgs* which holds any additional data to be passed to the event handler. An event is generally raised like

```
Class SomeEventArgs
        Inherits EventArgs
        ...
End Class
Dim someData As New SomeEventArgs('some necessary arguments')
RaiseEvent MyEvent(Me, someData)
```

or if no data needs to be sent, the event is raised as

```
RaiseEvent MyEvent(Me, Nothing)
```

3. Define a module or class to receive the events. This module or class is usually the main application class containing the *Main()* method

- Write an event handler method in the class. The signature of the event handler must be similar to the Public delegate created in step 1. The name of the event handler method conventionally starts with the word "*On*" like

```
Public Sub OnMyEvent(ByVal sender As Object, ByVal e As EventArgs)
        ' handle the event
End Sub
```

- Instantiate the event generator class created in step 2 like

```
Dim eventObj As New EventClass()
```

- Add the event handler written in the current class to the event generator class' event

```
AddHandler eventObj.MyEvent, AddressOf OnMyEvent
```

The event handler now *'OnMyEvent()'* will be called automatically whenever the event '*MyEvent*' is triggered.

A Clock Timer Example

Let's understand how events are implemented and received by the traditional "Clock Timer" example. The Clock Timer generates an event each second and notifies the interested clients through events. First we define a public delegate for the event calling it '*TimerEvent*' as

```
Public Delegate Sub TimerEvent(ByVal sender As Object, ByVal e As EventArgs)
```

Now we define a class '*ClockTimer*' to generate the event as

```
Class ClockTimer
    Public Event Timer As TimerEvent
    Public Sub Start()
```

```
        Dim i As Integer
        For i = 0 To 4
            RaiseEvent Timer(Me, Nothing)
            Thread.Sleep(1000)
        Next
    End Sub
End Class
```

The class contains an event '*Timer*' of type *TimerEvent* delegate. In the *Start()* method, the event '*Timer*' is raised each second for a total of 5 times. We have used the *Sleep()* method of *System.Threading*. This thread class takes the number of milliseconds the current thread will be suspended (end) as its argument. We will explore threading and its issues in coming lessons.

Next we need to define a module that will receive and consume the event. This is defined as so

```
Module Test
    Public Delegate Sub TimerEvent(ByVal sender As Object, ByVal e As EventArgs)
    Public Sub Main()
        Dim clockTimer As New ClockTimer()
        AddHandler clockTimer.Timer, AddressOf OnClockTick
        clockTimer.Start()
    End Sub
    Public Sub OnClockTick(ByVal sender As Object, ByVal e As EventArgs)
        Console.WriteLine("Received a clock tick event!")
    End Sub
End Module
```

The module contains an event handler method *'OnClockTick()'* which follows the ClockEvent delegates signature. In the *Main()* method of the module, we have created an instance of the event generator class '*ClockTimer*'. Later we registered (or subscribed) the *OnClockTick()* event handler to the '*Timer*' event of the *ClockTimer* class. Finally we called the *Start()* method, which will start the process of generating events in the *ClockTimer* class. The complete source code of the program is

```
Imports System
Imports System.Threading
Module Test
    Public Delegate Sub TimerEvent(ByVal sender As Object, ByVal e As EventArgs)
    Public Sub Main()
        Dim clockTimer As New ClockTimer()
        AddHandler clockTimer.Timer, AddressOf OnClockTick
        clockTimer.Start()
    End Sub
    Public Sub OnClockTick(ByVal sender As Object, ByVal e As EventArgs)
        Console.WriteLine("Received a clock tick event!")
    End Sub
End Module
Class ClockTimer
    Public Event Timer As TimerEvent
    Public Sub Start()
        Dim i As Integer
        For i = 0 To 4
            RaiseEvent Timer(Me, Nothing)
            Thread.Sleep(1000)
```

```
        Next
    End Sub
End Class
```

Note that we have also included the *System.Threading* namespace in the start of the program as we are using its Thread class in our code. The output of the program is

```
Received a clock tick event!
Received a clock tick event!
Received a clock tick event!
Received a clock tick event!
Received a clock tick event!
Press any key to continue
```

Each message is printed with a delay of one second and five messages are printed in total.

Multicast events
Since events are implemented as multicast delegates in VB.Net, we can subscribe multiple event handlers to a single event. For example review the Revised *Test* class

```
Module Test
    Public Delegate Sub TimerEvent(ByVal sender As Object, ByVal e As EventArgs)
    Public Sub Main()
        Dim clockTimer As New ClockTimer()
        AddHandler clockTimer.Timer, AddressOf OnClockTick1
        AddHandler clockTimer.Timer, AddressOf OnClockTick2
        clockTimer.Start()
    End Sub
    Public Sub OnClockTick1(ByVal sender As Object, ByVal e As EventArgs)
        Console.WriteLine("Received a clock tick event!")
    End Sub
    Public Sub OnClockTick2(ByVal sender As Object, ByVal e As EventArgs)
        Console.WriteLine("Received a clock tick event in OnClockTick2!")
    End Sub
End Module
```

Here we have introduced another event handler '*OnClockTick2*' and have subscribed it also to the *Timer* event in the *Main()* method using the *AddHandler* keyword. The output of this program is

```
Received a clock tick event!
Received a clock tick event in OnClockTick2!
Received a clock tick event!
Received a clock tick event in OnClockTick2!
Received a clock tick event!
Received a clock tick event in OnClockTick2!
Received a clock tick event!
Received a clock tick event in OnClockTick2!
Received a clock tick event!
Received a clock tick event in OnClockTick2!
Press any key to continue
```

As can be seen in the output, now both the *OnClockTick()* and *OnClockTick2()* are invoked each time the event is raised.

An alternate implementation of event handlers in VB.Net
In the previous example we attached the list of multiple event handlers with a single event as:

```
    Dim clockTimer As New ClockTimer()
    AddHandler clockTimer.Timer, AddressOf OnClockTick1
    AddHandler clockTimer.Timer, AddressOf OnClockTick2
    ...
```

There is an alternate method in VB.Net to implement the same functionality. Here we attached different events with the same event handler. Suppose we have two buttons in our application and we want to attach the same event handler on the Click event of both the buttons. We can do this using the Handles keyword after the event handler sub procedure signature.

```
    Public Sub OnClockTick2(ByVal sender As Object, ByVal e As
EventArgs) Handles MyFirstButton.Click, MySecondButton.Click
        Console.WriteLine("Received a click event of first or second button")
    End Sub
```

However, to apply the Handles keyword to catch events we need to declare the Button object with the *WithEvents* keyword like:

```
    Private WithEvents MyFirstButton As Button
    Private WithEvents MySecondButton As Button
```

The *WithEvents* keyword tells the runtime (CLR) to search for the event handlers that process the events caused by objects.

Author's Note: This procedure of event handling using the 'Handles' keyword is more commonly used in VB.Net applications and is actually the one followed by the Visual Studio.Net IDE to implement event handling in Windows applications.

Passing data with the Event: Sub-classing System.EventArgs
Finally we can pass some additional information while raising an event. For this we need to perform the following three steps 1. Define a class that inherits from *System.EventArgs*
2. Encapsulate the data to be passed with the event within this class (preferably using properties)
3. Create an instance of this class in the event generator class and pass it with the event

Let's now change our previous Clock Timer example so that the event raised may also contain the sequence number of the clocks tick. First we need to define a new class '*ClockTimerArgs*' by inheriting it with the *System.EventArgs* class as

```
Public Class ClockTimerArgs
    Inherits EventArgs
    Private mTickCount As Integer
```

```
    Public Sub New(ByVal pTickCount As Integer)
        mTickCount = pTickCount
    End Sub
    Public ReadOnly Property TickCount() As Integer
        Get
            Return mTickCount
        End Get
    End Property
End Class
```

The *ClockTimerArgs* class contains a private variable '*mTickCount*' to hold the current tick number. This value is passed to the object through a public constructor and is accessible to the event handler through the public property. Next we need to change the delegate definition for the event as

```
    Public Delegate Sub TimerEvent(ByVal sender As Object, ByVal e As ClockTimerArgs)
```

The argument type in the delegate is changed from *EventArgs* to the *ClockTimerArgs* so that the publisher (event generator) can pass this particular type of arguments to the subscriber (event handler). The event generator class is defined as

```
Class ClockTimer
    Public Event Timer As TimerEvent
    Public Sub Start()
        Dim i As Integer
        For i = 0 To 4
            RaiseEvent Timer(Me, New ClockTimerArgs(i + 1))
            Thread.Sleep(1000)
        Next
    End Sub
End Class
```

The only change in this class is that instead of passing Nothing as the second argument; we are passing a new object of *ClockTimerArgs* type with the sequence number of the current clock tick.

Finally, the event handler is written as

```
    Public Sub OnClockTick(ByVal sender As Object, ByVal e As ClockTimerArgs)
        Console.WriteLine("Received a clock tick event. This is clock tick number
{0}", e.TickCount)
    End Sub
```

Here we have simply printed the clock tick number using the *ClockTimerArgs*' *TickCount* Property. The complete source code is

```
Imports System
Imports System.Threading
Module Test
    Public Delegate Sub TimerEvent(ByVal sender As Object, ByVal e As ClockTimerArgs)
    Public Sub Main()
        Dim clockTimer As New ClockTimer()
        AddHandler clockTimer.Timer, AddressOf OnClockTick
```

```
        clockTimer.Start()
    End Sub
    Public Sub OnClockTick(ByVal sender As Object, ByVal e As ClockTimerArgs)
        Console.WriteLine("Received a clock tick event. This is clock tick number
{0}", e.TickCount)
    End Sub
End Module
Class ClockTimer
    Public Event Timer As TimerEvent
    Public Sub Start()
        Dim i As Integer
        For i = 0 To 4
            RaiseEvent Timer(Me, New ClockTimerArgs(i + 1))
            Thread.Sleep(1000)
        Next
    End Sub
End Class
Public Class ClockTimerArgs
    Inherits EventArgs
    Private mTickCount As Integer
    Public Sub New(ByVal pTickCount As Integer)
        mTickCount = pTickCount
    End Sub
    Public ReadOnly Property TickCount() As Integer
        Get
            Return mTickCount
        End Get
    End Property
End Class
```

When the above code is compiled and executed, we will see the following output

```
Received a clock tick event. This is clock tick number 1
Received a clock tick event. This is clock tick number 2
Received a clock tick event. This is clock tick number 3
Received a clock tick event. This is clock tick number 4
Received a clock tick event. This is clock tick number 5
Press any key to continue
```

As the output of the program illustrates, now we are also receiving the clock tick number along with each event.

Food for thought: Exercise 9

1. What are delegates? How they are represented in .Net?
2. What is the difference between the delegate type and the delegate reference?
3. What is the significance of multicast delegates?
4. How are events presented in VB.Net?
5. Events are implemented in VB.Net using multicast delegates. The multicast delegates must have no return type. This means that events can't return anything. Is this a weak point of the event handling mechanism in VB.Net?

Solutions of Last Issue's Exercise (Exercise 8)

1. What is the difference between an error and an exception?

Computer Programs may have two types of errors

i. Syntax errors which can be identified by compiler at compile time, e.g., using an undeclared variable

ii. Runtime errors which can not be identified (or predicted) by the compiler and show up only at runtime. The runtime errors can be permanent, e.g., attempt of using a null reference; this type of errors can be corrected by the programmer during debugging. Alternatively, the runtime errors can be unexpected like the absence of a file or the internet connection being dropped; the correction of these errors is the responsibility of the users of the program.

Exceptions are used for indicating runtime errors. Programmers usually write exception handling procedures to deal with the second type of runtime errors (unexpected runtime errors)

2. Why do we need exceptions?

As stated in the previous question, exceptions are used for runtime error handling. Errors are an inevitable part of the software life cycle. Hence, some scheme must be followed by the programmer to deal with the occurrence of these errors so their programs don't crash. This error handling scheme is called exception handling.

3. The Finally block is generally placed after the Try...Catch block. Why it is used? What is the difference between the regular code after the Try...Catch block and code in the Finally block

```
    Public Sub Main()
        Try
            ' some code
        Catch e As InvalidArgumentException
            ' some code
```

```
        Finally
            ' code in Finally block
        End Try
        ' some regular code
    End Sub
```

The code in the Finally block is guaranteed to always execute whether an exception is raised in the Try block or not. On the other hand, the code after Try...Catch...Finally or Try...Finally block will only be executed
i. If no exception is raised in the Try block,
ii. Or the Catch block for the raised exception is present.

It will not execute if an exception is raised in the Try block which is not handled by an immediate Catch block.

4. Why should we define our own custom exception? What will be the difference if instead of throwing the custom InvalidArgumentException we throw an Exception class object in Divide() method like

```
Throw New Exception("The second argument of Divide is found zero")
```

The purpose of an exception is not just displaying the error message. The basic purpose of exception handling is to enable the program to execute even in the presence of an unexpected error which may cause the program to degrade in performance or functionality. We write our own custom exceptions so as to identify a particular type of error (failure) during the execution of the program so that we can do some work for the recovery related to this error.

5. Why do we need multiple Catch blocks when we can catch all the exceptions by the base class exception (Exception)?
The answer to this question is same as that of the previous question. We need multiple catch blocks so that we can separately identify each kind of error and can perform some repair work related to the kind of error.

What's Next...
From next time, we will start building the Windows Applications using VB.Net. We will explore

- WinForms and Windows Applications
- WinForm Basics

- Building "Hello WinForm" Application
- Describing the code-behind WinForm
- Using Visual Studio.Net to build Windows Form Applications
- Understanding Forms, Labels, Buttons, TextBox, Panels, CheckBoxes, RadioButtons; their properties, events and methods

WinForms and Windows Applications

Lesson Plan

Today we will start building Windows Applications in VB.Net. We will start by looking at the architecture of Windows Applications and their support in .Net. Later we will design our first "Hello WinForm" Application and learn about various windows form controls. Finally we will look at how Visual Studio.Net eases the creation of Windows Applications.

Windows Applications and .Net

VB.Net and .Net provide extensive support for building Windows Applications. The most important point about windows applications is that they are 'event driven'. All windows applications present a graphical interface to their users and respond to user interaction. This graphical user interface is called a 'Windows Form' or a 'WinForm' for short. A windows form may contain text labels, push buttons, text boxes, list boxes, images, menus and a vast range of other controls. In fact, a WinForm is also a windows control just like a text box, label, etc. In .Net all windows controls are represented by base class objects contained in the System.Windows.Forms namespace.

WinForm Basics

As stated earlier, .Net provides the WinForm and other controls through base classes in the System.Windows.Forms namespace. The class System.Windows.Forms.Form is the base class of all WinForms in .Net. In order to design a windows application, we need to:
1.Create a Windows Application project in Visual Studio.Net, or add references to System.Windows.Forms and System.Drawing to your current project. If you are not using Visual Studio at all, use the /reference option of the command line compiler to add these assemblies.
2. Write a new class to represent the WinForm and derive it from the System.Windows.Forms.Form class:

```
Public Class MyForm
    Inherits System.Windows.Forms.Form
    ...
End Class
```

3. Instantiate various controls, set their appropriate properties and add these to MyForm's Controls collection.
4. Write another class containing the Main() method. In the Main() method, call the System.Application.Run() method, supplying it with an instance of MyForm.

```
Class Test
    Public Sub Main()
        Application.Run(New MyForm())
    End Sub
End Class
```

The Application.Run() method registers your form as a windows application in the operating system so that it may receive event messages from the Windows Operating System.

Building the "Hello WinForm" Application

Let's build our first windows application, which we will call "Hello WinForm". The application will present a simple window with a "Hello WinForm" greeting at the center. The source code of the program is:

```
Imports System
Imports System.Windows.Forms
Imports System.Drawing
Class Test
    Public Sub Main()
        Application.Run(New MyForm())
    End Sub
End Class

Class MyForm
    Inherits Form
    Public Sub New()
        MyBase.new()
        Me.Text = "My First Windows Application"
        Me.Size = New Size(300, 300)
        Dim lblGreeting As New Label()
        lblGreeting.Text = "Hello WinForm"
        lblGreeting.Location = New Point(100, 100)
        Me.Controls.Add(lblGreeting)
    End Sub
End Class
```

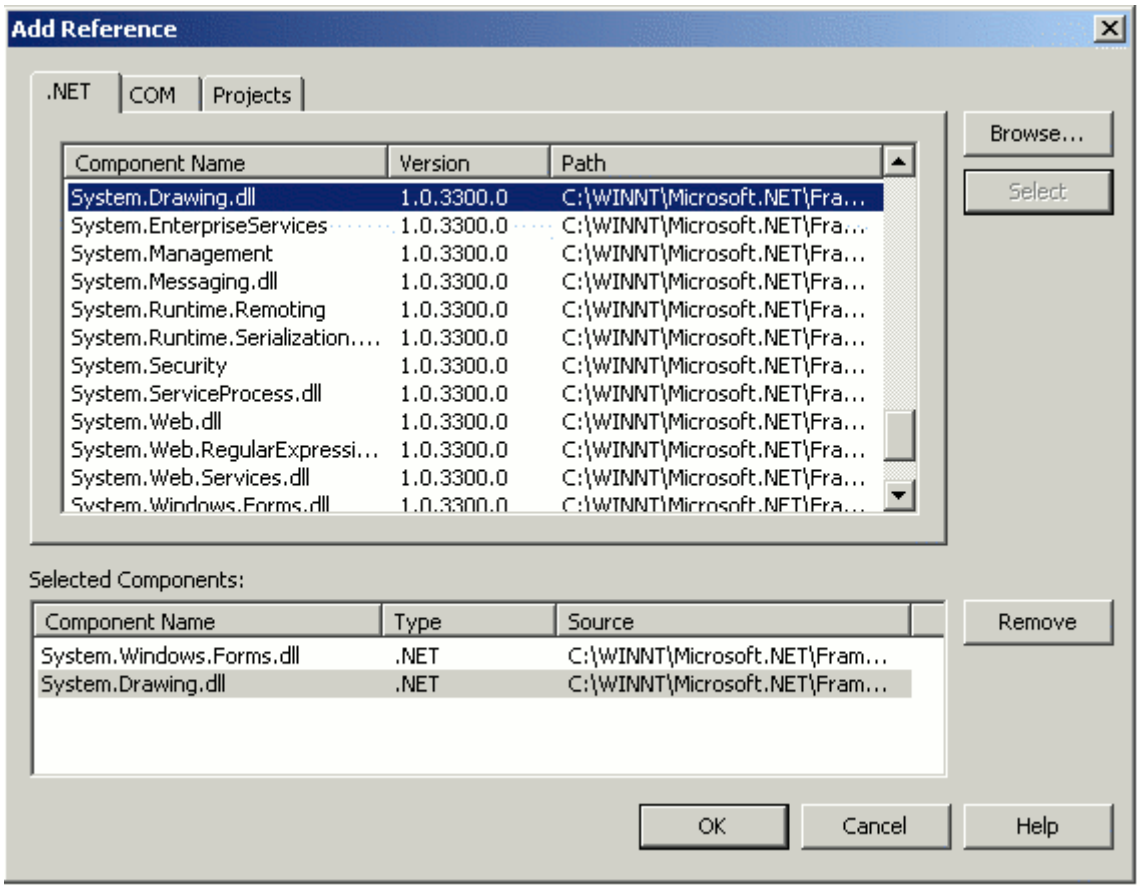
Understanding the Code

At the start we included three namespaces in our application:

```
Imports System
Imports System.Windows.Forms
Imports System.Drawing
```

The System namespace as stated in the first lesson, is the necessary ingredient of all VB.Net applications. In fact the Application class that we used later in the Main() method is defined in this namespace. The System.Windows.Forms namespaces contains the base classes for windows controls, e.g. Forms, Labels and Buttons. Finally, including the System.Drawing namespace is necessary as it contains the classes related to the drawing of the controls. The Size and Point classes used later in the program are actually defined in the System.Drawing namespace. The System.Windows.Forms and System.Drawing namespaces can not be added to a project just by typing the Imports statement for them. We also need to add references to these assemblies (for now think of assemblies as Windows DLL's or library files for .Net) that contain the code of these namespaces. For this we need to add a reference to the System.Windows.Forms.dll and the System.Drawing.dll. To add a

reference, right click the 'Reference' folder under the current project in the solution explorer and select Add Reference... It will show the following window



From the '.Net' tab of the Add Reference Dialog box, select the System.Windows.Forms.dll and System.Drawing.dll and select OK. This completes the process of adding references to assemblies in your project. Lets go back to our sample application. Later we derived a new class 'MyForm' from the Form class defined in System.Windows.Forms.

```
Class MyForm
    Inherits Form
    ...
    End Sub
End Class
```

In the constructor of MyForm, we specified the size and title of the form (by setting the size and text properties). The size is defined using the System.Drawing namespace's Size class. We passed two integers to the constructor of Size to specify the width and the height of the form.

```
    Public Sub New()
        MyBase.new()
```

```
        Me.Text = "My First Windows Application"
        Me.Size = New Size(300, 300)
```

In the constructor we created a text label and added it to the Controls collection of the Form. A text label is used to write some text on the form. The System.Windows.Forms.Label class defines a text label in a Windows application. We set the text of the Label using its Text property, which is of the string type. All the controls contained by a form must be added to its Controls collection; hence we have also added our label to this collection.

```
    Public Sub New()
        MyBase.new()
        Me.Text = "My First Windows Application"
        Me.Size = New Size(300, 300)
        Dim lblGreeting As New Label()
        lblGreeting.Text = "Hello WinForm"
        lblGreeting.Location = New Point(100, 100)
        Me.Controls.Add(lblGreeting)
    End Sub
```

Finally we have created a Test class containing a Main() method. In the Main() method, we have instantiated the MyForm class and passed its reference to the Application.Run() method so it may receive messages from the Windows Operating System.

When we execute the above code, the following screen is displayed:

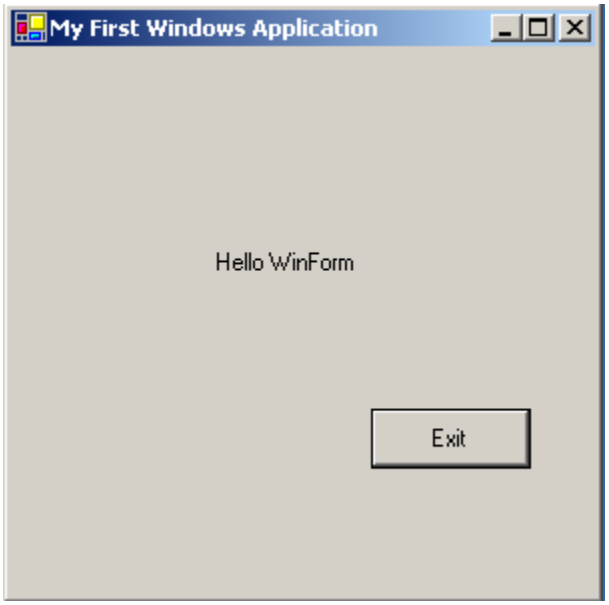
To close the application, press the close button on the title bar.

Adding Event Handling

Let's now add a button labeled 'Exit' to the form. The 'Exit' button will close the application when it is clicked. In .Net Push Buttons are instances of the System.Windows.Forms.Button class. To associate an action with the button click, we need to create an event handler and register (or add) it to the Button's Click event. Below is the code for this application.

```
Imports System
Imports System.Windows.Forms
Imports System.Drawing
Class Test
    Public Sub Main()
        Application.Run(New MyForm())
    End Sub
End Class

Class MyForm
    Inherits Form
    Public Sub New()
        MyBase.new()
        ' Form
        Me.Text = "My First Windows Application"
        Me.Size = New Size(300, 300)
        ' Label
        Dim lblGreeting As New Label()
        lblGreeting.Text = "Hello WinForm"
        lblGreeting.Location = New Point(100, 100)
        ' Button
        Dim btnExit As New Button()
        btnExit.Text = "Exit"
        btnExit.Location = New Point(180, 180)
        btnExit.Size = New Size(80, 30)
        AddHandler btnExit.Click, AddressOf BtnExitOnClick

        Me.Controls.AddRange(New Control() {lblGreeting, btnExit})
    End Sub
    Public Sub BtnExitOnClick(ByVal sender As Object, ByVal e As EventArgs)
        Application.Exit()
    End Sub
End Class
```

In the constructor of MyForm, we have set certain properties of the Form. In this code we have also used the StartPosition property of the Form, which sets the position of the form on the screen when the application starts. The type of this property is an enumeration called 'FormStartPosition'. We have set the start position of the form to the center of the screen. The new inclusion in the code is the Exit button called 'btnExit'. We have created the button using the base class System.Windows.Forms.Button. We have set various properties of the button, specifically its text label (Text), its Location and its Size. Finally we have created an event handler method for this button called BtnExitOnClick(). In the BtnExitOnClick() method, we have written the code to exit the application. We have also subscribed this event handler to the btnExit's Click event (To understand the event handling in VB.Net, see lesson 10 of the VB.Net school). In the end, we have added both the label and the button to the form's Controls collection. Note that this time we have used the AddRange() method of the form class to add an array of controls to the Controls collection of the form. This method takes an array of type Control as its parameter. When the code is run, the following window will be displayed:



Now you can press either the Exit Button or the close button at the title bar to exit the application.

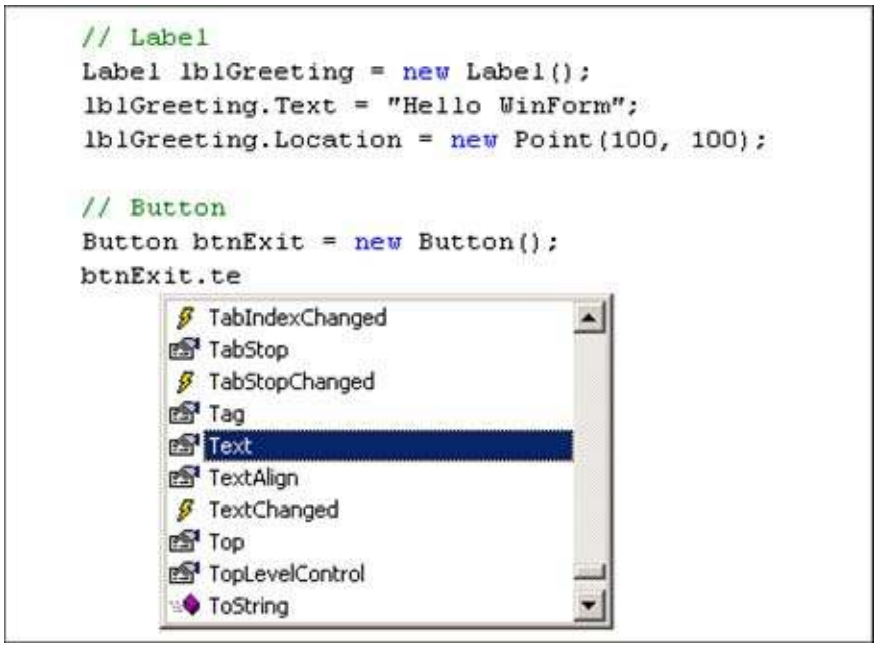
Alternate procedure for Event Handling - Using the 'Handles' Keyword
Note that in the previous code, we subscribed Exit buttons event handler using the 'AddHandler' keyword. We can also do this using the 'Handles' keyword with the Event Handler method. But to use the 'Handles' keyword, the Exit button needs to be an instance variable of the form class defined with the 'WithEvents' keyword. The complete source code of the MyForm class is

```
Class MyForm
    Inherits Form
    ' Form controls are usually private
    ' instance members of the form class
    Private WithEvents btnExit As New Button()
    Private lblGreeting As New Label()
    Public Sub New()
        MyBase.new()
        ' Form
        Me.Text = "My First Windows Application"
        Me.Size = New Size(300, 300)
        ' Label
        lblGreeting.Text = "Hello WinForm"
        lblGreeting.Location = New Point(100, 100)
        ' Button
        btnExit.Text = "Exit"
        btnExit.Location = New Point(180, 180)
        btnExit.Size = New Size(80, 30)
        'AddHandler btnExit.Click, AddressOf BtnExitOnClick

        Me.Controls.AddRange(New Control() {lblGreeting, btnExit})
    End Sub
    Public Sub BtnExitOnClick(ByVal sender As Object, ByVal e As EventArgs) Handles
btnExit.Click
        Application.Exit()
    End Sub
End Class
```

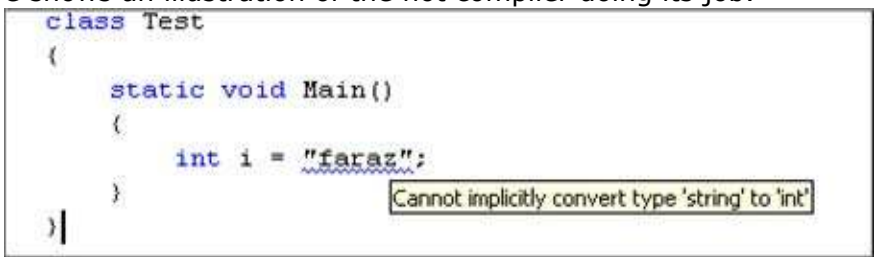
This procedure of event handling (using the 'Handles' keyword) is usually followed by Visual Studio.Net standard code for Form based applications (Windows and Web Applications)

Visual Studio.NET & its IDE (Integrated Development Environment)
Most of the time, you will be using Visual Studio.Net to develop Windows applications with VB.NET. Visual Studio.NET provides a lot of tools to help develop applications and cuts out a lot of work for the programmer. Visual Sutdio.NET provides a standard code editor and an IDE for all .NET applications. Along with a standard debugger, a project and solution manager, a form designer, an integrated compiler and a host of other useful tools.

IntelliSense and Hot Compiler
The Visual Studio.NET IDE provides a standard text editor to write .NET applications. The text editor is loaded with IntelliSense and a hot compiler. IntelliSense gives the text editor the ability to suggest different options with the programming syntax. For example, when you place a dot after the name of an object, the IDE automatically provides you a list of all the members (properties, methods, etc) of the object. The following figure shows IntelliSense at work in the Visual Studio.NET IDE.



The hot compiler highlights the syntax errors in your program as you type the code. The following figure shows an illustration of the hot compiler doing its job.



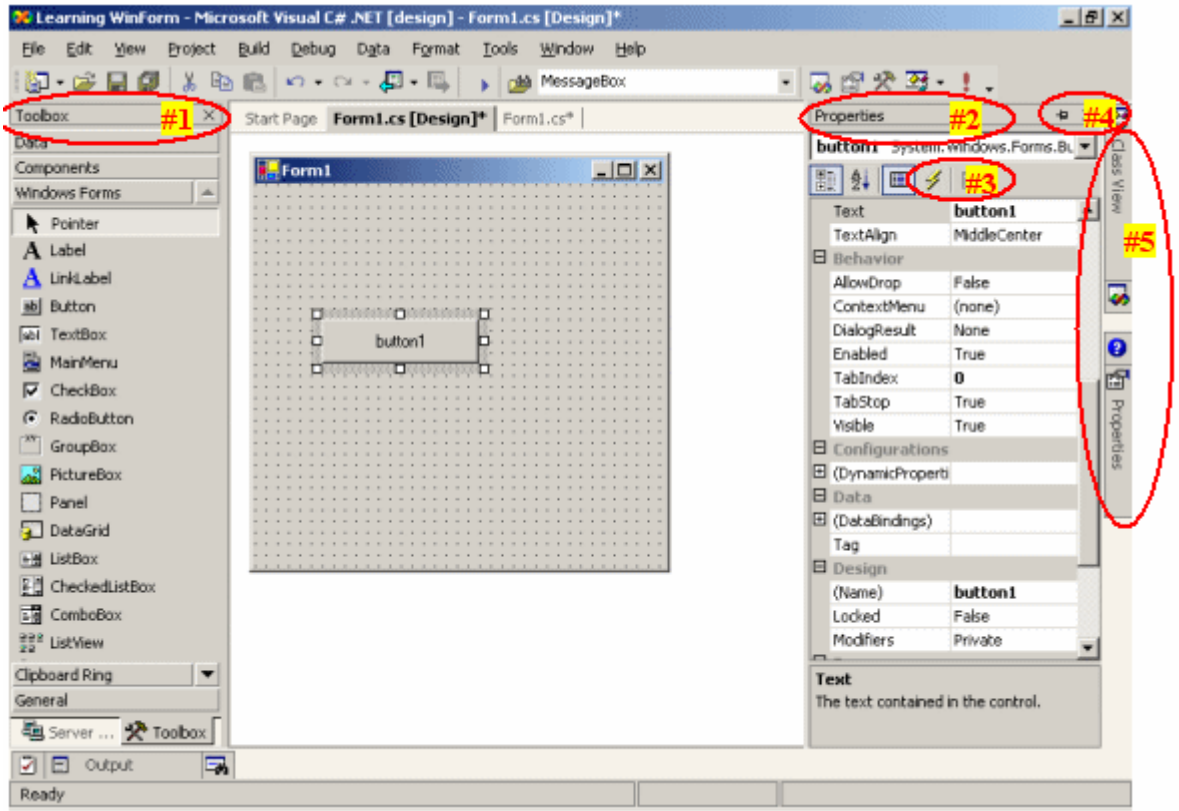
Code Folding
One of the pleasant new features introduced in Visual Studio.NET is code folding. With code folding, you can fold/unfold the code using the + and - symbols. Usually the code can be folded/unfolded at each scope boundary (method, class, namespace, property, etc). You can

also define regions within your code and can fold/unfold the code within that region. The region is defined using the #region...#end-region preprocessor directives.

```
Class MyForm
    Inherits Form

    ' Form controls are usually private
    ' instance members of the form class
    Private WithEvents btnExit As New Button()
    Private lblGreeting As New Label()

    Public Sub New()...

    Public Sub BtnExitOnClick(ByVal sender As Object, ByVal e As EventArgs) Handles btnE
        Application.Exit()
    End Sub

End Class
```

Integrated Compiler, Solution builder and Debugger
Visual Studio.NET provides an integrated compiler to compile and execute your application during development. You can either compile a single source file or the complete project and solution (a group of files that make up an application). Once you have compiled your application, you can debug it using the Visual Studio.NET debugger. You can even create an installer for your application using Visual Studio.NET.

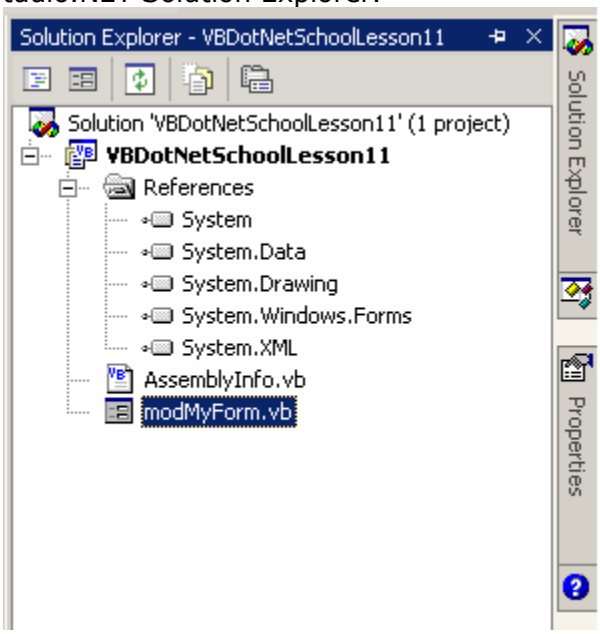
Form Designer
Perhaps the most useful feature of the Visual Studio.NET IDE is its form designer. The form designer allows you to design the graphical user interface by placing the controls on the form from the Toolbox. You can set a lot of the properties of the form and its controls using the Properties window. The Visual Studio.NET IDE automatically writes the code in the source file as you place the controls on the form and changes their properties. You can also use the IDE to create and set up the event handlers for your controls. The following figure presents an introductory view of the Visual Studio.NET Form Designer and its different supporting functions.



You can see the toolbox window at the left hand side (#1) and the properties window at the right hand side (#2) of the above snapshot. The toolbox allows you to add different controls to your form. Once the control is placed on the form, you can change its various properties from the Properties window. You can also change the location and size of the controls using the mouse. The Event properties can be changed by switching to the Event Properties pane (#3) in the Properties Window. The Toolbox, Properties Window, Help Window, Solution Explorer Window, Class View Window, Output Window and other helping windows in the Visual Studio IDE can be set for **Docking and Auto hiding**. Windows that are set for auto hide appears only when they get focus (e.g. they have mouse pointer over them or have receive a mouse click), and hide when they lose focus. A window can be set to auto hide by the button marked #4 in the above figure. The hidden windows are always accessible through the left and right panes of the form designer window. The right pane is marked with #5 in the above figure and holds the class view, help and solution explorer windows in a hidden state. If some of these windows are not visible in your visual studio IDE, you can make them visible from the View menu on the menu bar.

Solution Explorer

The Solution Explorer is a very useful window. It presents the files that make up the solution in a tree structure. A solution is a collection of all the projects and other resources that make up a .NET application. A solution may contain projects created in different .NET based languages such as VB.NET, C#.NET and VC++.NET. The following figure presents a snapshot of the Visual Studio.NET Solution Explorer.



The Reference node contains all the assemblies referenced in the respective project. AssemblyInfo.vb is a VB.NET file that contains information about the current assembly. modMyForm.vb in the above figure is the name of the source file of the program. A .NET solution is saved in a .sln file, a VB.NET project is saved in a .vbproj file and VB.NET source code is saved in a .vb file. It is important to understand here that Projects and Solutions are standards of Visual Studio.NET and are not part of the requirement of the .NET language. In fact, the language compiler is not even aware of a project or solution.

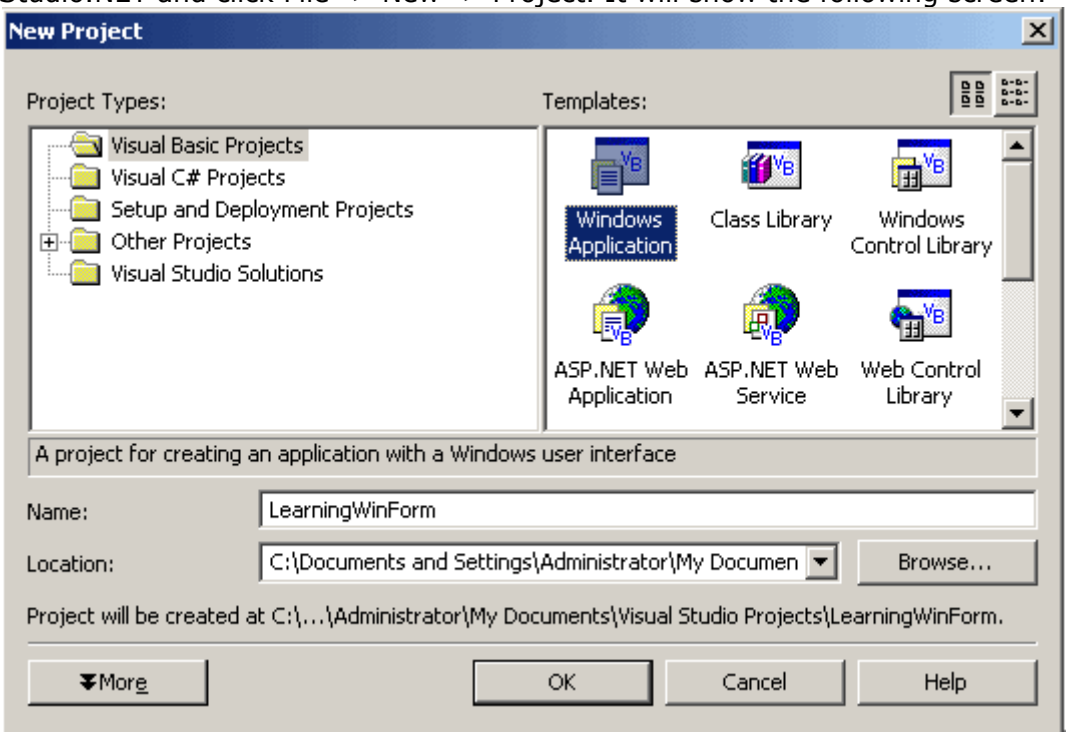
Menus in the Visual Studio .NET IDE

- File Menu: Used to create, open, save and close a project, solution or individual source files.
- Edit Menu: Used for text editing and searching within the Visual Studio source code editor.
- View Menu: Provides options for setting the visibility of different Visual Studio windows and to switch between code and designer views.
- Project Menu: Used for setting different properties of the Visual Studio Project. A Visual Studio project is a collection of files that make up a single assembly or a single object file (we will explore the concept of assemblies in the coming lessons).
- Build Menu: This menu is used to compile and build the source files, projects or solutions. The result of a build is an executable file or a code library.
- Debug Menu: This menu provides various options related to the Visual Studio.NET Debugger. Debugging is the process of finding logical errors in the code and the debugger helps make this proccess easier.

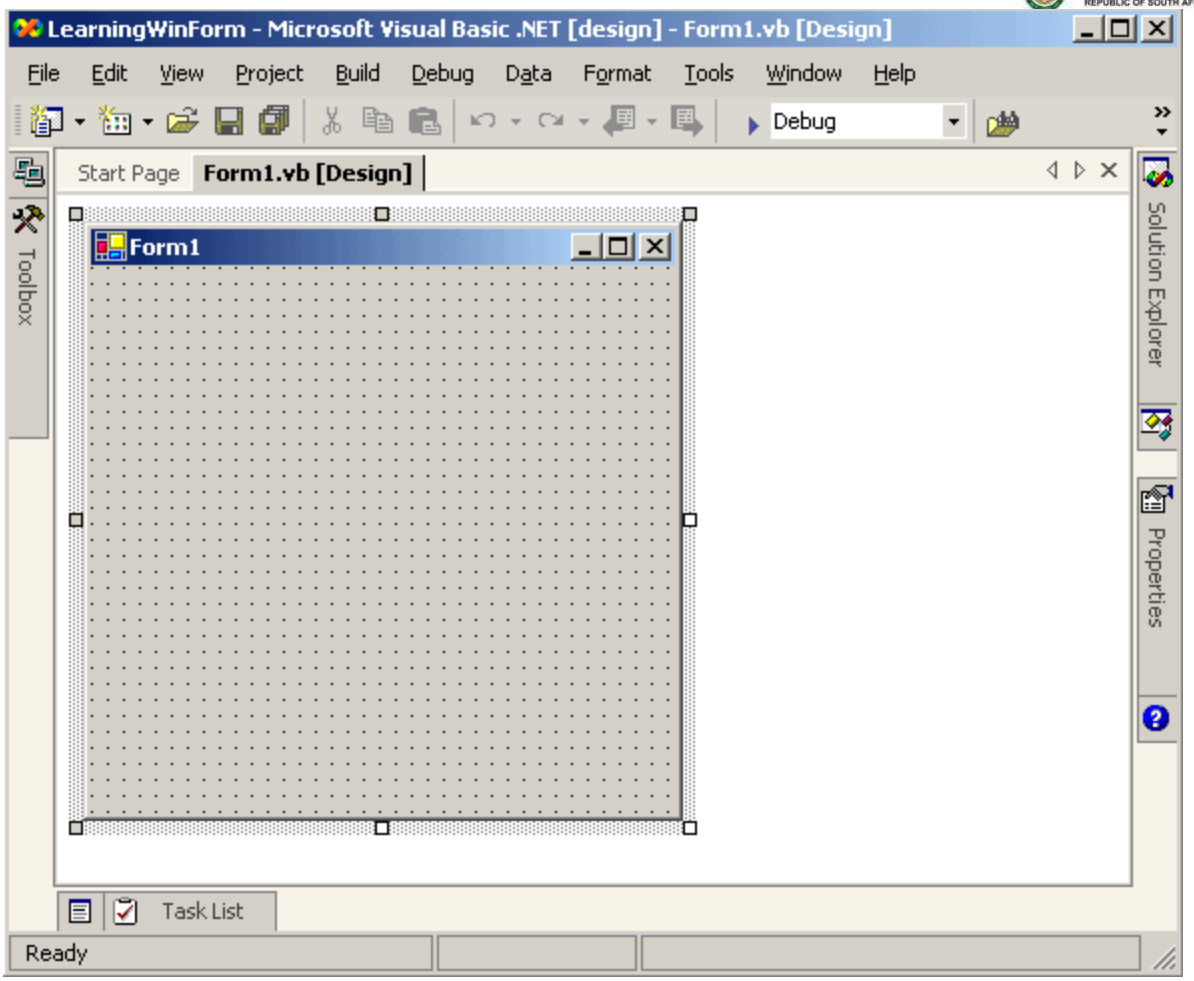
- Data Menu: Provides various options for Data Access in .NET
- Format Menu: Provides access to a set of useful operations for formatting the controls regards their layout in the Form Designer view.
- Tools Menu: Provides access to various useful Visual Studio.NET tools.

Using Visual Studio.NET to build the "Hello WinForm" Application
Now we've had a quick tour of Visual Studio.NET, let's use the Visual Studio.NET IDE to build the "Hello WinForm" application which we created earlier in the lesson.

Creating a new Project
First of all we need to create a new VB.NET Windows Application Project. For this, start Visual Studio.NET and click File -> New -> Project. It will show the following screen:



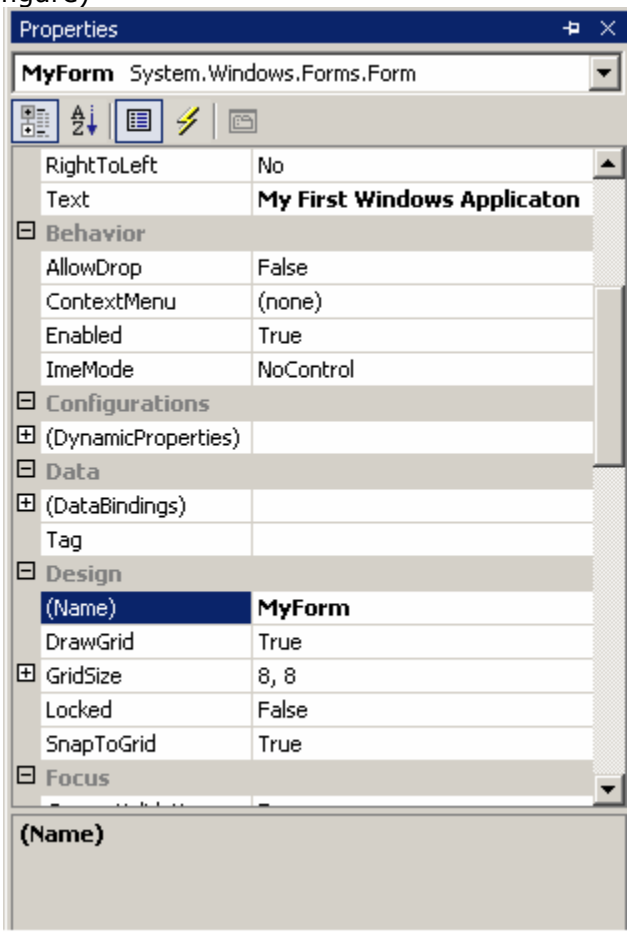
From the above screen, select 'Visual Basic Projects' from Project types and 'Windows Application' from the choice of Templates. Type the name of the new project ('LearningWinForm' in the above figure) in the text box labeled Name. Select the location where you wish to store the project using the Browse... Button and click OK. An empty form in the designer view similar to the figure below will now be at your command:



The layout of the form designer screen may be somewhat different from the one shown above. Your toolbox and properties window may be visible and some other windows may not be visible. You can change these settings using the View menu as described earlier in the lesson.

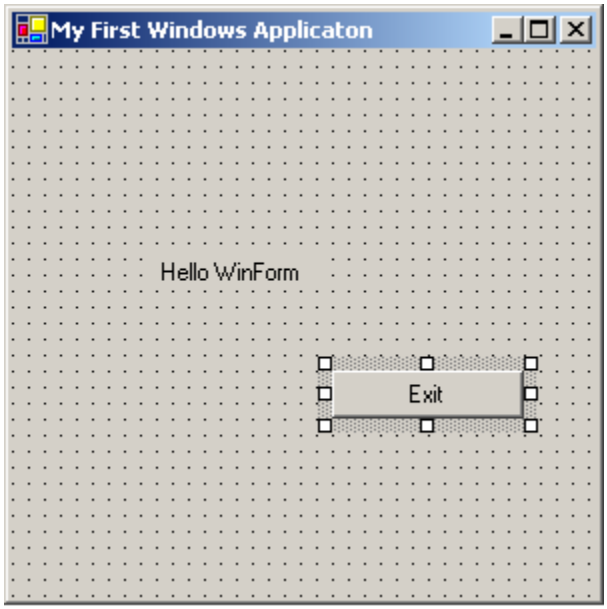
Setting various properties of the form

You can change the default properties of a form using the Properties window. For this, select (click) the form and select the properties window (If the properties window is not visible in the right hand pane, select View -> Properties -> Window). Change the title of the form and the name of the form's class in the code by changing the Text and Name properties. (As shown in the following figure)



Adding Controls to the Form

Select the Label control from the toolbox and place it on the form. Resize it as appropriate. Select (click) the label on the form, and in the properties window set its Text property to "Hello WinForm" and the Name property to 'lblGreeting'. The name of a control is the name of its corresponding instance in the source code. Select the Button control from the toolbox, place it on the form and resize it appropriately. Select (click) the button and set its Name property to 'btnExit' and its Text property to 'Exit' using the properties window. The form should now look like this:



Adding Event Handling

We need to add the event handling code for the Exit button. To do this, simply double click the Exit button in the designer. This will generate a new event handler for the Exit button's Click event and take you to the source code as shown in the following figure:

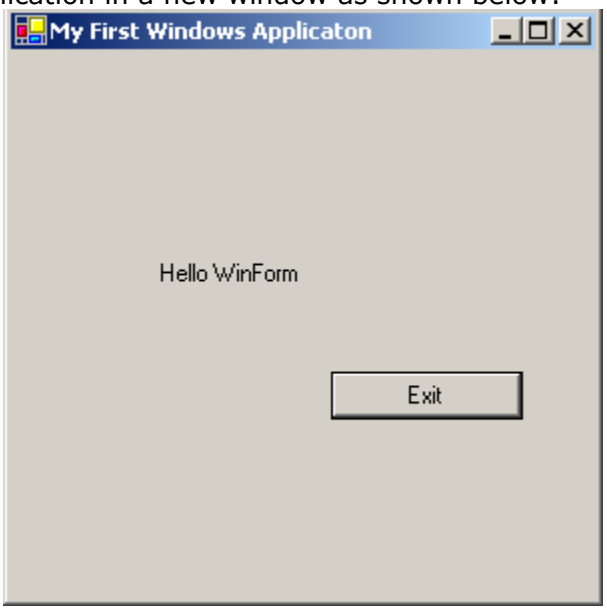
```
        Windows Form Designer generated code

        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.Run(new Form1());
        }

        private void btnExit_Click(object sender, System.EventArgs e)
        {
            Application.Exit();
        }
    }
}
```

Write the code to close the application (Application.Exit()) in the event handler. The IDE in fact has not only created the event handler but also has registered it with the Exit button's Click event.

Executing the application

That is it! The 'Hello WinForm' application is complete. To compile and execute the application, select Build -> Build Solution (or press Ctrl+Shift+B) and then select Debug ->

Start Without Debugging (or press Ctrl+F5). This will compile and start the 'Hello WinForm' application in a new window as shown below:



To close the application, click the Exit button or the close button on the title bar of the window.

The code generated by the Form Designer

You can toggle between the Form Designer and Code using View -> Designer and View -> Code. After switching to the code, you will find the code generated by the form designer to be very similar to that we have written earlier in the lesson. To understand the code better, we recommend removing all the comments and region boundaries.

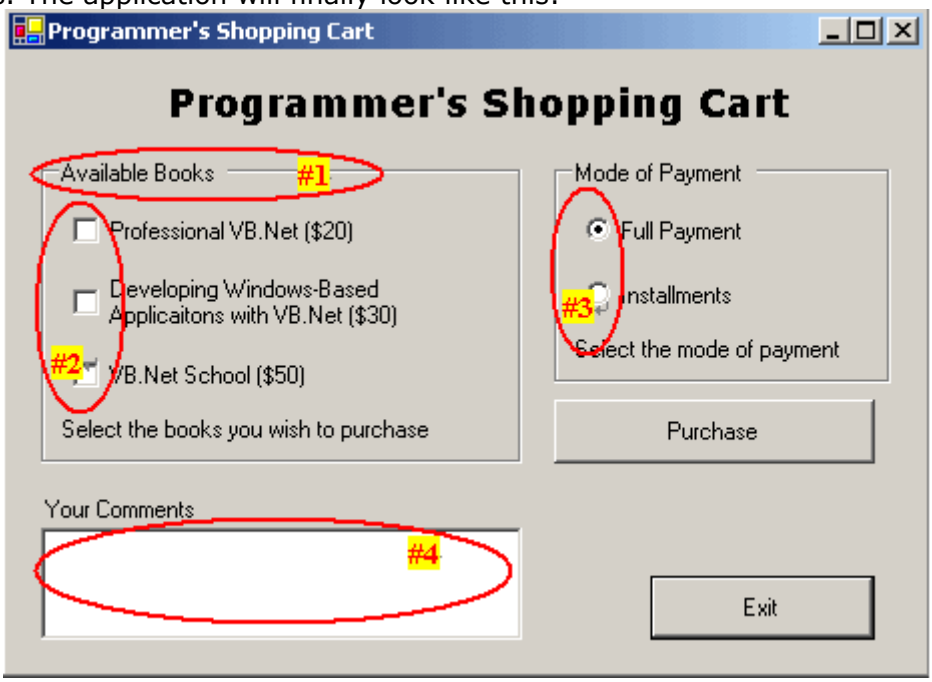
Using More Controls

We are getting more familiar with the Visual Studio.Net IDE and its controls. Let's learn about some more controls. Note that we will not completely define any control and will demonstrate only some of the more common properties and events. A summary of some of these controls is presented below.

Control	Description
Label	Used to display some text on a form. An instance of System.Windows.Forms.Label. The important properties are Text, Name and Font. Usually events are not handled for the Label control.
Button	Used to display a Push Button on a form. An instance of System.Windows.Forms.Button. The important properties are Text, Name and Font. Usually the Click event is handled for the Button.
TextBox	Provides the user with an area to write/edit text. An instance of System.Windows.Forms.TextBox. Its important properties are Text (to set startup text and get what the user has entered), Name, Alignment, Multiline (boolean), ScrollBars (a set of scroll bars attached with the text box), ReadOnly (boolean), WordWrap (boolean) and PasswordChar (character used for password masking). Important events are TextChanged (default) and KeyPress.
GroupBox	Used to group other controls. An instance of System.Windows.Forms.GroupBox. The important properties are Text, Visible (boolean) and Enabled (boolean). Usually events are not handled for the GroupBox.
RadioButton	Allows a user to select one out of available radio options. RadioButtons are usually used in a

	group contained in a GroupBox. Only one of the RadioButton can be selected in a group at a time. An instance of System.Windows.Forms.RadioButton. Important properties are Text, Name and Checked (boolean). Important event is CheckedChanged. Usually the events of a RadioButton are not handled; rather the selected choice is identified on the click of a push button or an action of other control.
CheckBox	Allows a user to tick or untick a box to state their preference for something. CheckBoxes are usually used in a group contained in a GroupBox. Any of the checkboxes in a group can be selected simultaniously. An instance of System.Windows.Forms.CheckBox. The important properties are Text, Name, Checked (boolean) and CheckState. Important events are CheckedChanged and CheckStateChanged.

Using various controls in an application: Programmer's Shopping Cart
Let's create a 'Programmer's Shopping Cart' application. The Programmer's Shopping Cart is an online bookstore. The best thing about it is that it sells books in both full payment and in installments. The application will finally look like this:



As you can see, we have used the GroupBox (#1), CheckBox (#2), RadioButton (#3), TextBox (#4), Label and Button controls in the above application.

Designing the form and placing the controls
First of all, start Visual Studio.NET and create a new Windows Application Project. Set the properties of the form using the Properties Window to the following values:- Size = 460, 340
StartPosition = CenterScreen
Name = MyForm
Text = Programmer's Shopping Cart

The StartPosition is the startup position of the form on the screen, the Name is the name of form class generated by the designer and Text is the title of the Application's Window.

Add a Label to the form as shown in the figure above. Set the Text property of the Label to 'Programmer's Shopping Cart'.

From the toolbox window, add a GroupBox to the form. Modify its size appropriately to accommodate three checkboxes and a Label. Set the Text property of the GroupBox to 'Available Books' and the Name property to 'gbxAvlblBooks'.

From the toolbox window, add three checkboxes to the GroupBox you just made. Set the Text property of the three checkboxes to 'Professional VB.NET ($20)', 'Developing Windows-Based Applications with VB.NET ($30)' and 'VB.NET School ($50)'. Also set the Name

property of the checkboxes to 'cbxProfVB', 'cbxDevWinApp' and 'cbxVBDotNetSchool'. Add a Label in the GroupBox and set its Text property to 'Select the books you wish to purchase', as shown in the figure.

Add another GroupBox, set its Text property to 'Mode of Payment', its Name property to 'gbxPaymentMode' and its Enabled property to 'False'. Making the Enabled property 'False' will disable the groupbox and all its contents at startup. It will only be enabled (through coding) when the user has selected some books. Add two RadioButtons in the GroupBox as shown in the figure. Set their Name properties to 'rbnFullPayment' and 'rbnInstallments'.

Set the Text property of the radio buttons to 'Full Payment' and 'Installments'. To make the first option (Full Payment) the default option, set its Checked property to True. Add a Label in the GroupBox and set its Text property to 'Select the mode of payment' as shown in the figure.

Add the Purchase button in the form. Set its Name property to 'btnPurchase', its Text property to 'Purchase' and its Enabled property to 'False'. Again the Purchase button will be enabled only when the user has selected some books.

Add a TextBox and a Label to the form. Set the Text property of the Label to 'Comments'. Set the Text property of the TextBox to "" (empty), the Name property to 'txtComments' and the MultiLine property to True. The purpose of setting the MultiLine property to true is to allow the TextBox to have text on more than one line. The default value of the MultiLine property is False.

Finally, add an Exit Button to the form. Set its Name property to 'btnExit' and its Text property to 'Exit'.

Writing Code for Event Handling

First of all, add an event handler for the Exit Button's Click event by double clicking on it in the designer. Write the following code to exit the application:

```
    Private Sub btnExit_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles btnExit.Click
        Application.Exit()
    End Sub
```

When the user has selected a book, the 'Mode of Payment' group box and the Purchase button should be enabled. If all the books are unselected they should be disabled. This is done by writing an event handler for CheckedChanged event of the checkboxes. To add an event handler for a checkbox, double click the checkbox in the designer. The CheckedChanged event of the checkbox is triggered whenever the checkbox is checked, unchecked or its Checked property is changed. Write the following code in the event handler of the first checkbox:

```
    Private Sub cbxProfVB_CheckedChanged(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles cbxProfVB.CheckedChanged
        If cbxProfVB.Checked = False And cbxDevWinApp.Checked = False And
cbxVBDotNetSchool.Checked = False Then
            gbxPaymentMode.Enabled = False
            btnPurchase.Enabled = False
        Else
            gbxPaymentMode.Enabled = True
```

```
            btnPurchase.Enabled = True
        End If
    End Sub
```

In the code above, if all the checkboxes are unchecked we disable the 'Mode of Payment' group box and the Purchase button; otherwise we will enable them. Copy and paste the same code for the CheckedChanged event of the other two checkboxes (cbxDevWinApp and cbxVBDotNetSchool). Execute the program and check/uncheck the different checkboxes. You will notice if any of checkboxes are checked, the Mode of Payment group box and Purchase button are enabled, and if none of the checkboxes are checked, they are disabled.

Instead of writing the same but separate event handler for each checkbox, we can also write a single event handler to handle the CheckedChanged event of all the checkboxes as

```
    Private Sub cbxProfVB_CheckedChanged(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles cbxProfVB.CheckedChanged, _
                                                     cbxDevWinApp.CheckedChanged, _
                                                     cbxVBDotNetSchool.CheckedChanged
        If cbxProfVB.Checked = False And cbxDevWinApp.Checked = False And
cbxVBDotNetSchool.Checked = False Then
            gbxPaymentMode.Enabled = False
            btnPurchase.Enabled = False
        Else
            gbxPaymentMode.Enabled = True
            btnPurchase.Enabled = True
        End If
    End Sub
```

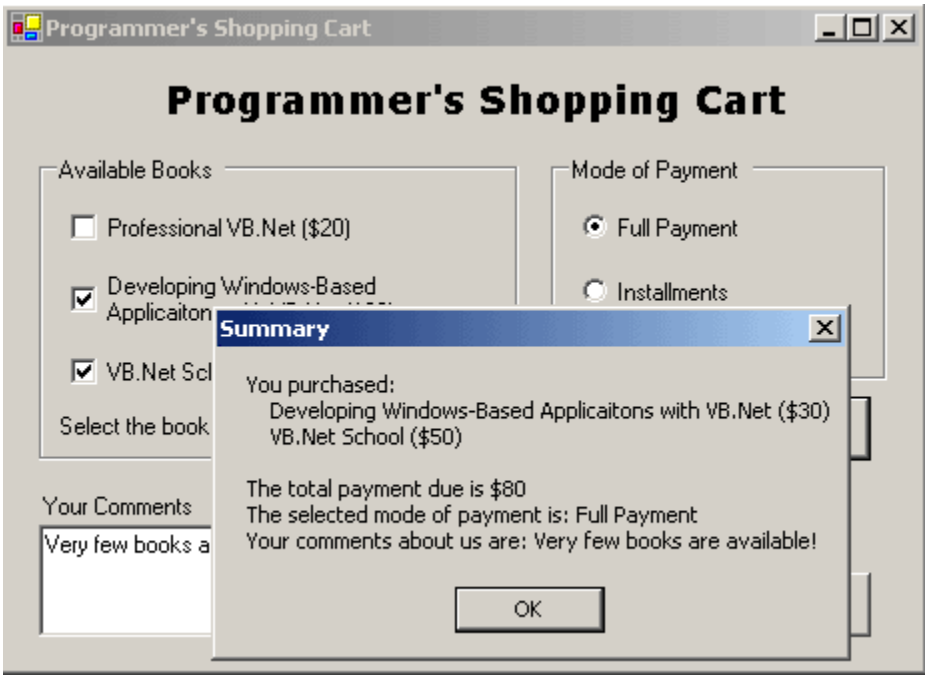
Here we have specified multiple events for this event handler separating each of them with a comma.

Finally, on the Click event of the Purchase Button. The program should display a summary containing a list of books selected, the total cost of the purchase, the mode of payment selected and any comments the user has provided. Add a Click event handler for the Purchase button by double clicking it in the designer and type the following code:

```
    Private Sub btnPurchase_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles btnPurchase.Click
        Dim message As String = "You purchased:" + vbCrLf + "    "
        Dim amount As Integer = 0
        If (cbxProfVB.Checked) Then
            amount += 20
            message += cbxProfVB.Text + vbCrLf + "    "
        End If
        If cbxDevWinApp.Checked Then
            amount += 30
            message += cbxDevWinApp.Text + vbCrLf + "    "
        End If
        If cbxVBDotNetSchool.Checked Then
            amount += 50
            message += cbxVBDotNetSchool.Text + vbCrLf + "    "
        End If
        Dim paymentMode As String = ""
        If (rbnFullPayment.Checked) Then
            paymentMode = rbnFullPayment.Text
        Else
            paymentMode = rbnInstallments.Text
        End If
        message += vbCrLf + "The total payment due is $" + amount.ToString()
        message += vbCrLf + "The selected mode of payment is: " + paymentMode
        If txtComments.Text <> "" Then
            message += vbCrLf + "Your comments about us are: " +
txtComments.Text
        End If
        MessageBox.Show(message, "Summary")
    End Sub
```

We have used three variables in the above code. The integer variable amount is used to hold the total amount of the purchase. The string variable paymentMode is used to hold the selected mode of payment. The string variable message will hold the summary message that will finally be displayed in the MessageBox. First, we checked each of the checkboxes and calculated the total amount; at the same time we also collated the list of books selected in the string variable message. We then found the mode of payment and concatenated (added) it to the message variable. Next, we concatenated the total payment and comments provided to the string variable message. Finally, we displayed the summary message in a Message Box. A message box is a dialog window that pops up with a message. A message box is a modal kind of dialog box, which means when it is displayed you can not access any other window of your application until you have closed the message box. The MessageBox class' shared 'Show' method has many overloaded forms; the one we used takes a string to display in the message box and the title of the message box.

When we execute the program and press the purchase button after selecting some books, we see the following output:



The complete source code of the application can be downloaded by clicking here

We hope you have started to get a good understanding of Windows applications and WinForms in this lesson. Practice is the key to success in windows programming. The more applications you design and the more experiments you perform, the better and stronger your understanding will be. No one can teach you all the properties and events of the controls. You must experiment and discover the use of these for yourself.

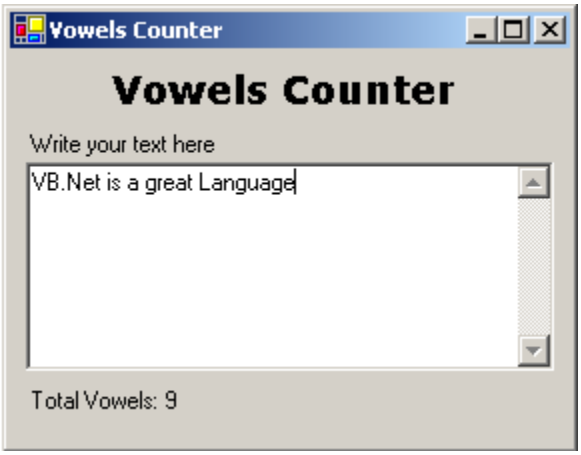
Some Important Points for designing Windows Applications

- Make your form layout simple and easy to understand. It is important that the user of your application finds it familiar. The behavior should be expected and should not surprise the user.
- The Format menu of the Visual Studio.NET IDE is very useful when designing the form layout. It provides a number of useful options for alignment and size of the controls.
- Almost all the controls have some similar properties like Location, Size, Enabled, Visible, TabIndex etc. The TabIndex property is very important. It describes the sequence followed by the windows focus when the user presses the Tab key.
- The controls should be named so that their purpose can be recognized, e.g., we have named the Purchase button 'btnPurchase' in the previous example.
- Although now it is not a standard convention, it is useful to add a three letter prefix to the name of your controls so that they are recognizable by their name.

Throughout the lesson, we have followed the convention by prefixing a Label control's name with lbl (lblGreeting), TextBox with txt (txtComments), Button with btn (btnPurchase), CheckBox with cbx (cbxProfVB), RadioButton with rbn (rbnFullPayment) and GroupBox with gbx (gbxPaymentMode).

Food for thought: Exercise 11

1. How does the .NET platform support windows applications?
2. The TextBox control has the properties ReadOnly and Enabled. Both do not allow you to change the contents of a TextBox. What is the difference between these?
3. How can you create a form with no maximize and/or minimize button?
4. Write a windows form application to count the number of vowels in a textbox. The application should have a TextBox and a Label. The Label should be updated as the user types in the TextBox and should display the number of vowels in the text in the TextBox. The application should look like this:



Solution of Last Issue's Exercise (Exercise 10)

1. What are delegates? How they are represented in .NET?

Delegates are reference types derived from System.Delegate. An instance of a delegate can reference (and thus call) methods with a particular signature irrespective of their names. Delegates can be multicast, i.e., they may have more than one method in their invocation list. Multicast delegates are derived from the System.MulticastDelegate type, which itself is a sub-type of System.Delegate.

2. What is the difference between the delegate type and the delegate reference?

The delegate type is the specific signature of a method that can be referenced by the delegate reference. Many delegate references can be of the same delegate type just as many object references of a particular class can exist.

3. What is the significance of multicast delegates?

Multicast delegates may have more than one method in their invocation list. Multicast

delegate can be used to send a single piece of information to multiple clients through their respective method invocation. Multicast delegates are used in the event handling mechanism in the .NET environment. Multicast delegates provide a clean interface to create the Publisher-Subscriber scheme. Many subscribers may subscribe (or register) to a single publisher by adding their respective methods to the invocation list of the publisher's multicast delegate.

4. How are events presented in VB.NET?

Events in VB.NET are presented in the form of multicast delegates. An event generator class defines and generates (raises) an event which can be handled by a number of event handlers in event consuming classes.

5. Events are implemented in VB.NET using multicast delegates. The multicast delegates must have no return type. This means that events can't return anything. Is this a weak point of the event handling mechanism in VB.NET?

No! Events don't need to return anything logically. Events are means of sending certain information to the event handlers. The event handlers are receivers of information and they don't need to return anything. It's like a television broadcast; a number of television sets may be interested in receiving a transmission, but none actually needs to send anything in return.

What's Next

- Next time we will look at more Windows Controls and Standard Dialog Boxes. We will explore:
- Collection Controls such as the List Box, Combo Box, Tree View and List View
- Main menu, Image List, Toolbar and the DateTime Picker
- OpenFile, SaveFile, Font & Color Dialog Boxes

More Windows Controls and Standard Dialog Boxes

Lesson Plan

In this installment we will learn about some more windows controls and standard dialog boxes. We will start out by looking at the collection controls, such as the List box, Combo box, Tree View and the List View control. Later we will learn about other common controls, including the main menu, image list, Toolbar and the Date Time Picker. Finally we will explore some of the standard dialog boxes, e.g. the Open File, Save File, Font and Color Dialog Boxes.

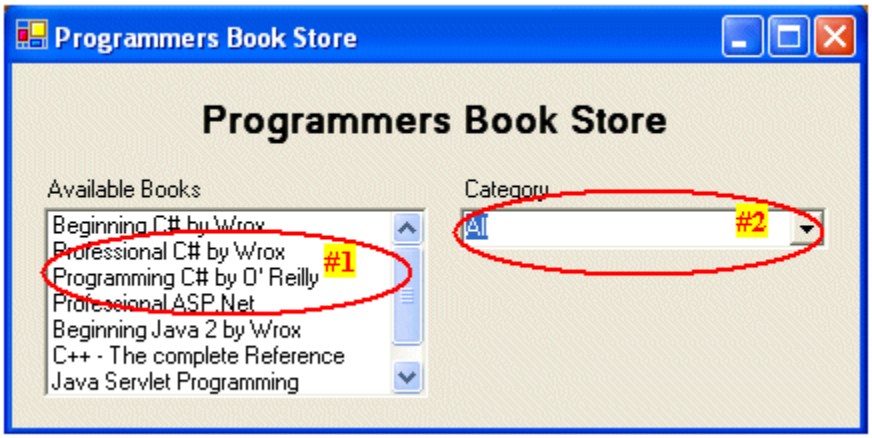
Collection Controls

Some of the windows controls contain some types of collection, such as names of books and images in a folder. These are called Collection controls. Examples of collection controls include the List Box, Combo Box, Tree View, List View, Toolbar, Image List and Main Menu. In this lesson we will explore these controls one by one.

List Box Control

A list box control contains a list of items which can be selected by the user. A list box can be set to allow the user to select one or more items. In .NET the list box is represented by the System.Windows.Forms.ListBox class. Each list box instance contains a collection called 'Items' that holds the items present in the list. An item (usually a string) can be added or removed from the list box.

The following screen shot presents a window containing a list box (#1) and a combo box (#2).



A list box can be added to the form from the Visual Studio.NET toolbar. Usually a label is also added above the list box to serve as the title of the list box (e.g., The label 'Available Books' in the above picture). The Name property of the list box is used to set the name of the list box object in the code. The System.Windows.Forms.ListBox contains a property

called 'Items' which provides access to the items contained in the list box. The Items collection can be used to read, add and remove items from the list box.

Adding items to the list box

A list box can be populated either at design time using the Visual Studio IDE or at runtime using code. To add items to the list box using the Visual Studio IDE, select the list box control in the designer and in the properties window click the Items property. It will show you an interface where you can enter the list of items to be added to the list box. Alternatively, you can write code to add items to the list box. An item can be added to the list box using the Add() method from the Items collection of the ListBox object. Assume that we have made a list box to store the names of books and that we have set the Name property of the list box to 'lbxBooks'. The following code can be used to add items to the list box.

```
        lbxBooks.Items.Add("Programming C#")
        lbxBooks.Items.Add("Professional C#")
```

Or if you want to add a series of items, you can use the AddRange() method of the Items collection

```
        lbxBooks.Items.AddRange(New String() _
                                {"Beginning C# by Wrox", _
                                 "Professional C# by Wrox", _
                                 "Programming C# by O' Reilly", _
                                 "Professional ASP.Net", _
                                 "Beginning Java 2 by Wrox", _
                                 "C++ - The complete Reference", _
                                 "Java Servlets Programming", _
                                 "Java Server Pages - JSP)"})
```

Accessing items in the list box

The Items collection of the list box provides the default property that allows items in the list box to be accessed programmatically (through code).

```
        lbxBooks.Items(0) = "Program Development in Java"
                                        ' changing list box item
        Dim book1 as String = CStr(lbxBooks.Items(1))
                                        ' reading list box item
```

In the above code, the first statement uses the default property of the Items collection to change a list box item and the second statement reads an item in the list box. Note that we have applied the cast to the list box item as the return type of the Items indexer (Items()) is object. Remember you will only need this cast if you are using the Option Strict Option 'On' which we prefer to use.

You may also see at run time, the currently selected item using the SelectedItem property of the list box.

```
        MessageBox.Show(lbxBooks.SelectedItem.ToString())
```

Removing items from the list box

Individual items can be removed from the list box either by calling the Remove() or RemoveAt() method of the Items collection of the list box. The Remove() method accepts an object to be removed from the list box as

```
        lbxBooks.Items.Remove("Programming C#")
```

The above line will remove the item 'Programming C#' from the list box. You can also use the RemoveAt() method which accepts the index of the item to be removed from the list box.

```
        lbxBooks.Items.RemoveAt(0)
```

This line will remove the element at index 0 (the first element) from the list box. Finally, to remove all the items contained in a list box, you can call the Clear() method.

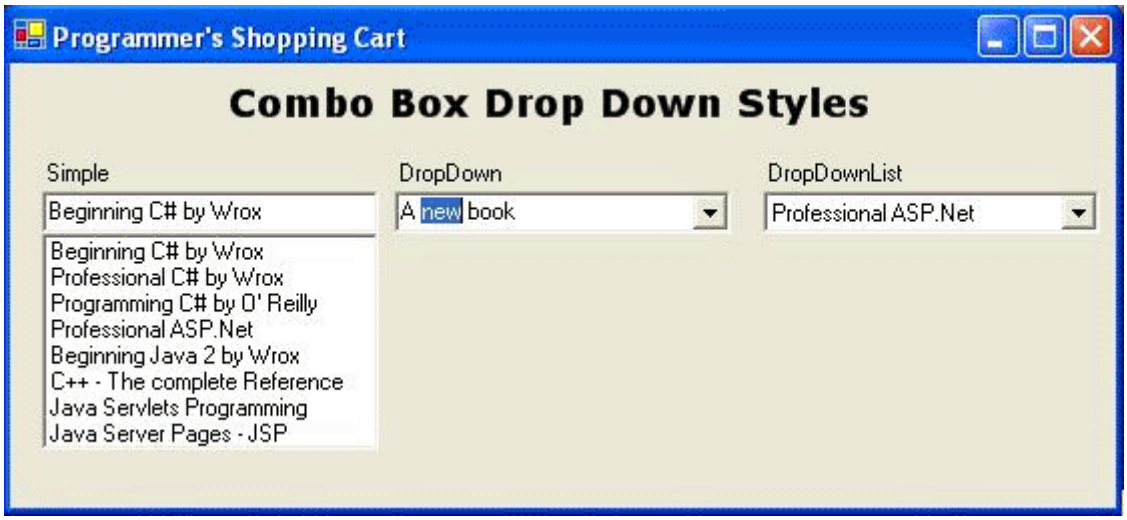
```
        lbxBooks.Items.Clear()
```

List Box Events

The most important and frequently used event of the list box is SelectedIndexChanged. This event is triggered when an item is selected in the list box. To add an event handler for this event, double click the list box in the form designer. The following code will show the selected item in the message box whenever the selection in the list box is changed.

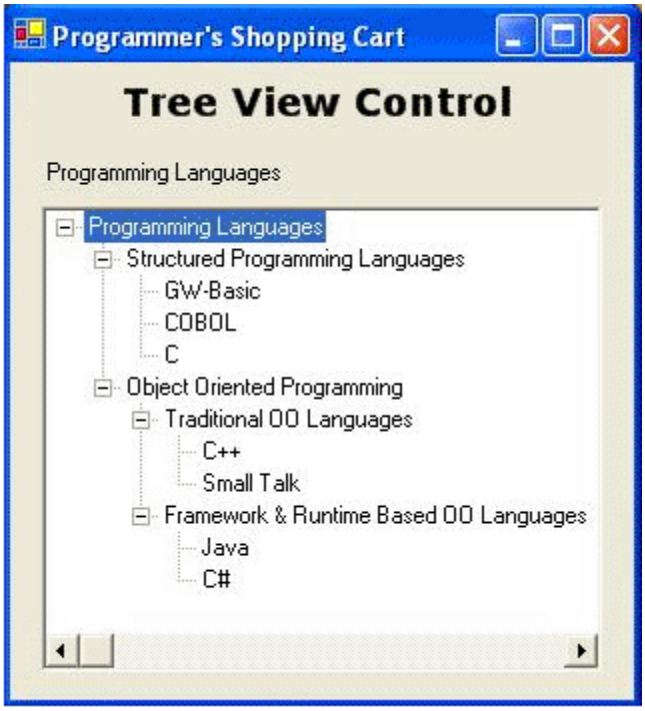
```
    Private Sub lbxBooks_SelectedIndexChanged(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles lbxBooks.SelectedIndexChanged
       MessageBox.Show("The selected item is " + _
       lbxBooks.SelectedItem.ToString(), "Selected Item")
    End Sub
```

Combo Box Control
The combo box is similar to the list box in that it is used to display a list of items. The combo box is presented in .NET through the System.Windows.Forms.ComboBox class. The combo box has three visual designs which can be toggled using the ComboBox class' 'DropDownStyle' property. The three designs are named 'Simple', 'DropDown' and 'DropDownList'. The following screen shot demonstrates these three drop down styles.

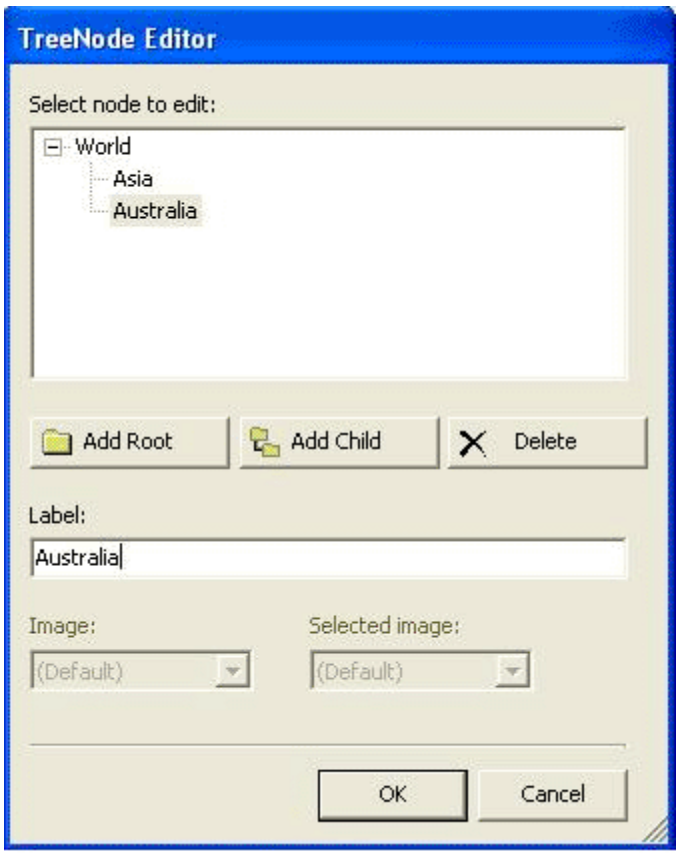


The **Simple** style displays the list of items and the selected item at the same time in separate areas. The simple drop down style combo box is typically used in the font dialog box (discussed later in the lesson). The **DropDown** style shows the items in a drop down list. The user can write a new item in its text area. The **DropDownList** style is similar to the drop down style but here the user can only select one of the available choices and can not type in a new item. Items can be inserted, removed and accessed in just the same way we demonstrated with the listbox control.

Tree View
The Tree View Control is used for hierarchical representations of items. It is represented in .NET by the System.Windows.Forms.TreeView class. The following screen shot gives an example of how the tree view control can look.



The TreeNode Editor
The easiest way to add and remove items to and from the tree view control is through its 'Nodes' property in the form designer. When you click the Nodes property in the properties windows of the form designer, it will show the following screen, which is known as the Tree Node Editor.



'Add Root' will add a root element (one with no parent e.g., World in the above window). 'Add Child' will add a child node to the selected element. The label of the nodes can be changed using the text box labeled 'Label'. A node can be deleted using the 'Delete' button.

Adding/Removing items at runtime

Items can be added to or removed from the tree view using the TreeNode editor at design time. However most of the time we need to add/remove items at runtime using our code. The TreeView class contains a property called 'Nodes' which provides access to the individual nodes of the control. The Add() method of the Nodes collection can be used to add a text item or a TreeNode object which itself may have child tree nodes. The following code adds a sample tree hierarchy on the form's Load event.

```
Private Sub MyForm_Load(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load
        treeView1.Nodes.Clear()
        treeView1.Nodes.Add("Programming Languages")
        Dim node As New TreeNode("Object Oriented Programming Languages")
        node.Nodes.Add("C++")
        Dim subnode As TreeNode = node.Nodes.Add("Framework and runtime based
languages")
        subnode.Nodes.Add("Java")
        subnode.Nodes.Add("C#");
        treeView1.Nodes(0).Nodes.Add(node)
End Sub
```

We have added a root node labeled 'Programming Languages'. We then created a new TreeNode and added sub nodes to it. Finally we added this node to the root node. When we execute the program, the following screen is produced.

Tree View Events

Tree View has a number of important events, some of which are listed below.

Event	Description
AfterSelect	Fired when an item (node) is selected in the tree view control. TreeViewEventArgs are passed with this event which contain 1) TreeViewAction enumeration which describes the action caused the selection such as ByKeyboard, ByMouse, Collapse, etc 2) The Node object which represents the selected node. The selected node can also be accessed using the SelectedNode property of the tree view control
BeforeExpand	Fired just before the node is expanded
BeforeCollapse	Fired just before the node is collapsed
AfterExpand	Fired just after the node is expanded
AfterCollapse	Fired just after the node is collapsed
BeforeLabelEdit	Fired just before an attempt is made to edit the Label of the node. You need to set the LabelEdit property of the tree view to true if you wish to allow your user change the label of a node
AfterLabelEdit	Fired just after the label of a node has been edited.

The following event handler will show the label of the node in a Message box whenever it is selected.

```
    Private Sub TreeView1_AfterSelect(ByVal sender As System.Object,
_
                ByVal e As System.Windows.Forms.TreeViewEventArgs) _
                Handles TreeView1.AfterSelect
        MessageBox.Show("'" + e.Node.Text + "' node selected", "Selected Node")
    End Sub
```

The program will look like this when run:

Image List Control

The image list is an invisible control. It is merely used to store images to be used in other controls such as the tree view or list view controls. An image list (like a main menu, context menu, toolbar or standard dialog) is added in your program as a resource and does not have any graphical presentation in your application. It is represented in .NET by the System.Windows.Forms.ImageList class.

When you select the image list control from the Visual Studio tool box and place it on the form, it is literally added as a resource and is displayed below the form in the designer as an icon. You can select it from there and change the necessary properties.

The fundamental property of the image list is its Name. An Image List has relatively few properties. The most important is the 'Images' collection which holds the images stored in the image list. When you click the Images property of the Image List in Visual Studio's designer, it shows an image collection editor which allows you to add different .bmp, .jpg or .gif pictures to your image list collection. The 'ImageSize' property represents the size of the images in the list. The default is 16, 16 in my Visual Studio when using a screen resolution of 800 x 600.

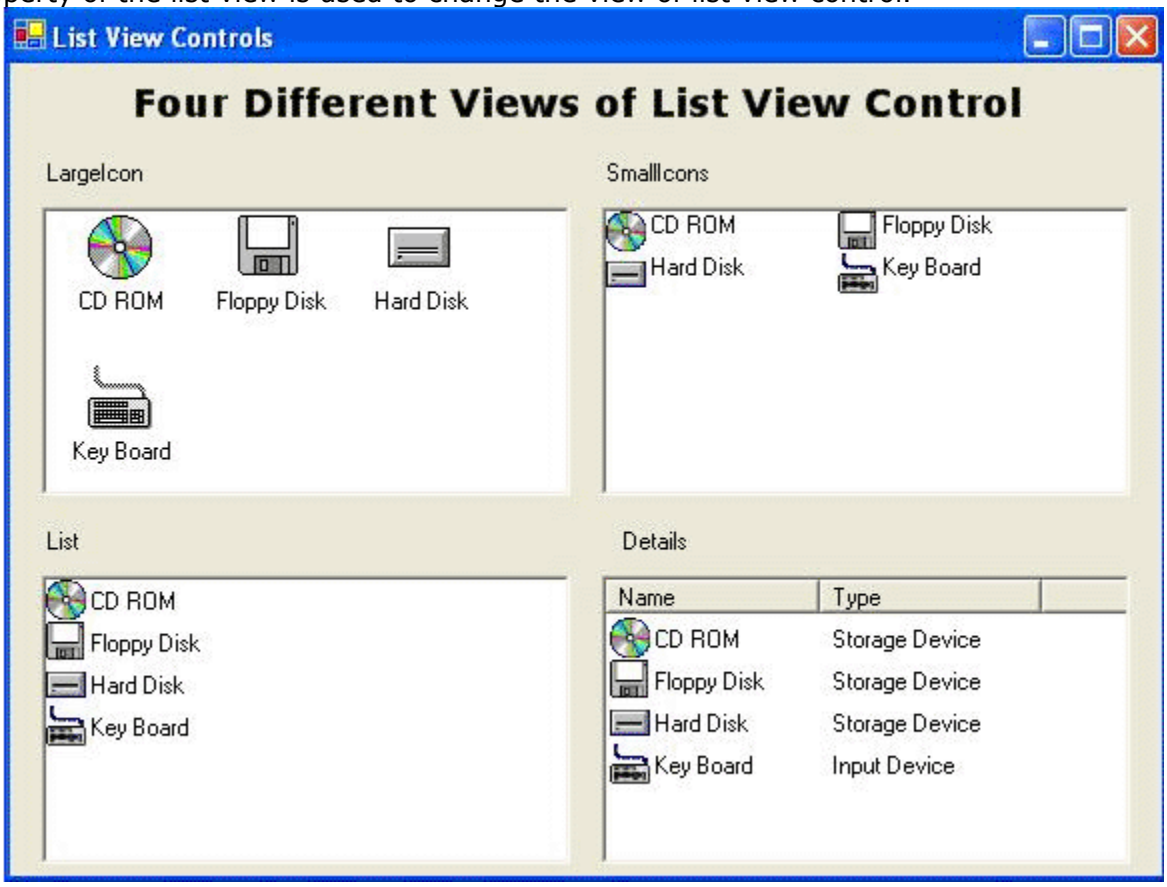
Attaching an Image List to different controls

An image list is attached to different controls, e.g. a tree view, list view or toolbar. The tree

view and tool bar controls have a property called ImageList which is used to attach an image list to them. The individual images are attached to individual nodes or buttons when adding these. For example, the Add() method of the Nodes collection of the tree view control has an overloaded version which accepts the text for the node and the index of the image in the attached image list.

List View Control

The List View control is a very important and interesting control. It is used to hold a list of items and can display them in four different views; Large Icons, Small Icons, List and Details. The all famous 'Windows Explorer' uses the list view control to show different icons. A list view is represented in .NET by the System.Windows.Forms.ListView class. The following screen shot demonstrates four different views of the list view controls. The 'View' property of the list view is used to change the view of list view control.

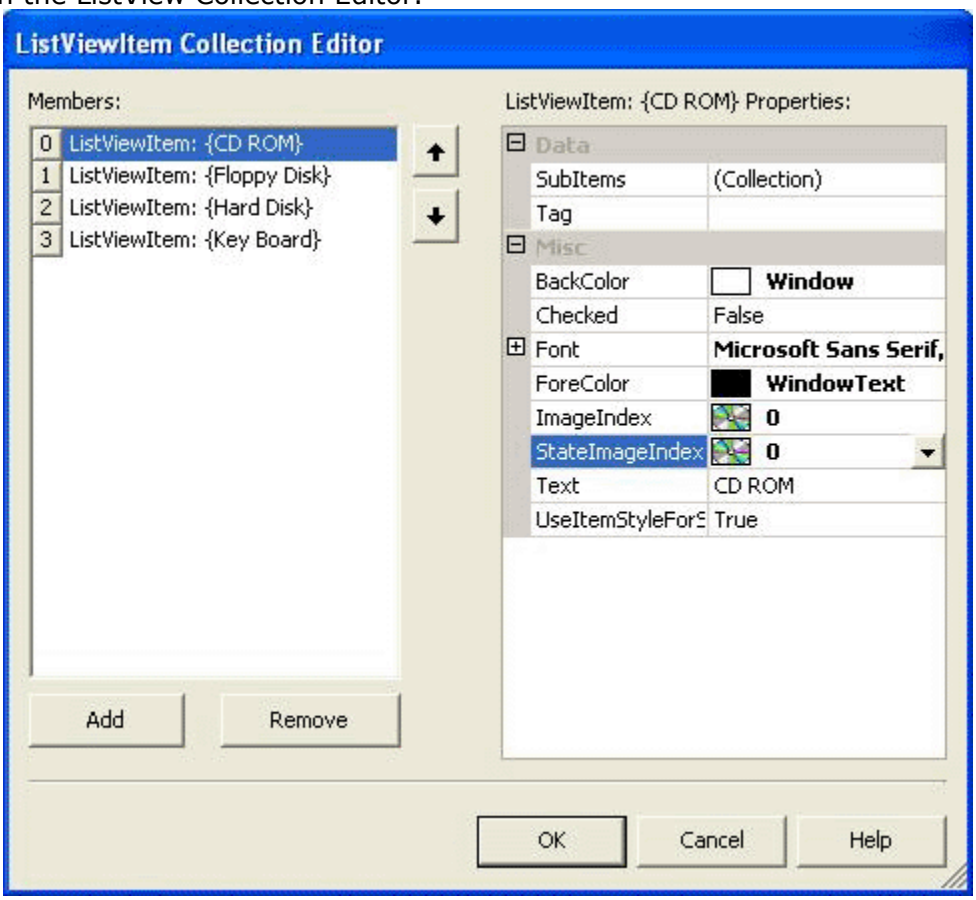


Two Image Lists in the List View Control

Two image lists can be added to a list view control. One is called the LargeImageList and its images are used for the icons in large icon view. The other image list is called SmallImageList and its images are used for the icons when small, list and detail view. Usually the two image lists are the same with the only difference being in their ImageSize property. For example in the above screen shot, the size of images in the large image list is 32x32 while the size of images in the small image list is 20x20 and we have used the same images for the two image lists; in fact we just copy-paste one image list and changed the size of it.

Adding items to the list view control using the designer

Visual studio provides an easy way to add items to the list view control. To add items to the list view control, simply click the 'Items' property of the list view in the properties window. It will open the ListView Collection Editor.



Here you can use the Add button to add items to the list. Each item has a Text property which represents the text displayed with each item. Each item also has the ImageIndex property which represents the image to be attached with the item. The images are loaded from the attached image list (An image list can be attached using LargeImageList or SmallImageList property of the list view control as described earlier). If a multi-column list view is used (which we will describe later in the lesson), The SubItems property can be used to add the sub items of an item. Similarly, items can be removed using the Remove Button.

Adding Items at runtime using code

The ListView control has a property called Items which stores the collection of ListViewItem to be stored in the list view control. Items can be added or removed using its Add() and Remove() methods. The Add() method has three overloaded versions. One takes only a string which is used to represent the text of the item. The second one takes a string and an image list index. The string is used for the text label and the image index is used to attach the particular image from the attached image list to this item. The third one takes the ListViewItem object. The following code adds some items to the list view

```
listView1.Items.Add("Disk")                ' text label is passed
listView1.Items.Add("Disk", 0);    ' text label and image index is passed
Dim item As new ListViewItem("Disk", 2)
                ' a new ListViewItem object is created
listView1.Items.Add(item)
                ' and added to the list view control
```

Events for List View Control

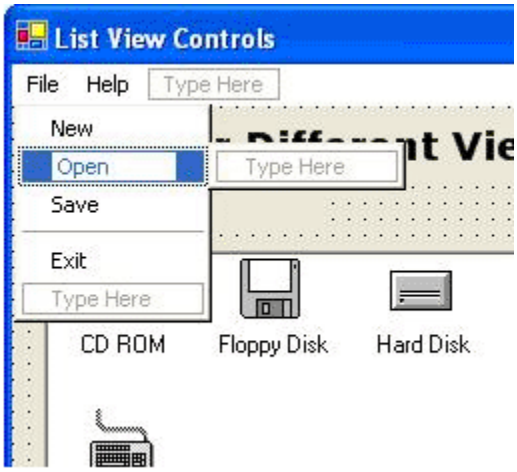
The most frequently used event in the list view control is SelectedIndexChanged event which is triggered when the selection in the list view is changed. The following event handler prints the selected item's text in the message box whenever the selection in the list view changes.

```
    Private Sub ListView1_SelectedIndexChanged(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles ListView1.SelectedIndexChanged
        Dim item As ListViewItem
        For Each item In ListView1.SelectedItems
            MessageBox.Show(item.Text)
        Next
    End Sub
```

The complete source code of the program can be downloaded by clicking here.

Main Menu

The Main menu is also added to the form as a resource. The Main menu is represented in .NET by the System.Windows.Forms.MainMenu class. You can just pick and place the Main Menu control from the visual studio toolbox on to the form. As a result Visual studio will show an empty main menu at the top of the form. Change the name of the main menu and add options in the menu by clicking it.



Important Points about Main menu

- You can change the name and text of the main menu items from the properties window while selecting it in the designer.
- You can also apply shortcut keys (like Ctrl+t or Alt+F5) for the menu items using the properties window.
- You can make a dashed line (just like the one between Save and Exit option in the above screen shot) in the menu by writing only dash - in the menu item text.
- You can add a checkbox before a menu item using its RadioCheck property.
- You can add an event handler for the menu item just by double clicking it in the designer

Tool Bar

The Toolbar is also added like other resources (menu, image list, etc). It is represented in .NET by the System.Windows.Forms.ToolBar class. The ToolBar class contains a collection called Buttons which can be used to add/remove items from the toolbar. An Image List can also be attached to the toolbar. When you click the Buttons property in the properties window, Visual Studio presents you the familiar interface to add and remove buttons. Event handlers can be added for the toolbar just by double clicking the button in the designer.

Date Time Picker

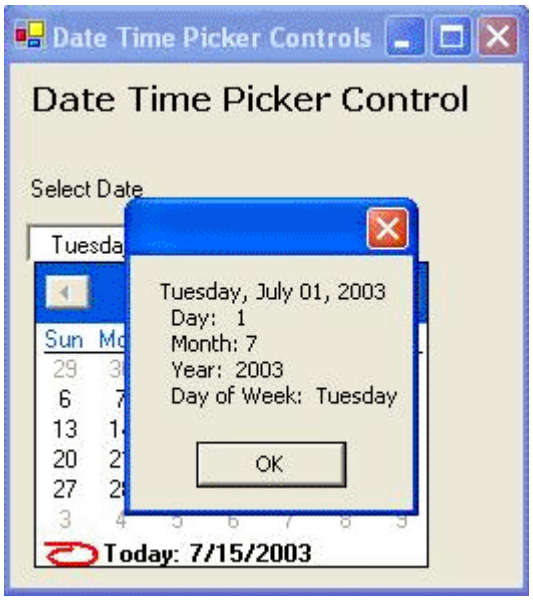
The Date Time Picker control is commonly used in the windows environment to allow the user select a particular date. In .NET it is represented by the System.Windows.Forms.DateTimePicker class. It can be selected from the toolbox and placed on the form in the designer. Usually we only change its Name property. The following screen shot shows a form containing the date time picker control.

The most frequently used event for the date time picker is ValueChanged. It is triggered when the user selects a new date from the control. The following event handler uses this event to print the value of selected date in a message box

```
Private Sub dtpSelectedDate_ValueChanged(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles dtpSelectedDate.ValueChanged
```

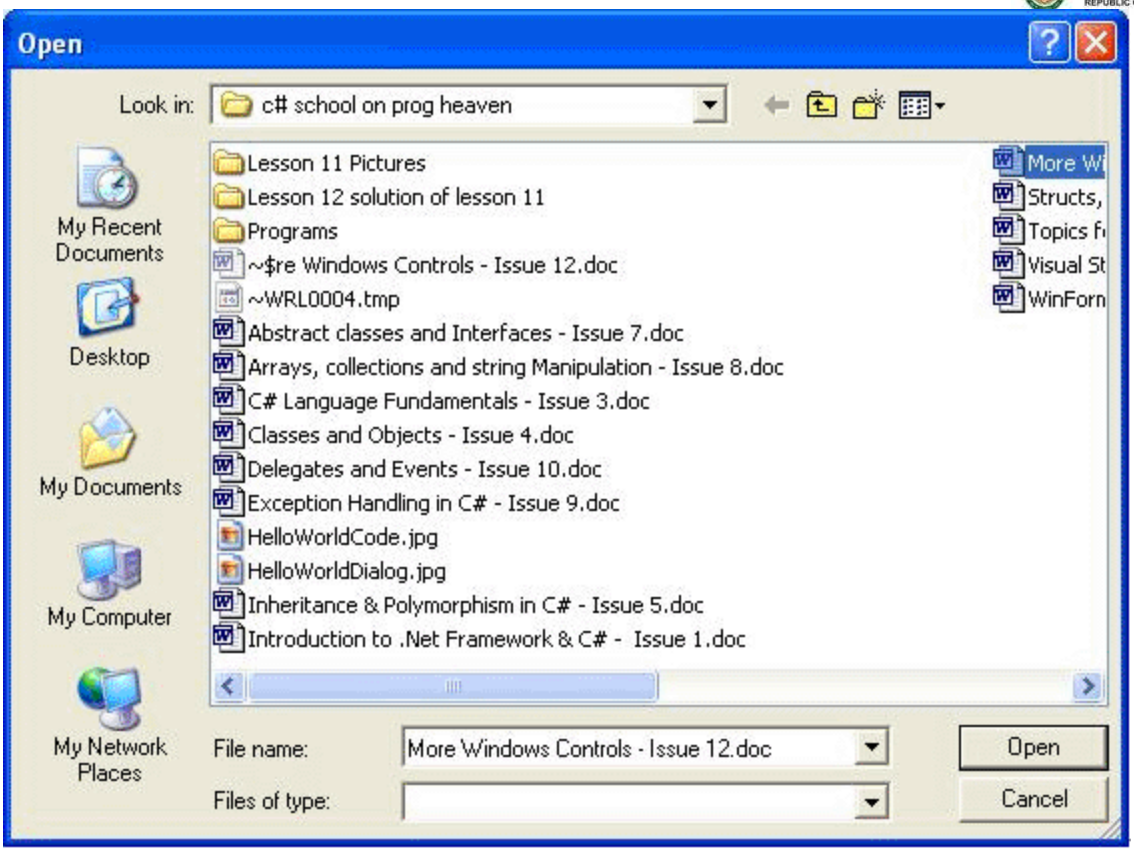
```
        Dim msg As String = dtpSelectDate.Text
        msg += vbcrlf + "  Day:    " + dtpSelectDate.Value.Day.ToString()
        msg += vbcrlf + "  Month: " + dtpSelectDate.Value.Month.ToString()
        msg += vbcrlf + "  Year:   " + dtpSelectDate.Value.Year.ToString()
        msg += vbcrlf + "  Day of Week:   " + dtpSelectDate.Value.DayOfWeek.ToString()
        MessageBox.Show(msg)
End Sub
```

In the code above, we have separately collected and printed the day, month, year and the day of the week in a message box. When the program is executed the following output is shown.



Windows Standard Dialog Boxes
Windows allows developers to provide some commonly used dialog boxes in their applications. They include the Open File, Save File, Font Settings, Color Selection and Print dialog boxes. We will discuss the first four of these dialogs here. The dialog boxes are also added to the form as a resource which means they don't have any permanent visual representation and pop up only when needed by the developer and the application.

Open File Dialog Box
This is the common dialog box presented in the Windows environment for opening files. It is an instance of System.Windows.Forms.OpenFileDialogBox. The following screen shot shows the common open file dialog.



The important properties include:

Property	Description
DefaultExt	The default extention for opening files. It uses a wild card technique for filtering file names. If the value of DefaultExt is *.doc only do then files with extention 'doc' will be visible in the open file dialog
FileName	The full path and file name of the selected file
InitialDirectory	The initial directory (folder) to be opened in the dialog
MultiSelect	Boolean property. Represents whether multiple file selection is allowed or not
DialogResult	DialogResult enumeration that shows whether user has selected the OK or the Cancel button to close the dialog box

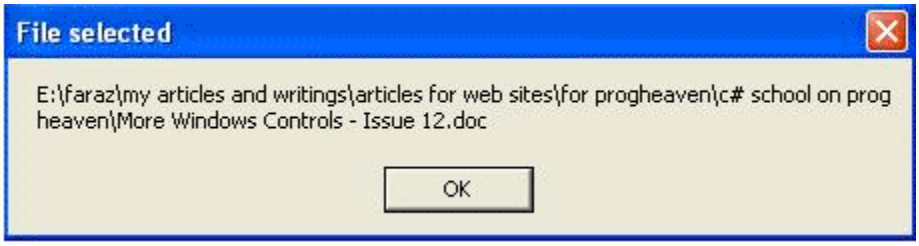
Using the Open File Dialog Box

Usually the open file dialog box is presented on the screen when a button is pressed or a menu item is selected. The following button event handler presents the open file dialog box and prints the name of the file selected in a message box.

```
Private Sub btnOpenFile_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles btnOpenFile.Click
Dim res As DialogResult = openFileDialog.ShowDialog()
If res = DialogResult.OK Then
    MessageBox.Show(openFileDialog.FileName, "File selected")
End If
```

```
End Sub
```

Here we first presented the dialog box on the screen using its ShowDialog() method. This method presents the dialog box on screen and returns the DialogResult enumeration which represents how the user closes the dialog box. In the next line we check whether the user the pressed the OK button to close the dialog box. If yes then we printed the name of the selected file in the message box. When the above code is executed, it shows the following result.



It is clear from the above message box that the FileName property returns the complete path of the selected file.

Save File Dialog Box

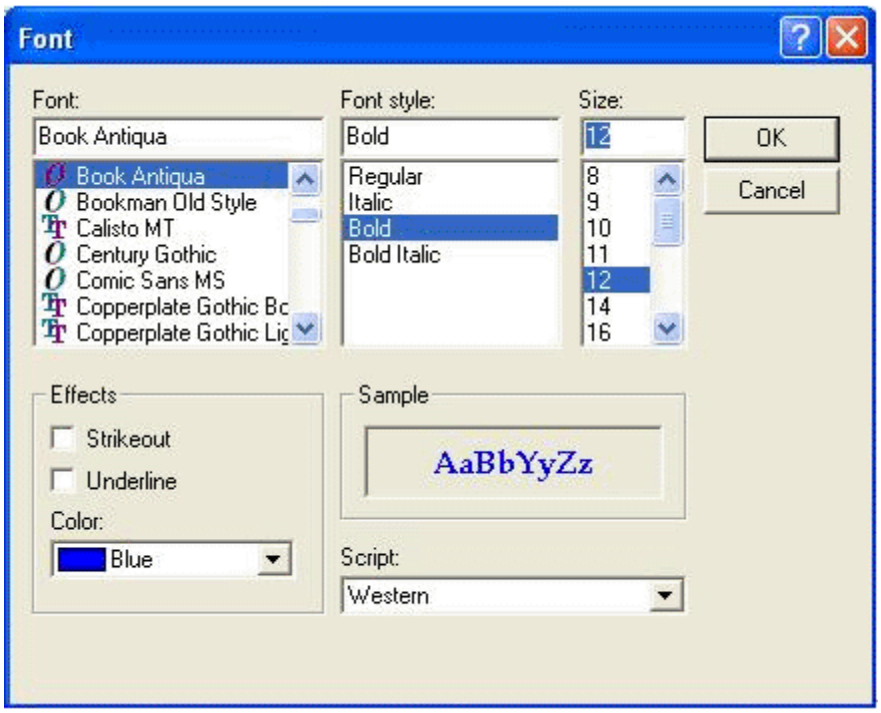
The save file dialog box is used to allow the user to select the destination and name of the file to be saved. It is an instance of System.Windows.Forms.SaveFileDialog. It is very similar to the open file dialog box. The following code will show the save file dialog box on the screen and print the name of the file to be saved in the message box

```
Private Sub btnSaveFile_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles btnSaveFile.Click
Dim res As DialogResult = saveFileDialog.ShowDialog()
If res = DialogResult.OK Then
    MessageBox.Show(saveFileDialog.FileName, "File saved")
End If
End Sub
```

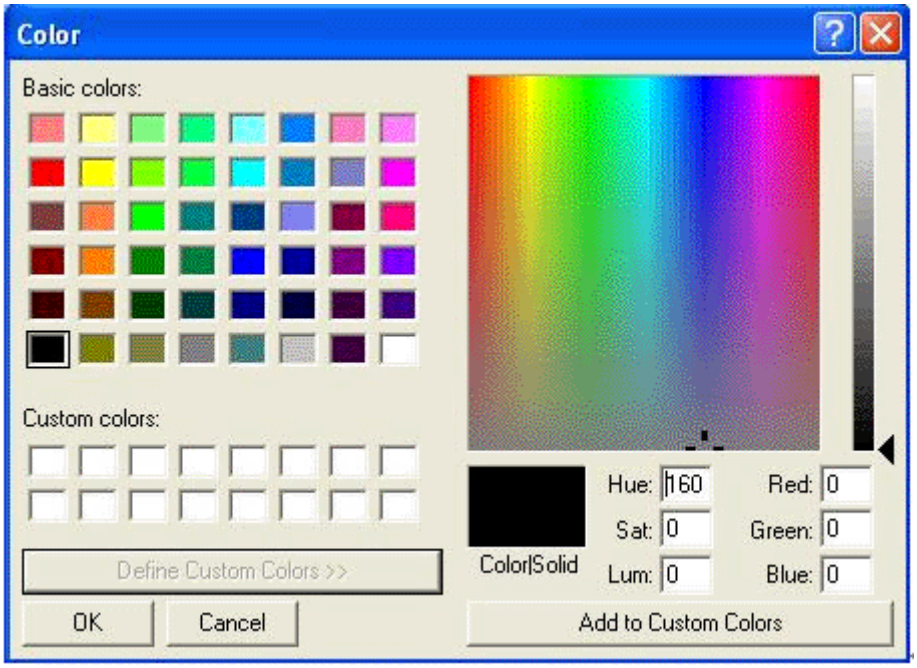
The code above will show the following message box:

Font and Color Dialog Boxes

The font dialog box is used to allow the user to select font settings. You have seen this dialog box in Microsoft Word Pad, Notepad and MSN Messenger. It is an instance of System.Windows.Forms.FontDialog. The color dialog box is used to allow the user to select a color. You might have seen this dialog box in Microsoft Paint perhaps. It is an instance of System.Windows.Forms.ColorDialog. The font dialog box looks like this:



And the color dialog box looks like this:



Lets make an application which presents a form with a text box to write comments. It will allow the user to change the font and color of the text in the text box. The form looks like this:

The form also contains two dialog boxes. One is a fontDialog to change the font and the other is a colorDialog to change the color. The event handler for two buttons are as follows:

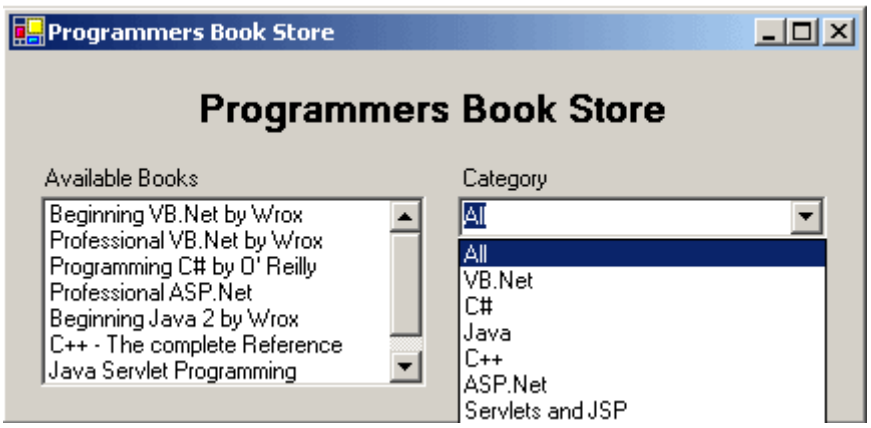
```
Private Sub btnChangeFont_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles btnChangeFont.Click
Dim res As DialogResult = fontDialog.ShowDialog()
If res = DialogResult.OK Then
    txtComment.Font = fontDialog.Font
    txtComment.ForeColor = fontDialog.Color
End If
End Sub
Private Sub btnChangeColor_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles btnChangeColor.Click
Dim res As DialogResult = fontDialog.ShowDialog()
If res = DialogResult.OK Then
     txtComment.ForeColor = colorDialog.Color
End If
End Sub
```

The change font button event handler presents the font dialog and sets the font and color of the text of the text box to the selected font and color. Note that we have set the ShowColor property of the font dialog box to true, which is false by default.

In the change color button's event handler, the program presents the color dialog box and sets the color of the text in the text box to the selected one.

Food for thought: Exercise 12

1. Write an application that presents the user with a list of books in a list box and ask them to select a category in a combo box. When the user selects a category, filter the list and show only the books that belong to the selected category. The program should look like this:



2. Write a program that shows the folder hierarchy of the C drive of your system in a tree view. Use the System.IO namespace to get files and folders of your system. The Directory class has a method named GetDirectories() which returns a list of directories in a particular path.
3. Write a program that shows the files in the windows folder of your system in a list view. Use the System.IO namespace to get files and folders of your system. The Directory class has a method GetFiles() which returns the list of files in a particular path.
4. Extend the program in question 3 and add an icon with each file name. For convenience, you can use the same icon for all files. Provide a toolbar that allows user to change the view of the list view control (i.e., Large Icons, Small Icons, List, Details).
5. Extend the program in question 4 and provide the option of sorting in the toolbar. The options should include Sort by name and sort by extention (Sort by type). Also provide the toolbar options in the main menu.

Solution to Last Issue's Exercise (Exercise 11).

1. How does .NET support windows applications?

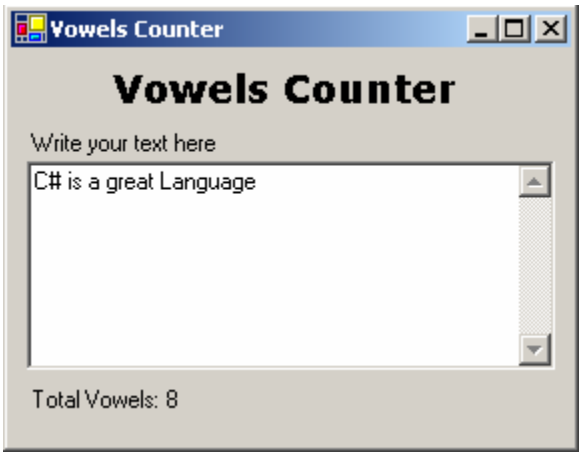
Microsoft .NET (and hence VB.NET) supports the development of windows applications in a pure object orientated style. The System.Windows.Forms namespace contains a number of classes, interfaces and enumerations for windows application development. The presentation of windows controls in an object oriented way makes it easier to use the existing controls, define new controls and to extend the behavior of existing controls.

2. The TextBox control has two Boolean properties ReadOnly and Enabled. Both do not allow you to change the contents of a TextBox. What is the difference between these?

When set to false, the Enabled property actually disables the text box which means you can not select, copy and modify the text. On the other hand the ReadOnly property, when set to true, makes the text box read only; which means you can not modify the text in the text box but you can select and copy the text present in the text box to the windows clipboard and then paste it to any other window.

3. How can you create a form with no maximize and/or minimize button?
The base class System.Windows.Forms.Form has two Boolean properties; MaximizeBox and MinimizeBox. A form with no maximize and/or minimize box can be created by setting these two properties to False.

4. Write a windows form application to count the number of vowels in a textbox. The application should have a TextBox and a Label. The Label should be updated as the user types the text in the TextBox and should display the number of vowels in the text in the TextBox. The application should look like



The complete code can be downloaded here.

What's Next...

- Next time we will explore data access with ADO.NET. We will cover:
- ADO.NET Introduction
- Different components of ADO.NET
- Connection, Command, DataSet, DataTable, DataRow, DataColumn, Relationship, DataReader
- Review of basic SQL queries
- Performing common data access tasks with ADO.NET
- Demonstration example
- Using Stored Procedure with ADO.NET
- Using Data Grid control
- Supplementary Topic: Working under connected environment

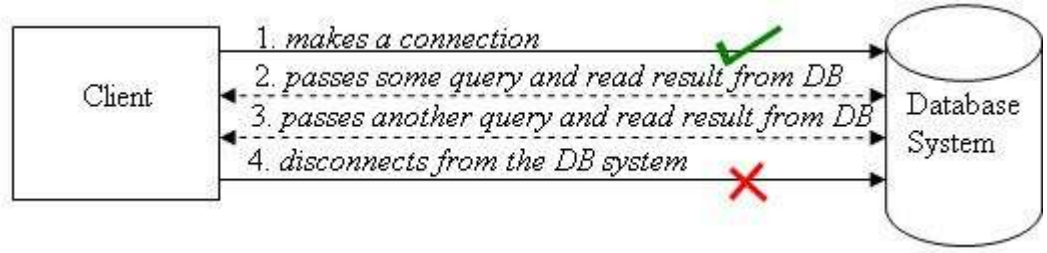
Data Access in .Net using ADO.Net
Lesson Plan

In this lesson we will learn how VB.NET applications can interact with database systems. We will start out by looking at the architecture of ADO.NET and its different components. Later we will demonstrate data access in .NET through an application. Finally we will learn about stored procedures and explore the Data Grid control which is commonly used for viewing data.

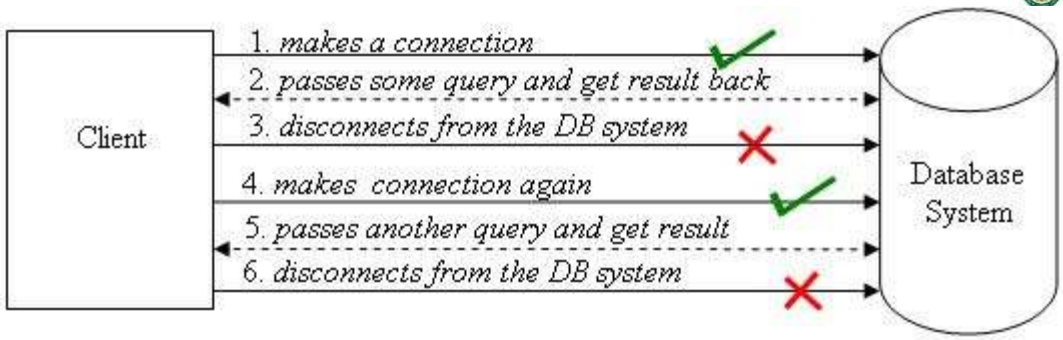
Introducing ADO.NET

Most of today's applications need to interact with database systems to persist, edit or view data. In .NET, data access services are provided through ADO.NET components. ADO.NET is an object oriented framework that allows you to interact with database systems. We usually interact with database systems through SQL queries or stored procedures. The best thing about ADO.NET is that it is extremely flexible and efficient. ADO.NET also introduces the concept of disconnected data architecture. In traditional data access components, you made a connection to the database system and then interacted with it through SQL queries using the connection. The application stays connected to the DB system even when it is not using DB services. This commonly wastes valuable and expensive database resources, as most of the time applications only query and view the persistent data. ADO.NET solves this problem by managing a local buffer of persistent data called a data set. Your application automatically connects to the database server when it needs to run a query and then disconnects immediately after getting the result back and storing it in the dataset. This design of ADO.NET is called disconnected data architecture and is very much similar to the connectionless services of HTTP on the internet. It should be noted that ADO.NET also provides connection oriented traditional data access services.

Traditional Data Access Architecture



ADO.NET Disconnected Data Access Architecture



Another important aspect of disconnected architecture is that it maintains a local repository of data in the dataset object. The dataset object stores the tables, their relationships and their different constraints. The user can perform operations like update, insert and delete on this local dataset. The changes made to the dataset are applied to the actual database as a batch when needed. This greatly reduces network traffic and results in better performance.

Different components of ADO.NET

Before going into the details of implementing data access applications using ADO.NET, it is important to understand its different supporting components or classes. All of the generic classes for data access are contained in the System.Data namespace.

Class	Description
DataSet	The DataSet is a local buffer of tables or a collection of disconnected recordsets
DataTable	A DataTable is used to contain data in tabular form using rows and columns.
DataRow	Represents a single record or row in a DataTable
DataColumn	Represents a column or field of a DataTable
DataRelation	Represents the relationship between different tables in a DataSet.
Constraint	Represents the constraints or limitations that apply to a particular field or column.

ADO.NET also contains some database specific classes. This means that different database system providers may provide classes (or drivers) optimized for their particular database system. Microsoft itself has provided the specialized and optimized classes for their SQL server database system. The names of these classes start with 'Sql' and are contained in the System.Data.SqlClient namespace. Similarly, Oracle has also provides its classes (drivers) optimized for the Oracle DB System. Microsoft has also provided the general classes which can connect your application to any OLE supported database server. The name of these classes start with 'OleDb' and these are contained in the System.Data.OleDb namespace. In fact, you can use OleDb classes to connect to SQL server or Oracle database; using the database specific classes generally provides optimized performance however.

Class	Description
SqlConnection, OleDbConnection	Represents a connection to the database system
SqlCommand, OleDbCommand	Represents SQL a query
SqlDataAdapter, OleDbDataAdapter	A class that connects to the database system, fetches the records and fills the DataSet
SqlDataReader, OleDbDataReader	A stream that reads data from the database in a connected design
SqlParameter, OleDbParameter	Represents a parameter to a stored procedure

A review of basic SQL queries
Here we present a brief review of four basic SQL queries.

SQL SELECT Statement
This query is used to select certain columns of certain records from one or more database tables.

```
SELECT * from emp
```
selects all the fields of all the records from the table named 'emp'

```
SELECT empno, ename from emp
```
selects the fields empno and ename for all of the records from the table named 'emp'

```
SELECT * from emp where empno < 100
```
selects all records from the table named 'emp' where the value of the field empno is less than 100

```
SELECT * from article, author where article.authorId = author.authorId
```
selects all records from the tables named 'article' and 'author' that have the same value of the field authorId

SQL INSERT Statement
This query is used to insert a record into a database table.

```
INSERT INTO emp(empno, ename) values(101, 'John Guttag')
```
inserts a record in to the emp table and sets its empno field to 101 and its ename field to 'John Guttag'

SQL UPDATE Statement
This query is used to modify existing records in a database table.

```
UPDATE emp SET ename = 'Eric Gamma' WHERE empno = 101
```
updates the record whose empno field is 101 by setting its ename field to 'Eric Gamma'

SQL DELETE Statement
This query is used to delete existing record(s) from a database table.

```
DELETE FROM emp WHERE empno = 101
```

deletes the record whose empno field is 101 from the emp table

- Note that its not good practice to allow users to actually delete records from your database. This is open to abuse an human error. A more safer method is to flag a field with an end date. I.e When a user "deletes" a record, what really happens is this.

```
UPDATE emp SET enddate = GetNow(date) WHERE empno = 101
```

To remove this record from the users reach in future quieries.

```
Select * FROM emp WHERE enddate = Null
```

Performing common data access tasks with ADO.NET

Enough review and introduction! Let's start something practical. Now we will build an application to demonstrate how common data access tasks are performed using ADO.NET.

We will use MS SQL server and MS Access database systems to perform the data access tasks. SQL Server is used because probably most of the time you will be using MS SQL server when developing .NET applications (And theres a free cut down version available from Microsoft). For SQL server, we will be using classes from the System.Data.SqlClient namespace. Access is used to demonstrate the OleDb databases. For Access we will be using classes from the System.Data.OleDb namespace. In fact, there is nothing different in these two approaches for developers and only two or three statements will be different in both cases. We will highlight the specific statements for these two using comments like:

```
' For SQL server
Dim dataAdapter As New SqlDataAdapter(commandString, conn)
' For Access
Dim dataAdapter As New OleDbDataAdapter(commandString, conn)
```

For the example code, we will be using a database named 'ProgrammersHeaven'. The database will have a table named 'Article'. The fields of the table 'Article' are

Field Name	Type	Description
artId	(Primary Key)Integer	The unique identifier for an article
title	String	The title of an article
topic	String	Topic or Series name of the article like 'Multithreading in Java' or 'VB.NET School'
authorId	(Foreign Key)Integer	Unique identity of author
lines	Integer	Number of lines in the article
dateOfPublishing	Date	The date the article was published

The 'ProgrammersHeaven' database also contains a table named 'Author' with the following fields:

Field Name	Type	Description
authorId	(Primary Key)Integer	The unique identity of the author
name	String	Name of the author

Accessing Data using ADO.NET
Data access using ADO.NET involves the following steps:

- Defining the connection string for the database server
- Defining the connection (SqlConnection or OleDbConnection) to the database using a connection string
- Defining the command (SqlCommand or OleDbCommand) or command string that contains the query
- Defining the Data Adapter (SqlDataAdapter or OleDbDataAdapter) using the command string and the connection object
- Creating a new DataSet object
- If the SQL command is SELECT, filling the DataSet object with the results of the query through the Data Adapter
- Reading the records from the DataTables in the DataSets using the DataRow and DataColumn objects
- If the SQL command is UPDATE, INSERT or DELETE. The dataset will be updated through the data adapter
- Accepting to save the changes in the DataSet to the database

Since we are demonstrating an application that uses both SQL Server and Access databases we need to include the following namespaces in our application:

```
Imports System.DataImports System.Data.OleDb     ' for Access database
Imports System.Data.SqlClient ' for SQL Server
```

Let's now discuss each of the above steps individually

Defining the connection string
The connection string defines which database server you are using, where it resides, your user name and password and optionally the database name.

For SQL Server we have written the following connection string:

```
' for Sql Server
Dim connectionString As String = "server=P-III; database=programmersheaven;" + _
"uid=phuser; pwd=nicecoding;"
```

First of all we have defined the instance name of the server, which is "P-III" on our system. Next we defined the name of the database, the user id (uid) and the password (pwd). These days when you install Sql server the installation forces you to think of a password for the SA

(System Administrator) user . Its good practice for you to create another admin user and not use the SA user ever again. This will help stop intruders breaking in to your database.

For Access, we have written the following connection string:

```
' for MS Access
Dim connectionString As String = "provider=Microsoft.Jet.OLEDB.4.0;" + _
"data source = c:\programmersheaven.mdb"
```

We have defined the provider of the access database. Then we have defined the data source which is the location of the target database.

Author's Note: Connection string details are vendor specific. A good source of connection strings for different databases is http://www.connectionstrings.com

Defining a Connection

A connection is defined using the connection string. This object is used by the Data Adapter to connect to and disconnect from the database. For SQL Server, a connection is created like this:

```
' for Sql Server
Dim conn As New SqlConnection(connectionString)
```

And for Access, a connection is created like this:

```
' for MS Access
Dim conn As New OleDbConnection(connectionString)
```

Here we have passed the connection string to the constructor of the connection object.

Defining the command or command string

The command contains the query to be passed to the database. We are using a command string. We will see the command object (SqlCommand or OleDbCommand) later in the lesson. The command string we have used in our application is:

```
Dim commandString As String = "SELECT " + _
                      "artId, title, topic, " + _
                       "article.authorId as authorId, " + _
                       "name, lines, dateOfPublishing " + _
                       "FROM " + _
                       "article, author " + _
                       "WHERE " + _
                       "author.authorId = article.authorId"
```

We have passed a query to select all the articles along with the author's name. Of course you may want to use a simpler query, such as:

```
Dim commandString As String = "SELECT * from article"
```

Defining the Data Adapter

We need to define the Data Adapter (SqlDataAdapter or OleDbDataAdapter). The Data Adapter stores your command (query) and connection. Using the connection and query the DaraAdapter connects to the database when asked, fetches the result of the query and stores it in a local dataset.

For SQL Server, a Data Adapter is created like this:

```
' for Sql Server
Dim dataAdapter As New SqlDataAdapter(commandString, conn)
```

And for Access, a data adapter is created like this:

```
' for MS Access
Dim dataAdapter As New OleDbDataAdapter(commandString, conn)
```

We have created a new instance of the Data Adapter and supplied it the command string and connection object in the constructor call.

Creating and filling the DataSet

Finally, we need to create an instance of the DataSet. As we mentioned earlier, a DataSet is a local and offline container of data. The DataSet object is created simply as:

```
Dim ds As New DataSet()
```

We need to fill the DataSet with the results from the query. We will use the DataAdapter object for this purpose and call its Fill() method. This is the step where the Data Adapter connects to the physical database and fetches the result of the query.

```
dataAdapter.Fill(ds, "prog")
```

We have called the Fill() method of dataAdapter object. We have supplied it the dataset to fill and the name of the table (DataTable) in which the result of query is filled.

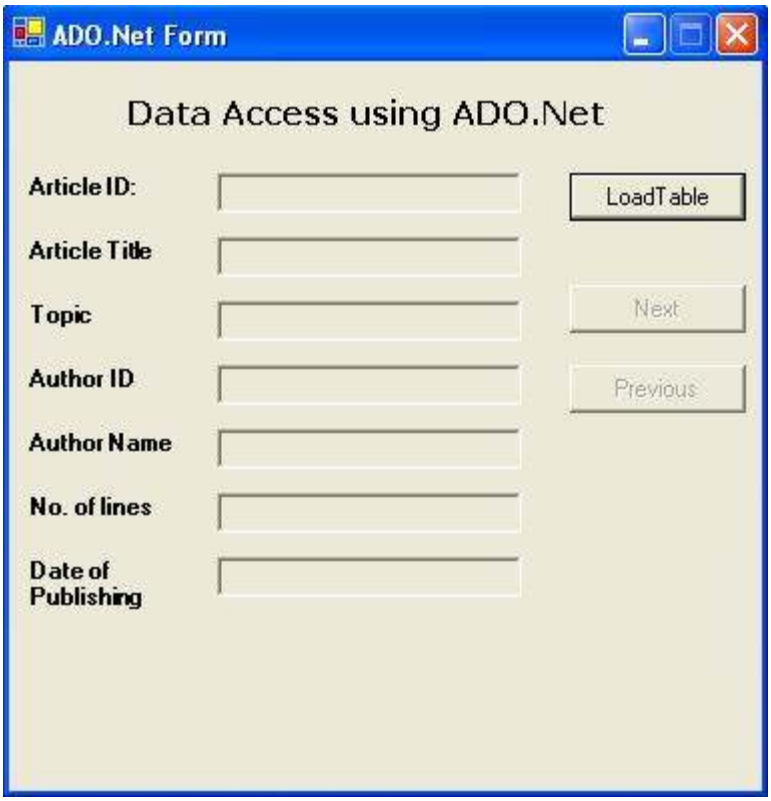
This is all we need to connect and fetch data from the database. Now the results of the query is stored in the dataset object in the prog table, which is an instance of the DataTable. We can get a reference to this table by using the indexer property of the DataSet object's Tables collection.

```
Dim dataTable As DataTable = ds.Tables("prog")
```

The indexer we have used takes the name of the table in the DataSet and returns the corresponding DataTable object. We can use the tables Rows and Columns collections to access the data in the table.

A Demonstration Application

Let's now create a demonstration application for accessing data. First create a windows form application and make the layout shown the following snapshot.



We have set the Name property of the text boxes (from top to bottom) as txtArticleID, txtArticleTitle, txtArticleTopic, txtAuthorId, txtAuthorName, txtNumOfLines and txtDateOfPublishing. Also we have set the ReadOnly property of all the text boxes to true as we don't want the user to change the text. The names of the buttons (from top to bottom) are btnLoadTable, btnNext and btnPrevious. Initially we have disabled the Next and Previous buttons (by setting their Enabled properties to false).

We have also defined three variables in the Form class:

```
Public Class ADOForm
        Inherits System.Windows.Forms.Form
        ' Private global members to be used in various methods
        Private dataTable As dataTable
        Private currRec As Integer = 0
        Private totalRec As Integer = 0
         ...
```

The dataTable object will be used to reference the table returned as a result of the query. The currRec and totalRec integer variables are used to keep track of the current record and total number of records in the table.

Loading table

For the LoadTable button, we have written the following event handler

```
Private Sub btnLoadTable_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnLoadTable.Click
' for Sql Server
Dim connectionString As String = "server=P-III; database=programmersheaven;" + _
                                 "uid=phuser; pwd=nicecoding;"
' for MS Access
'Dim connectionString As String = "provider=Microsoft.Jet.OLEDB.4.0;" +
```

```
        '                               "data source =
c:\\programmersheaven.mdb"
' for Sql Server
Dim conn As New SqlConnection(connectionString)
' for MS Access
'Dim conn As New OleDbConnection(connectionString)
Dim commandString As String = "SELECT " + _
                              "artId, title, topic, " + _
                              "article.authorId as authorId, " + _
                              "name, lines, dateOfPublishing " + _
                              "FROM " + _
                              "article, author " + _
                              "WHERE " + _
                              "author.authorId = article.authorId"
' for Sql Server
Dim dataAdapter As New SqlDataAdapter(commandString, conn)
' for MS Access
'Dim dataAdapter As New OleDbDataAdapter(commandString, conn)
Dim ds As New DataSet()
dataAdapter.Fill(ds, "prog")
dataTable = ds.Tables("prog")
currRec = 0
totalRec = dataTable.Rows.Count
FillControls()
btnNext.Enabled = True
btnPrevious.Enabled = True
End Sub
```

We created the connection, the Data Adapter and filled the DataSet object. All of which we have discussed earlier. It should be noted that we have commented out the code for the OleDb provider (MS-Access) and are using SQL Server specific code. If you would like to use an Access databases, you can simply comment the SQL server code out and uncomment the Access code.

Next, we have assigned the Data Table resulting from the query to the dataTable object which we declared at class level. We assigned zero to the currRec variable and assigned the number of rows in the dataTable to the totalRec variable:

```
dataTable = ds.Tables("prog")currRec = 0totalRec = dataTable.Rows.Count
```

Then we called the FillControls() method, which fills the controls (text boxes) on the form with the current record of the table "prog". Finally we enabled the Next and Previous Buttons.

Filling the controls on the Form

The FillControls() method in our program fills the controls on the form with the current record of the Data Table. The method is defined as follows:

```
Private Sub FillControls()
        txtArticleId.Text = dataTable.Rows(currRec)("artId").ToString()
        txtArticleTitle.Text = dataTable.Rows(currRec)("title").ToString()
        txtArticleTopic.Text = dataTable.Rows(currRec)("topic").ToString()
        txtAuthorId.Text = dataTable.Rows(currRec)("authorId").ToString()
        txtAuthorName.Text = dataTable.Rows(currRec)("name").ToString()
        txtNumOfLines.Text = dataTable.Rows(currRec)("lines").ToString()
        txtDateOfPublishing.Text =
dataTable.Rows(currRec)("dateOfPublishing").ToString()
End Sub
```

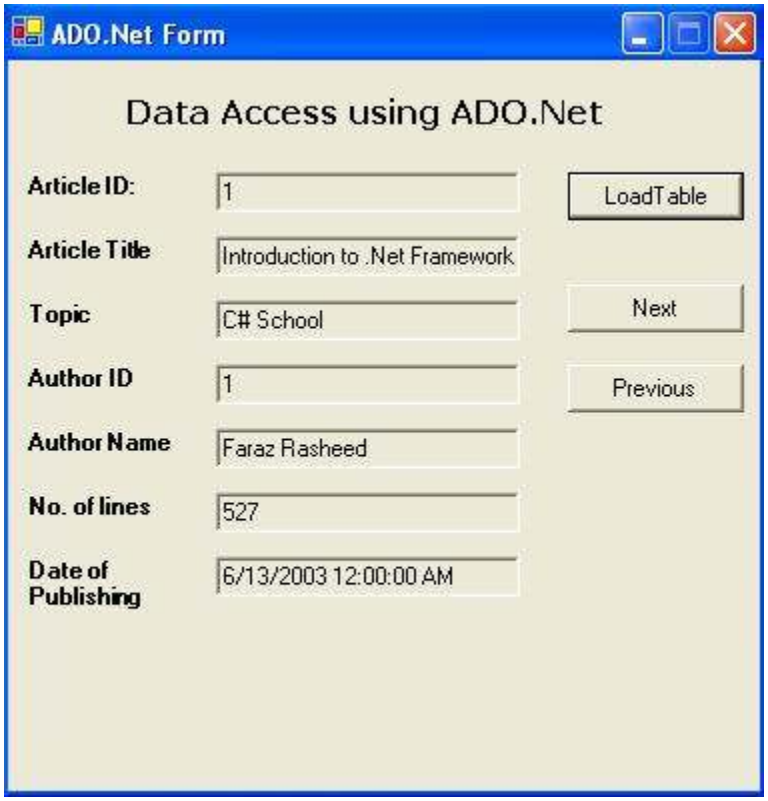
Here we have set the Text property of the text boxes to the string values of the corresponding fields of the current record. We have used the Rows collection of the dataTable and using its indexer. We have got the DataRow representing the current record. We have then accessed the indexer property of this DataRow using the column name to get the data in the respective field. If this explanation looks weird to you, you can simplify the above statements to:-

```
Dim row As DataRow = dataTable.Rows(currRec) ' getting current row
Dim data As Object = row("artId")            ' getting data in the artId field
Dim strData As String = data.ToString()      ' converting to string
txtArticleId.Text = strData                  ' display in the text box
```

which is equivalent to

```
txtArticleId.Text = dataTable.Rows(currRec)("artId").ToString()
```

Hence when you start the application and press the LoadTable button, you will see the following output:



Navigating through the records

Navigating through the records is again very easy. For the "Next" button, we have written the following simple event.

```
Private Sub btnNext_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles btnNext.Click
        currRec += 1
        If currRec >= totalRec Then
         currRec = 0
        End If
```

```
        FillControls()
End Sub
```

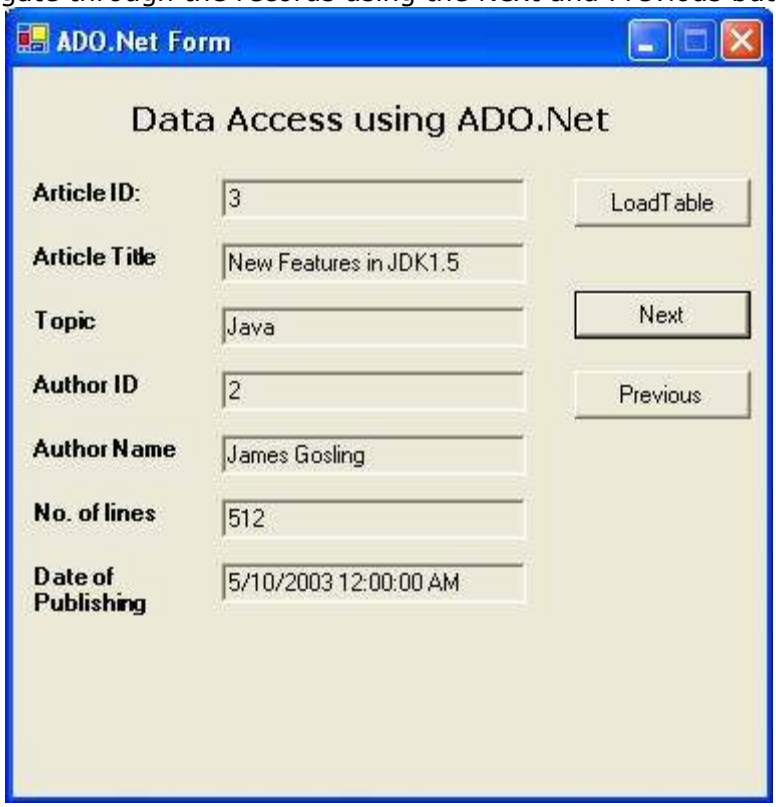
We first increment the integer variable currRec and check if it has crossed the last record (using the totalRec variable) in the table. If it has, then we move the current record to the first record. We then call the FillControls() method to display the current record on the form.

Similarly the event for the "Previous" button looks like this:

```
Private Sub btnPrevious_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles btnPrevious.Click
        currRec -= 1
        If currRec < totalRec Then
            currRec = totalRec - 1
        End If
        FillControls()
End Sub
```

Here we decrement the currRec variable and check if has crossed the first record and if it has then we move to the last record. Once again, we call the FillControls() method to display the current record.

You can now navigate through the records using the Next and Previous buttons.



Updating the table

So far we have only selected the data from the database and haven't changed any rows, inserted new rows or deleted existing rows. Let's learn how to perform these tasks one by one.

Please note that for this section, we will only use the "Article" table and will not be using the

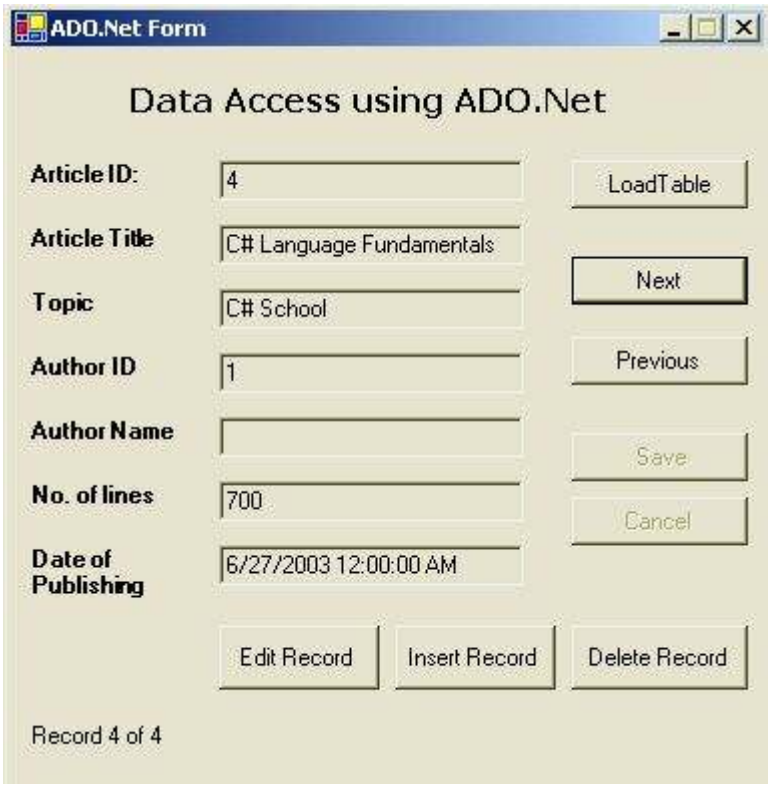
"Author" table for the sake of simplicity.

Steps for updating the table
Updating a table in ADO.NET is very interesting and easy. You need to follow these steps to update, insert and delete records:

- The Data Adapter class (SqlDataAdapter) has properties for each of the insert, update and delete commands. First of all we need to prepare the command (SqlCommand) and add it to the Data Adapter object. The commands are simple SQL commands with parameters. To implement this step, we will introduce a method InitializeCommands() in the following example.
- Secondly we need to add parameters to these commands. The parameters are simply the names of the data table fields involved in the particular commands. To implement this step, we will introduce a method named AddParams() in the following example.
- The two steps described above are only done once in the application. For each insert, update and delete; we insert, update and delete the corresponding Data Row (DataRow) of the Data Table (DataTable) object.
- After an update we call the Update() method of the Data Adapter class by supplying to it, the DataSet and table name as parameters. This updates our local DataSet.
- Finally we call the AcceptChanges() method of the DataSet object to store the changes in the DataSet to the physical database.

Again for simplicity we haven't included the steps for data validation, which is a key part of a data update in a "live" application.

Building the Application
The application will finally look like this:



In this application, we have defined several database access objects (like SqlDataAdapter, DataSet) as private class members so that we can access them in different methods.

```
Public Class ADOForm
        Inherits System.Windows.Forms.Form
        ' Private global members to be used in various methods
        Private conn As SqlConnection
        Private conn As SqlConnection
        Private ds As DataSet
        Private dataTable As dataTable
        Private currRec As Integer = 0
        Private totalRec As Integer = 0
        Private insertSelected As Boolean
        ...
```

Loading the table and displaying data in the form's controls
The event for the 'Load Table' button has changed somewhat and now looks like this:

```
Private Sub btnLoadTable_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles btnLoadTable.Click
        Me.Cursor = Cursors.WaitCursor
        Dim connectionString As String = "server=P-III; database=programmersheaven;" +
_
                                         "uid=phuser; pwd=nicecoding;"        conn =
New SqlConnection(connectionString)
        Dim commandString As String = "SELECT * from article"
```

```
        dataAdapter = New SqlDataAdapter(commandString, conn)
         ds = New DataSet()        dataAdapter.Fill(ds, "article")
         dataTable = ds.Tables("article")
         currRec = 0
         totalRec = dataTable.Rows.Count
        FillControls()
        ' show current record on the form
         InitializeCommands() ' prepare commands
        ToggleControls(True) ' enable corresponding controls
         Me.Cursor = Cursors.Default
End Sub
```

We have changed the cursor to WaitCursor at the start of the method, and changed it to Default at the end of the method. Later we have called the InitializeCommands() method after filling the controls with the first record.

Initialing Commands

The InitializeCommands() is the key method to understand in this application. It is defined in the program as:

```
    Private Sub InitializeCommands()
        ' Preparing Insert SQL Command
        dataAdapter.InsertCommand = conn.CreateCommand()
        dataAdapter.InsertCommand.CommandText = _
            "INSERT INTO article " + _
            "(artId, title, topic, authorId, lines, dateOfPublishing) " + _
            "VALUES(@artId, @title, @topic, @authorId, @lines, @dateOfPublishing)"
        AddParams(dataAdapter.InsertCommand, "artId", "title", "topic", _
        "authorId", "lines", "dateOfPublishing")
        ' Preparing Update SQL Command
        dataAdapter.UpdateCommand = conn.CreateCommand()
        dataAdapter.UpdateCommand.CommandText = _
            "UPDATE article SET " + _
            "title = @title, topic = @topic, authorId = @authorId, " + _
            "lines = @lines, dateOfPublishing = @dateOfPublishing " + _
            "WHERE artId = @artId"
        AddParams(dataAdapter.UpdateCommand, "artId", "title", "topic", _
            "authorId", "lines", "dateOfPublishing")
        ' Preparing Delete SQL Command
        dataAdapter.DeleteCommand = conn.CreateCommand()
        dataAdapter.DeleteCommand.CommandText = "DELETE FROM article WHERE artId =
@artId"
        AddParams(dataAdapter.DeleteCommand, "artId")
    End Sub
```

The SqlDataAdapter (and OleDbDataAdapter) class has properties for each of the Insert, Update and Delete commands. The type of these properties is SqlCommand (and OleDbCommand respectively). We have created the Commands using the connection (SqlConnection) object's CreateCommand() method. We then set the CommandText property of these commands to the respective SQL queries in string format. The thing to note here is that the above commands are very general and we have used the name of the fields with an '@' sign wherever the specific field value is required. For example, we have written the DeleteCommand's CommandText as:

```
"DELETE FROM article WHERE artId = @artId"
```

Here we have used @artId instead of a physical value. In fact, this value will be replaced by the specific value when we delete a particular record.

Adding Parameters to the commands

After defining the CommandText for each command, we call the AddParams() method by

passing it to the command itself and the names of any fields used in the corresponding query. The AddParams() method is very simple and adds the fields with an '@' symbol to the Parameters collection of the command. The AddParams() method is defined as

```
    Private Sub AddParams(ByVal cmd As SqlCommand, ByVal ParamArray cols() As String)
        ' Adding Hectice parameters in SQL Commands
        Dim col As String
        For Each col In cols
            cmd.Parameters.Add("@" + col, SqlDbType.Char, 0, col)
        Next
    End Sub
```

The very first thing to note in the AddParams method is that the type of the second parameter is ' ParamArray cols() As String '

```
Private Sub AddParams(ByVal cmd As SqlCommand, ByVal ParamArray cols() As String)
```

The ParamArray keyword is used to tell the compiler that the method will take a variable number of string elements to be stored in the string array named 'cols'. If the method is called like this:

```
AddParams(someCommand, "one")
```

The size of the cols array would be one, and if the method is called like this:

```
AddParams(someCommand, "one", "two", "numbers")
```

The size of the cols array would be three. Isn't it useful and extremely simple at the same time? VB.NET rules!

So let's get back to the method. What it does is simply adds the supplied column name prefixed with an '@' in the Parameters collection of the SqlCommand class. The other parameters of the Add() method are the type of the parameter, the size of the parameter and the corresponding column name.

After following the above steps, we have defined different commands for the record update. Now we can update records of the table.

The ToggleControls() method of our application

In the Load Table buttons event handler, we called the ToggleControls() method after initializing the commands

```
    Private Sub btnLoadTable_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles btnLoadTable.Click
        Me.Cursor = Cursors.WaitCursor
        ...
         FillControls()
       ' show current record on the form
        InitializeCommands() ' prepare commands
        ToggleControls(True) ' enable corresponding controls
        Me.Cursor = Cursors.Default
    End Sub
```

We have defined the ToggleControls() method in our application to change the Enabled and ReadOnly properties of buttons and text boxes in our form at the respective times. If the ToggleControls() method is called with a false boolean value it will take the form to the edit

mode and if called with a true value it will take the form back to normal mode. The method is defined as:

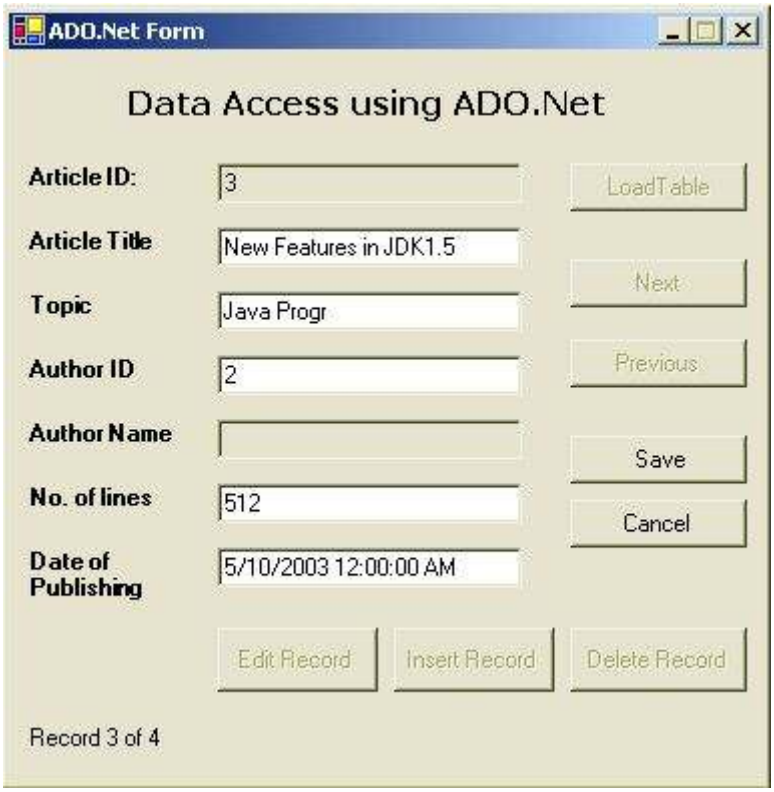
```
Private Sub ToggleControls(ByVal val As Boolean)
    txtArticleTitle.ReadOnly = val
    txtArticleTopic.ReadOnly = val
    txtAuthorId.ReadOnly = val
    txtNumOfLines.ReadOnly = val
    txtDateOfPublishing.ReadOnly = val
    btnLoadTable.Enabled = val
    btnNext.Enabled = val
    btnPrevious.Enabled = val
    btnEditRecord.Enabled = val
    btnInsertRecord.Enabled = val
    btnDeleteRecord.Enabled = val
    btnSave.Enabled = Not val
    btnCancel.Enabled = Not val
End Sub
```

Editing (or Updating) Records

For editing the current record, we have provided an Edit Record button on the form. The event for this button is surprisingly very simple and is:

```
Private Sub btnEditRecord_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles btnEditRecord.Click
    ToggleControls(False)
End Sub
```

This event simply takes the form and the controls to the edit mode by passing a false value to the ToggleControls() method presented above. When a user presses the Edit Record button, the form is changed, so it looks like:



As you can see now, the text boxes are editable and the Save and Cancel buttons are enabled. If the user wishes not to save the changes, they can select the Cancel button, and if they wishes to save the changes, they can select the Save button after making any changes.

Event for the Save Button

The event for the Save button reads the values from the text boxes and stores them in the current record in the data table. The event for the Save Button is:

```
    Private Sub btnSave_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles btnSave.Click
        lblLabel.Text = "Saving Changes..."
        Me.Cursor = Cursors.WaitCursor
        Dim row As DataRow = dataTable.Rows(currRec)
        row.BeginEdit()
        row("title") = txtArticleTitle.Text
        row("topic") = txtArticleTopic.Text
        row("authorId") = txtAuthorId.Text
        row("lines") = txtNumOfLines.Text
        row("dateOfPublishing") = txtDateOfPublishing.Text
        row.EndEdit()
        dataAdapter.Update(ds, "article")
        ds.AcceptChanges()
        ToggleControls(True)
        insertSelected = False
        Me.Cursor = Cursors.Default
        lblLabel.Text = "Changes Saved"
    End Sub
```

Here we first change the progress label (lblLabel) text to show the current status and then change the cursor to a wait cursor.

```
lblLabel.Text = "Saving Changes...
"Me.Cursor = Cursors.WaitCursor
```

Then we take a reference of the current record's row and call its BeginEdit() method. An update on a row is usually bound by a DataRow's BeginEdit() and EndEdit() methods. The BeginEdit() method temporarily suspends the events for the validation of row's data. Within the BeginEdit() and EndEdit() boundary, we stored the changed values in the text boxes to the row.

```
Dim row As DataRow = dataTable.Rows(currRec)
row.BeginEdit()
row("title") = txtArticleTitle.Text
row("topic") = txtArticleTopic.Text
row("authorId") = txtAuthorId.Text
row("lines") = txtNumOfLines.Text
row("dateOfPublishing") = txtDateOfPublishing.Text
row.EndEdit()
```

After saving the changes in the row, we update the DataSet and table by calling the Update method of the Data Adapter. This saves the changes in the local repository of data:

DataSet. To save the changed rows and tables to the physical database, we called the AcceptChanges() method of the DataSet class.

```
dataAdapter.Update(ds, "article")
ds.AcceptChanges()
```

Finally we brought the controls to the normal mode by calling the ToggleControls() method and passing it the true value. We set the insertSelected variable to false, changed the cursor back to normal and updated the progress label. (The use of the insertSelected variable is discussed later in the Cancel button event)

```
ToggleControls(True)
insertSelected = False
Me.Cursor = Cursors.Defaultlbl
Label.Text = "Changes Saved"
```

It is important to note here that the Save button is used to save the changes in the current record. It may be selected after either the Edit Record or Insert Record buttons. In the case of the Edit Record button, the pointer is already on the current record, while in the case of the Insert Record button, as we will see shortly in the Inserting Record section, the program inserts a new empty row. Then moves the current record pointer to it and presents it to the user to insert the values. Hence, the job of the Save button in both cases is to save the changes in the current record from the text boxes to the data table, updating the Data Set and finally updating the database.

Event for the Cancel Button

The Cancel button's event handler is:

```
    Private Sub btnCancel_Click(ByVal sender As System.Object,
                ByVal e As System.EventArgs) Handles btnCancel.Click
        If insertSelected Then
            btnDeleteRecord_Click(Nothing, Nothing)
            insertSelected = False
        End If
        FillControls()
        ToggleControls(True)
    End Sub
```

The form and controls can be brought into edit mode by pressing either the Edit Record or the Insert Record button. When the Insert Record button is selected, the program inserts an empty record (row) to the table (the details of which we will see shortly) and brings the form and controls to edit mode by using the ToggleControls(false) statement. The Cancel button is used to cancel both editing of the current record and the newly inserted record. When canceling the insertion of a new record, the program needs to delete the current (newly inserted) row from the table. We have used a Boolean variable 'insertSelected' in our application, which is set to true when the user selects the Insert Record button. This Boolean value informs the Cancel button whether the edit mode was set by the Edit Record button or by the Insert Record button. Hence the Cancel button's event first checks whether insertSelected is true and if it is, it calls the Delete button's event to delete the current record and sets insertSelected back to false.

```
If insertSelected Then
    btnDeleteRecord_Click(Nothing, Nothing)
```

```
    insertSelected = False
End If
```

Now fill the controls (text boxes) from the data in the current record and bring the controls back to the normal mode.

```
FillControls()
ToggleControls(True)
```

Inserting Records

To insert a record into the table, the user can select the Insert Record button. A record is inserted into the table by adding a new row to the DataTable's Rows collection. Here is the event for the Insert Record button.

```
    Private Sub btnInsertRecord_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles btnInsertRecord.Click
        insertSelected = True
        Dim row As DataRow = dataTable.NewRow()
        dataTable.Rows.Add(row)
        totalRec = dataTable.Rows.Count
        currRec = totalRec - 1
        row("artId") = totalRec
        txtArticleId.Text = totalRec.ToString()
        txtArticleTitle.Text = ""
        txtArticleTopic.Text = ""
        txtAuthorId.Text = ""
        txtNumOfLines.Text = ""
        txtDateOfPublishing.Text = DateTime.Now.Date.ToString()
        ToggleControls(False)
    End Sub
```

First of all we set the insertSelected variable to true, so that later the Cancel button may get informed that the edit mode was set by the Insert Record button. We then created a new DataRow using the DataTable's NewRow() method. Then we added it to the Rows collection of the data table and updated the currRec and totalRec variables.

```
Dim row As DataRow = dataTable.NewRow()
dataTable.Rows.Add(row)
totalRec = dataTable.Rows.Countcurr
Rec = totalRec - 1
```

The new row is ready to have the new values inserted into it. Here we have set the artId field to the total number of records as we don't want to allow the user to set the primary key field of the table. Of course this is just a design issue. You may want to allow your user to insert the primary key field value too.

```
row("artId") = totalRec
txtArticleId.Text = totalRec.ToString()
```

We then cleared all the text boxes, but filled the Date of the Publishing text box with the current date in order to help the user, and finally set the edit mode by calling the ToggleControls() method.

```
txtArticleTitle.Text = ""
txtArticleTopic.Text = ""
txtAuthorId.Text = ""
txtNumOfLines.Text = ""
txtDateOfPublishing.Text = DateTime.Now.Date.ToString()
ToggleControls(False)
```

Deleting a Record

Deleting a record is again very simple. All you need to do is get a reference to the target row and call its Delete() method. Then you need to call the Update() method of the data adapter and the AcceptChanges() method of the DataSet to permanently save your changes in the data table to the physical database. The event handler for the Delete Record button is:

```
    Private Sub btnDeleteRecord_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles btnDeleteRecord.Click
        Dim res As DialogResult = MessageBox.Show( _
                    "Are you sure you want to delete the current record?", _
                    "Confirm Record Deletion", MessageBoxButtons.YesNo)
        If res = DialogResult.Yes Then
            Dim row As DataRow = dataTable.Rows(currRec)
            row.Delete()
            dataAdapter.Update(ds, "article")
            ds.AcceptChanges()
            lblLabel.Text = "Record Deleted"
            totalRec -= 1
            currRec = totalRec - 1
            FillControls()
        End If
    End Sub
```

Since selecting the delete record button will permanently delete the current record, we need to seek confirmation from the user to check they are sure they wish to go ahead with the deletion. For this purpose we present the user with a message box with Yes and No buttons.

```
Dim res As DialogResult = MessageBox.Show( _
           "Are you sure you want to delete the current record?", _
           "Confirm Record Deletion", MessageBoxButtons.YesNo)
```

If the user selects the Yes button in the message box, the code to delete the current record is executed.

```
If res = DialogResult.Yes Then
Dim row As DataRow = dataTable.Rows(currRec)
    row.Delete()
    dataAdapter.Update(ds, "article")
    ds.AcceptChanges()
    lblLabel.Text = "Record Deleted"
    totalRec -= 1
    currRec = totalRec - 1
    FillControls()
End If
```

First we got a reference to the row representing the current record in the data table and called its Delete() method. We then saved the changes to the database and updated the totalRec and currRec variables. Finally, we filled the controls (text boxes) with the last record in the table. This concludes our demonstration application to perform common data access tasks using ADO.NET. The complete source code of this program can be downloaded by clicking here

---

Using Stored Procedures

Till now we have written the SQL statements inside our code which usually is not a good

idea. In this way you are exposing your database schema (design) in the code which may be changed. Hence most of the time we use stored procedures instead of plain SQL statements. A stored procedure is a precompiled executable object that contains one or more SQL statements. Hence you can replace your complex SQL statements with a single stored procedure. A stored procedure may be written to accept inputs and return output.

Sample Stored Procedures

We now present some sample stored procedures with short descriptions. This text assumes that you are familiar with stored procedures and other basic database concepts related to database systems as the intention here is not to discuss databases but the use of databases in the ADO.NET environment. But still we present a small review of four basic types of stored procedure for the SELECT, INSERT, UPDATE and DELETE operations. In SQL Server, you can create and add stored procedures to your database using the SQL Server Enterprise Manager.

UPDATE Stored Procedure

A simple stored procedure to update a record is

```
CREATE PROCEDURE UpdateProc (
        @artId as int,
        @title as varchar(100),
        @topic as varchar(100),
        @authorId as int,
        @lines as int,
        @dateOfPublishing as datetime)
AS
        UPDATE Article SET
                title = @title, topic = @topic, authorId = @authorId,
                lines = @lines, dateOfPublishing = @dateOfPublishing
        WHERE artId = @artId
GO
```

The name of the stored procedure is UpdateProc and it has input parameters for each of the fields of our Article table. The query to be executed when the stored procedure is run updates the record with the supplied primary key (@artId) using the supplied parameters. It is very similar to the code we have written to initialize commands in the previous example and we suppose you don't have any problem in understanding this even if you are not familiar with stored procedures.

INSERT Stored Procedure

A simple stored procedure to insert a record is

```
CREATE PROCEDURE InsertProc (
        @artId as int,
        @title as varchar(100),
        @topic as varchar(100),
        @authorId as int,
        @lines as int,
        @dateOfPublishing as datetime)
AS
        INSERT INTO article (artId, title, topic, authorId, lines, dateOfPublishing)
```

```
        VALUES(@artId, @title, @topic, @authorId, @lines,
@dateOfPublishing)
GO
```

The stored procedure above is named InsertProc and is very similar to the UpdateProc except that here we are using the INSERT SQL statement instead of the UPDATE command.

DELETE Stored Procedure

A simple stored procedure to delete a record is

```
CREATE PROCEDURE DeleteProc (@artId as int)
AS
        DELETE FROM article WHERE artId = @artId
GO
```

Here we have used only one parameter, as to delete a record you only need its primary key value.

SELECT Stored Procedure

A simple stored procedure to select records from a table is

```
CREATE PROCEDURE SelectProc
AS
        SELECT * FROM Article
GO
```

This probably is the simplest of all. It does not take any parameters and only selects all the records from the Article table.

All the four stored procedures presented above are kept extremely simple so that the reader does not find any difficulty in understanding the use of stored procedures in VB.NET code. Real world stored procedures are much more complex and offcourse, more useful than these.

Using Stored Procedures with ADO.NET in VB.NET

Using stored procedures with ADO.NET in VB.NET is extremely simple, especially when we have developed the application with SQL commands. All we need now is to replace the SQL commands in the previous with calls to stored procedures. For example in the InitalizeCommands() method of the previous example we created the insert command as

```
    Private Sub InitializeCommands()
        ' Preparing Insert SQL Command
        dataAdapter.InsertCommand = conn.CreateCommand()
        dataAdapter.InsertCommand.CommandText = _
            "INSERT INTO article " + _
            "(artId, title, topic, authorId, lines, dateOfPublishing) " + _
            "VALUES(@artId, @title, @topic, @authorId, @lines, @dateOfPublishing)"
        AddParams(dataAdapter.InsertCommand, "artId", "title", "topic", _
        "authorId", "lines", "dateOfPublishing")
        ...
```

When using the stored procedure the command will be prepared as

```
    Private Sub InitializeCommands()
        ' Preparing Insert SQL Command
        Dim insertCommand = New SqlCommand("InsertProc", conn)
        insertCommand.CommandType = CommandType.StoredProcedure
        dataAdapter.InsertCommand = insertCommand
        AddParams(dataAdapter.InsertCommand, "artId", "title", "topic", _
                  "authorId", "lines", "dateOfPublishing")
        insertCommand.UpdatedRowSource = UpdateRowSource.None
        ...
```

We first created a SqlCommand object named insertCommand which supplies the target stored procedure constructor call (InsertProc in this case) and the connection to be used.

```
Dim insertCommand = New SqlCommand("InsertProc", conn)
```

We then set the CommandType property of the insertCommand object to the StoredProcedure enumeration value. This tells the runtime that the command used here is a stored procedure.

```
insertCommand.CommandType = CommandType.StoredProcedure
```

Finally we added this command to the Data Adapter object using its InsertCommand property. Just like in the previous example we have also added parameters to the command.

```
dataAdapter.InsertCommand = insertCommand
AddParams(dataAdapter.InsertCommand, "artId", "title", "topic", _
          "authorId", "lines", "dateOfPublishing")
```

The modified InitializeCommands() method

Hence when using stored procedures, the modified InitializeCommands() method is:

```
    Private Sub InitializeCommands()
        ' Preparing Select Command
        Dim selectCommand = New SqlCommand("SelectProc", conn)
        selectCommand.CommandType = CommandType.StoredProcedure
        dataAdapter.SelectCommand = selectCommand
        ' Preparing Insert SQL Command
        Dim insertCommand = New SqlCommand("InsertProc", conn)
        insertCommand.CommandType = CommandType.StoredProcedure
        dataAdapter.InsertCommand = insertCommand
        AddParams(dataAdapter.InsertCommand, "artId", "title", "topic", _
                  "authorId", "lines", "dateOfPublishing")
        insertCommand.UpdatedRowSource = UpdateRowSource.None
        ' Preparing Update SQL Command
        Dim updateCommand As New SqlCommand("UpdateProc", conn)
        updateCommand.CommandType = CommandType.StoredProcedure
        dataAdapter.UpdateCommand = updateCommand
        AddParams(dataAdapter.UpdateCommand, "artId", "title", _
                  "topic", "authorId", "lines", "dateOfPublishing")
        ' Preparing Delete SQL Command
        Dim deleteCommand As New SqlCommand("DeleteProc", conn)
        deleteCommand.CommandType = CommandType.StoredProcedure
        dataAdapter.DeleteCommand = deleteCommand
        AddParams(dataAdapter.DeleteCommand, "artId")
    End Sub
```

Note that we have used the name of the stored procedure in the constructor calls to the SqlCommand's object. Also note that now we have also included the select command in the

Data Adapter. The only other change you need to do now is in the Load Table button's event.

The modified Load Table button's event

Since we are now using stored procedures to select the records from the table, we need to change the Load table button event handler. The modified version of this event is:

```
    Private Sub btnLoadTable_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles btnLoadTable.Click
        Me.Cursor = Cursors.WaitCursor
        Dim connectionString As String = "server=P-III; database=programmersheaven;" +
_
                                         "uid=phuser; pwd=nicecoding;"
        conn = New SqlConnection(connectionString)
        Dim commandString As String = "SELECT * from article"
        dataAdapter = New SqlDataAdapter()
        InitializeCommands() ' prepare commands
        ds = New DataSet()
        dataAdapter.Fill(ds, "article")
        dataTable = ds.Tables("article")
        currRec = 0
        totalRec = dataTable.Rows.Count
        FillControls()
       ' show current record on the form
        ToggleControls(True) ' enable corresponding controls
        Me.Cursor = Cursors.Default
    End Sub
```

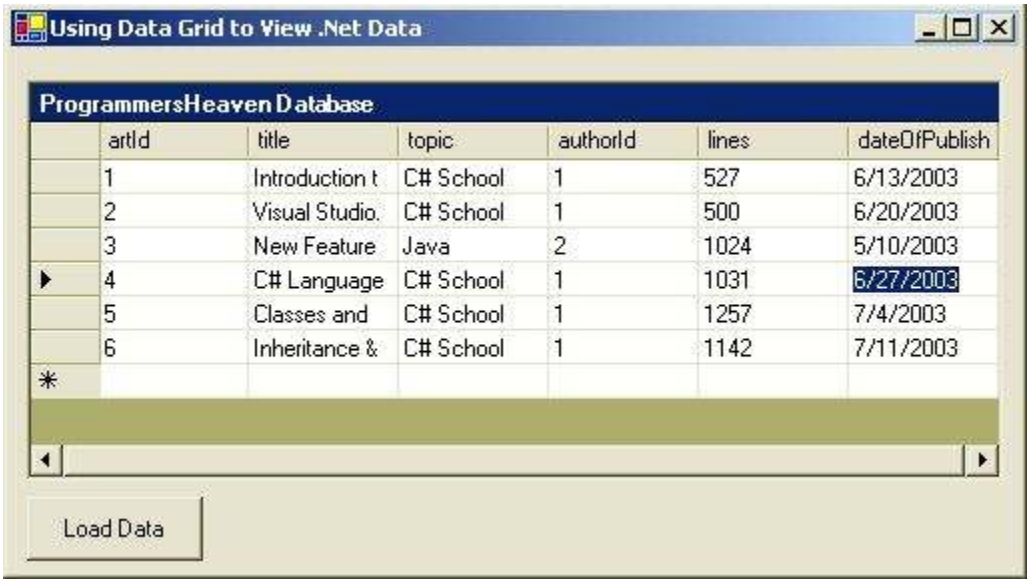
The two changes made are highlighted with bold formatting in the above code. Now we are creating the SqlDataAdapter object using the default no-argument constructor. This time we are not passing it the select command as we are using the stored procedure for it and we are not passing it the connection object as each command is holding the connection reference and all the commands have finally been added to the Data Adapter. The second change is the reordering of the InitializeCommands() method call. It is now called just before filling the DataSet as the select command is also being prepared inside the InitializeCommands() method.

This is all you need to change in the previous example application to replace the SQL commands with stored procedures. Isn't it simple?! The complete source code of this application can be downloaded by clicking here.

Author's Note: When using the DataSet and disconnected architecture, we don't update the Data Source (by calling the DataAdapter's Update() method and the DataSet's AcceptChanges() method) for each update. Instead we make the changes local and update all these changes later as a batch. This provides optimized use of network bandwidth. But, this offcourse is not a better option when multiple users are updating the same database. When changes are not to be done locally and need to be reflected at database server at the same time, it is preferred to use the connected oriented environment for all the changes (UPDATE, INSERT and DELETE) by calling the ExecuteNonQuery() method of your command (SqlCommand or OleDbCommand) object. A demonstration for the use of connected environment is presented in the supplementary topic of this lesson

Using the Data Grid Control to View .NET data

The Data Grid is the standard control for viewing data in the .NET environment. A data grid control is represented in .Net by the System.Windows.Forms.DataGrid class. The data grid control can display data from a number of data sources, e.g. a data table, DataSet, data view or an array. A data grid control looks like this:



A Demonstration Application for the Data Grid Control

Let's create a simple application that loads table data from a database server to the data grid control. First of all add a data grid control and a button to your form using the Visual Studio toolbox. We have set the Name property of the data grid to 'dgDetails' and its CaptionText property to 'ProgrammersHeaven Database'. The name of the button is 'btnLoadData'. The event for the button is:

```
    Private Sub btnLoadData_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles btnLoadData.Click
        Dim connectionString As String = "server=P-III; database=programmersheaven;" +
_
                                         "uid=phuser; pwd=nicecoding;"
        Dim conn As New SqlConnection(connectionString)
        Dim cmdString As String = "SELECT * FROM article"
        Dim dataAdapter As New SqlDataAdapter(cmdString, conn)
        Dim ds As New DataSet()         dataAdapter.Fill(ds, "article")
        dgDetails.SetDataBinding(ds, "article")
    End Sub
```

Here we first created a Data Adapter and filled the Data Set using it, as we did in the previous applications. The only new thing is the binding of the "article" table on the data grid control. This is done by calling the SetDataBinding() method of the DataGrid class. The first parameter of this method is the DataSet, while the second parameter is the name of the table in the DataSet.

```
dgDetails.SetDataBinding(ds, "article")
```

When you execute this program and select the Load button you will see the output presented in the previous figure.

Second Demonstration - Using multiple related tables

We will now see how we can use the Data Grid control to show multiple related tables. When two tables are related, one is called the parent table and the other is called the child table. The child table contains the primary key of the parent table as a foreign key. For example, in our ProgrammersHeaven database, table Author is the parent table of the Article table as the Article table contains 'AuthorId' as a foreign key, which is the primary key in the Author table.

In this example, we will use the Data Grid to show the related records from the Article and Author tables. In order to specify the relationship between the two tables we need to use the DataRelation class:

```
Dim relation As New DataRelation("ArtAuth", _
    ds.Tables("author").Columns("authorId"), _
    ds.Tables("article").Columns("authorId") _
    )
```

Here the first argument of the DataRelation constructor is the name for the new relation, while the second and third arguments are the columns of the tables which will be used to relate the two tables. After creating this relationship we need to add it to the Relations collection of the dataset.

```
ds.Relations.Add(relation)
```

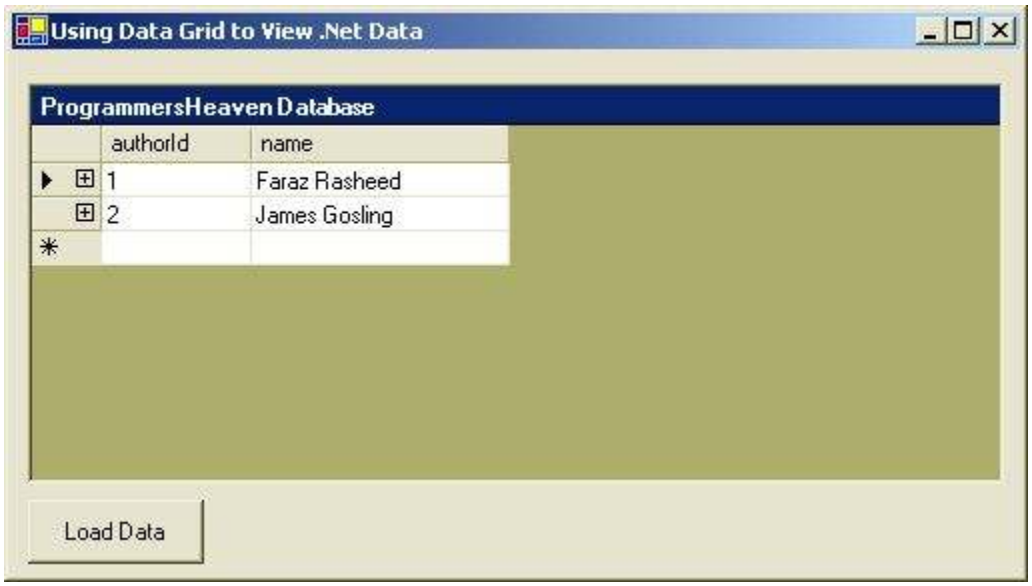
Hence the modified code for the Load Data button is:

```
    Private Sub btnLoadData_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles btnLoadData.Click
        Dim connectionString As String = "server=P-III; database=programmersheaven;" +
_
                                         "uid=phuser; pwd=nicecoding;"
        Dim conn As New SqlConnection(connectionString)
        Dim cmdString As String = "SELECT * FROM article"
        Dim dataAdapter As New SqlDataAdapter(cmdString, conn)
        Dim ds As New DataSet()
        dataAdapter.Fill(ds, "article")
        cmdString = "SELECT * FROM author"
        dataAdapter = New SqlDataAdapter(cmdString, conn)
        dataAdapter.Fill(ds, "author")
        Dim relation As New DataRelation("ArtAuth", _
                    ds.Tables("author").Columns("authorId"), _
                    ds.Tables("article").Columns("authorId") _
                    )
        ds.Relations.Add(relation)
        Dim dv As New DataView(ds.Tables("author"))
        dgDetails.DataSource = dv
    End Sub
```

In the above code we first filled the DataSet with the two tables, then defined the

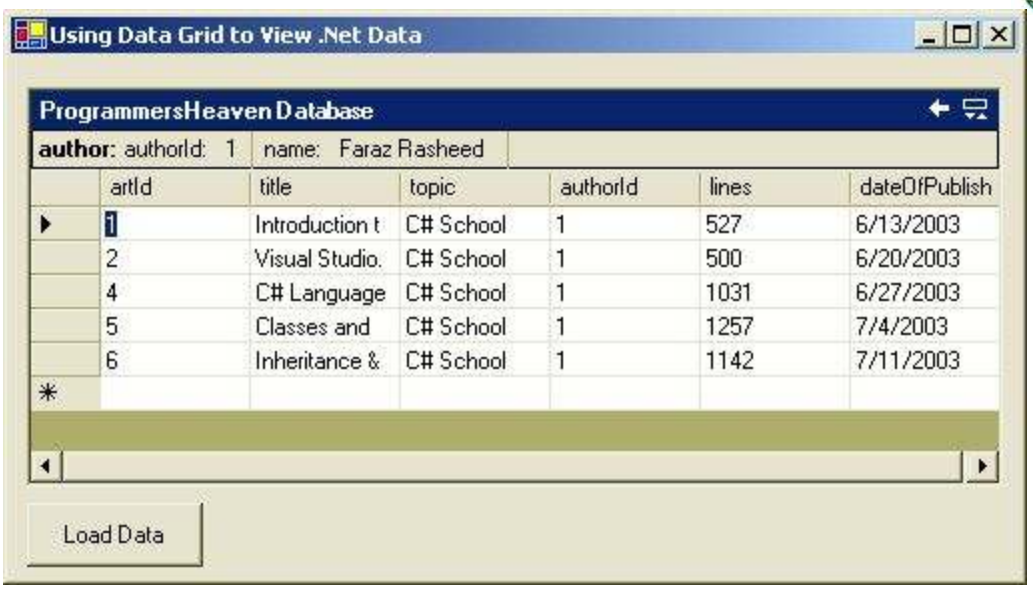
relationship between them and then added it to the dataset. In the last two lines of code, we created an instance of the DataView class by supplying the parent table in its constructor call and then set the DataSource property of the Data Grid to this Data View.

When we compile and execute our application, the Data Grid will show the records of the parent table with a '+' button on the left of each record:



When you press the '+' button on the left of the record, it will expand to show the name of the relationship as a link:

Now when you click the relation name, the Data Grid will show all the related records in the child table:



Still, you can see the parent record at the top of all the rows of the child table. You can go back to the parent table using the back arrow button (<==) at the title bar of the Data Grid.

You can download the source code of this sample from here

Supplementary Topic: Data Access in a Connected Environment

In a connected environment, opening and closing the database connection is your responsibility.

Retrieving data using the SELECT command

For retrieving data using the SELECT command, we can write the following code:

```
    Private Sub btnLoadData_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles btnLoadData.Click
        Dim connString As String = "server=siraj; database=programmersheaven;" + _
                                   "uid=sa; pwd="
        Dim conn As New SqlConnection(connString)
        Dim cmdString As String = "select * from author"
        Dim cmd As New SqlCommand(cmdString, conn)
        conn.Open()
        Dim reader As SqlDataReader = cmd.ExecuteReader()
        While reader.Read()
            txtData.Text += reader("authorId").ToString()
            txtData.Text += ", "
            txtData.Text += reader("name").ToString()
            txtData.Text += vbCrLf
        End While
        conn.Close()
    End Sub
```

Here we have created a SqlConnection and an SqlCommand object by using the SQL SELECT statement. We then open the connection using the Open() method of the SqlConnection class:

```
conn.Open()
```

After opening the connection we have executed the command using the ExecuteReader() method of the SqlCommand class. This method returns an object of type DataReader which can be used to read the data returned as a result of the select query.

```
Dim reader As SqlDataReader = cmd.ExecuteReader()
```

Then we read all the records in the table using the reader object. The Read() method of the DataReader class advances the current record pointer to the next record and returns a true value if it is successful. The DataReader class has overloaded an indexer property on its object which takes the name (or number) of the column and returns the value in the specified column of the current record.

```
While reader.Read()
    txtData.Text += reader("authorId").ToString()
    txtData.Text += ", "
    txtData.Text += reader("name").ToString()
    txtData.Text += vbCrLf
End While
```

Finally we have closed the database connection by calling the Close() method of the SqlConnection class.

```
conn.Close()
```

We have written the above code in the event handler of the Load Button of our form and the output is displayed in a textbox named 'txtData'

Updating the DB using INSERT, UPDATE and DELETE commands

The procedure for the data base records using INSERT, UPDATE and DELETE commands is very similar to the one we presented in the previous example. The exception being that here the command does not return anything and thus the method to call on the SqlCommand object is called ExecuteNonQuery(). The demonstration code for this scenario is:

```
    Private Sub btnExecuteNonQuery_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles btnExecuteNonQuery.Click
        Dim connString As String = "server=siraj; database=programmersheaven;" + _
                                   "uid=sa; pwd="
        Dim conn As New SqlConnection(connString)
        ' Un-comment any one of the following three Queries
        ' INSERT Query
        Dim cmdString As String = "INSERT INTO Author " + _
                                  "(authorId, name) " + _
                                  "VALUES(3, 'Anders Hejlsberg')"
        ' UPDATE Query
        'Dim cmdString As String =  "UPDATE Author " + _
        '                           "SET name = 'Grady Booch' " + _
        '                           "WHERE authorId = 3"
        ' DELETE Query
        'Dim cmdString As String =  "DELETE FROM Author " +
         '                          "WHERE authorId = 3"
       Dim cmd As New SqlCommand(cmdString, conn)
        conn.Open()
        cmd.ExecuteNonQuery()
        conn.Close()
    End Sub
```

Note that in the code above we have used the INSERT, UPDATE and DELETE commands as a command string and we have called the ExecuteNonQuery() method of the SqlCommand object.

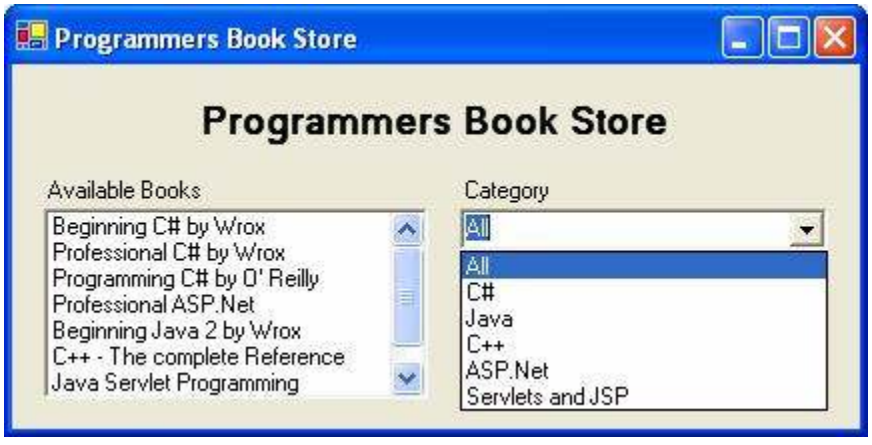
Download the source code of this sample from here

Food for thought: Exercise 13

1. What do you understand by the Disconnected Data Architecture of ADO.NET?
2. When using DataSet, you make changes (INSERT, UPDATE and DELETE) locally to the dataset. What is the use of it if the updates are not reflected back to the database immediately?
3. What are the advantages and disadvantages of using the disconnected data access architecture of ADO.NET?
4. When using the disconnected architecture, you can get the rows that have been modified in the current session and which are not yet saved in the back end database. We have not discussed how this is done in the lesson. Find out the method using MSDN or other resources on the internet.
5. In the examples of updating tables, we only considered the single table 'Article'. Modify the program so that the user can modify (INSERT, UPDATE, DELETE) records from both the 'Article' and 'Author' table.

Solution to Last Issue's Exercise (Exercise 12).

1. Write an application that presents the user a list of books in a list box and ask the user to select the category in a combo box. When the user selects a category, filter the list and show only the book that belongs to the selected category. The program should look like



Download Code

2. Write a program that shows the folder hierarchy of the C drive of your system in tree view. Use the System.IO namespace to get files and folders of your system. The Directory class has a method GetDirectories() which returns the list of directories in a particular path.

Download Code

3. Write a program that shows the files in the windows folder of your system in a list view. Use the System.IO namespace to get the files and folders of your system. The Directory class has a method GetFiles() which returns the list of files in a particular path.

Download Code

4. Extend the program in question 3 and add an icon with each file name. For convenience, you can use same icon for all files. Provide a toolbar that allows the user to change the view of the list view control (i.e., Large Icons, Small Icons, List, Details).

Download Code

5. Extend the program in question 4 by providing the options for sorting in toolbar. The options should include Sort by name and sort by extention (type). Also provide the toolbar options in the main menu.

Download Code

What's Next...

Next time we will explore multithreading in .NET. We will cover 1. What is multithreading? 2. Multithreading in VB.NET

- ThreadStart delegate
- Thread class functionality
- Running multiple threads in a program
- Common Operations with threads
- Thread Priority
- Controlling the execution of threads
- Interaction of multiple threads in a single program
- Thread Synchronization
- Thread Synchronization in VB.NET
- VB.NET Locking Mechanism
- Deadlocks

Multithreading in VB.NET

Lesson Plan

In this lesson we will learn how to achieve multithreading in VB.NET and .NET. We will start out by looking at what multithreading is and why we need it. Later we will demonstrate how we can implement multithreading in our VB.NET applications. Finally, we will learn about the different issues regarding thread synchronization and how VB.NET handles them.

What is Multithreading

Multithreading is a feature provided by the operating system that enables your application to have more than one execution path at the same time. We are all used to Windows' multitasking abilities, which allow us to execute more than one application at the same time. Just right now I am writing the 14th lesson of the Programmers Heaven's VB.NET School in Microsoft Word, listening to my favorite songs in WinAmp and downloading a new song using the Internet Download Manager. In a similar manner, we may use multithreading to run different methods of our program at the same time. Multithreading is such a common element of today's programming that it is difficult to find windows applications that don't use it. For example, Microsoft Word takes user input and displays it on the screen in one thread while it continues to check spelling and grammatical mistakes in the second thread, and at the same time the third thread saves the document automatically at regular intervals. In a similar manner, WinAmp plays music in one thread, displays visualizations in the second and takes user input in the third. This is quite different from multitasking as here a single application is doing multiple tasks at the same time, while in multitasking different applications execute at the same time.

Author's Note: When we say two or more applications or threads are running at the same time, we mean that they appear to execute at the same time, i.e. without one waiting for the termination of the other before starting. Technically, no two instructions can execute together at the same time on a single processor system (which most of us use). What the operating system does is divides the processor's execution time amongst the different applications (multitasking) and within an application amongst the different threads (multithreading).

Just consider the following small program:

```
Imports SystemModule Test
   Public Sub Main()
        Fun1()
        Fun2()
        Console.WriteLine("End of Main()")
    End Sub
    Public Sub Fun1()
        Dim i As Integer
        For i = 1 To 5
            Console.WriteLine("Fun1() writes: {0}", i)
        Next
```

```
    End Sub
    Public Sub Fun2()
        Dim i As Integer
        For i = 10 To 6 Step -1
            Console.WriteLine("Fun2() writes: {0}", i)
        Next
   End Sub
End Module
```

The output of the program is:

```
Fun1() writes: 1
Fun1() writes: 2
Fun1() writes: 3
Fun1() writes: 4
Fun1() writes: 5
Fun2() writes: 10
Fun2() writes: 9
Fun2() writes: 8
Fun2() writes: 7
Fun2() writes: 6
End of Main()
Press any key to continue
```

As we can see, the method Fun2() only started its execution when Fun1() had completed its execution. This is because when a method gets called, the execution control transfers to that method. And when the method returns the execution starts from the very next line of the code that called the method. i.e. the program implicitly has only one execution path. Using multithreading, we can define multiple concurrent execution paths within our program called threads. For example, we can use threads so that the two methods Fun1() and Fun2() may execute without waiting for each other to terminate.

Multithreading in VB.NET

The .NET Framework, and thus VB.NET provides full support for multiple execution threads in a program. You can add threading functionality to your application by using the System.Threading namespace. A thread in .NET is represented by the System.Threading.Thread class. We can create multiple threads in our program by creating multiple instances (objects) of this class. A thread starts its execution by calling the specified method and terminates when the execution of that method gets completed. We can specify the method name that the thread will call when it starts by passing a delegate of the ThreadStart type in the Thread class constructor. The delegate System.Threading.ThreadStart may reference any method which has the void return type and which takes no arguments.

```
Public Delegate Sub ThreadStart()
```

For example, we can change our previous application to run the two methods in two different threads like this:

```
Dim firstThread As New Thread(New ThreadStart(AddressOf Fun1))
```

```
Dim secondThread As New Thread(New ThreadStart(AddressOf Fun2))
```

Here we have created two instances of the Thread class and passed a ThreadStart type delegate in the constructor which references a method in our program. It is important that the method referenced in the Thread class constructor, through the ThreadStart delegate is parameter-less and has no return type. A thread does not start its execution when its object is created. Rather, we have to start the execution of a thread by calling the Start() method of the Thread class.

```
firstThread.Start()
secondThread.Start()
```

Here we have called the Start() method of the firstThread, which in turn will call the Fun1() method in a new thread. However this time the execution will not halt until the completion of the Fun1() method, but will immediately continue with the next statement which also starts the execution of the Fun2() method in a new thread. Again, the main thread of our application will not wait for the completion of the Fun2() method and will continue with the following statement. The complete source code for the application is:

```
Imports SystemImports System.ThreadingModule Test
    Public Sub Main()
        Dim firstThread As New Thread(New ThreadStart(AddressOf Fun1))
        Dim secondThread As New Thread(New ThreadStart(AddressOf Fun2))
        firstThread.Start()
        secondThread.Start()
        Console.WriteLine("End of Main()")
    End Sub
    Public Sub Fun1()
        Dim i As Integer
        For i = 1 To 5
            Console.WriteLine("Fun1() writes: {0}", i)
        Next
    End Sub
    Public Sub Fun2()
        Dim i As Integer
        For i = 10 To 6 Step -1
            Console.WriteLine("Fun2() writes: {0}", i)
        Next
    End Sub
End Module
```

One possible output of the program is:

```
End of Main()
Fun1() writes: 1
Fun1() writes: 2
Fun1() writes: 3
Fun1() writes: 4
Fun1() writes: 5
Fun2() writes: 10
Fun2() writes: 9
Fun2() writes: 8
Fun2() writes: 7
```

```
Fun2() writes: 6
Press any key to continue
```

Why did we say 'one possible output'? The reason is that we can't be sure about the execution sequence of the threads. Thread switching is completely Operating System dependent and may change each time you execute the program. Here what we notice is that the Main() thread ended before the start of any of the other two threads. However, after that the two functions seem to be calling in a sequence. What we might have expected was loop iterations of the two methods coming in a mixed way. So why didn't we get that output? In fact, the methods Fun1() and Fun2() have such short execution times that they get finished even before the switching of the two threads for a single time. If we increase the loop counters of these methods, we may notice the threads in execution.

Thread Functionality
The Thread class provides a number of useful methods and properties to control and manage thread execution.

Shared members of the System.Threading.Thread class
The shared property CurrentThread gives a reference to the currently executing thread. Another important shared member of the Thread class is the Sleep() method. It causes the currently executing thread to pause temporarily for the specified amount of time. The Thread.Sleep() method takes as an argument for the amount of time (in milliseconds) for which we want to pause the thread. For example, we can pause the currently executing thread for 1 second by passing 1000 as an argument to the Thread.Sleep() method.

```
Public Sub Main()
    Console.WriteLine("Before Calling the Thread.Sleep() method")
    Thread.Sleep(1000)    ' blocks the currently executing
                          ' thread (Main thread) for 1 second
    Console.WriteLine("After Calling the Thread.Sleep() method")
End Sub
```

When you execute the above program, you will notice a delay of 1 second between the printing of the two lines.

Instance members of the System.Threaing.Thread class
The most commonly used instance members of the thread class are:

Member	Description
Name	A property of string type used to get/set the friendly name of the thread instance.
Priority	A property of type System.Threading.ThreadPriority. This property is use to get/set the value indicating the scheduling priority of the thread. The priority can be any instance of the ThreadPriority enumeration which includes AboveNormal, BelowNormal, Normal, Highest and Lowest.
IsAlive	A Boolean property indicating whether the thread is alive or has been terminated.

ThreadState	A property of type System.Threading.ThreadState. This property is used to get the value containing the state of the thread. The value returned by this property is an instance of the ThreadState enumeration which includes Aborted, AbortRequested, Suspended, Stopped, Unstarted, Running, WaitSleepJoin, etc
Start()	Starts the execution of the thread
Abort()	Allows the current thread to stop the execution of a thread permanently. The method throws the ThreadAbortException in the executing thread.
Suspend()	Pauses the execution of a thread temporarily
Resume()	Resumes the execution of a suspended thread
Join()	Makes the current thread wait for another thread to finish

[/size] [/font]

Thread Demonstration Example - Basic Operations
Now we will start to understand the implementation of threads in VB.NET. Review the following VB.NET Console program:

```
Imports SystemImports System.ThreadingModule Test
    Dim mainThread As Thread
    Dim firstThread As Thread
    Dim secondThread As Thread
    Public Sub Main()
        mainThread = Thread.CurrentThread
        firstThread = New Thread(New ThreadStart(AddressOf Fun1))
        secondThread = New Thread(New ThreadStart(AddressOf Fun2))
        mainThread.Name = "Main Thread"
        firstThread.Name = "First Thread"
        secondThread.Name = "Second Thread"
        ThreadsInfo("Main() before starting the threads")
        firstThread.Start()         secondThread.Start()
        ThreadsInfo("Main() just before exiting the Main()")
    End Sub
    Public Sub ThreadsInfo(ByVal location As String)
        Console.WriteLine(vbCrLf + "In {0}", location)
        Console.WriteLine("Thread Name: {0}, ThreadState: {1}", _
                          mainThread.Name, mainThread.ThreadState)
        Console.WriteLine("Thread Name: {0}, ThreadState: {1}", _
                          firstThread.Name, firstThread.ThreadState)
        Console.WriteLine("Thread Name: {0}, ThreadState: {1}" + vbCrLf, _
                          secondThread.Name, secondThread.ThreadState)
    End Sub
    Public Sub Fun1()
        Dim i As Integer
        For i = 1 To 5
            Console.WriteLine("Fun1() writes: {0}", i)
            Thread.Sleep(100)
        Next
        ThreadsInfo("Fun1()")
    End Sub
    Public Sub Fun2()
        Dim i As Integer
        For i = 10 To 6 Step -1
            Console.WriteLine("Fun2() writes: {0}", i)
            Thread.Sleep(125)
        Next
```

```
        ThreadsInfo("Fun2()")
    End Sub
End Module
```

First of all we have defined three references of type System.Threading.Thread to reference the three threads (main, first and second thread) later in the Main() method:

```
Dim mainThread As Thread
Dim firstThread As Thread
Dim secondThread As Thread
```

We have defined a method called ThreadsInfo() to display the information (name and state) of the three threads. The two methods Fun1() and Fun2() are similar to the previous program and just print 5 numbers. In the loop of these methods we have called the Sleep() method which will make the thread executing the method suspend for the specified amount of time. We have set slightly different times in each the threads' Sleep() methods. After the loop, we have printed the information about all the threads again.

```
    Public Sub Fun2()
        Dim i As Integer
        For i = 10 To 6 Step -1
            Console.WriteLine("Fun2() writes: {0}", i)
            Thread.Sleep(125)
        Next
        ThreadsInfo("Fun2()")
    End Sub
```

Inside the Main() method we first instantiated the two thread instances (firstThread and secondThread) by passing constructors the references of the Fun1() and Fun2() methods respectively using the ThreadStart delegate. We also made the reference mainThread point to the thread executing the Main() method by using the Thread.CurrentThread property in the Main() method.

```
    Public Sub Main()
        mainThread = Thread.CurrentThread
        firstThread = New Thread(New ThreadStart(AddressOf Fun1))
        secondThread = New Thread(New ThreadStart(AddressOf Fun2))
```

We then set the Name property of these threads to the threads corresponding names.

```
        mainThread.Name = "Main Thread"
        firstThread.Name = "First Thread"
        secondThread.Name = "Second Thread"
```

After setting the names, we printed the current state of the three threads by calling the static ThreadsInfo() method, started the two threads and finally called the ThreadsInfo() method just before the end of the Main() method.

```
        ThreadsInfo("Main() before starting the threads")
        firstThread.Start()
        secondThread.Start()
        ThreadsInfo("Main() just before exiting the Main()")
```

One possible output of the program is:

In Main() before starting the threads
Thread Name: Main Thread, ThreadState: Running
Thread Name: First Thread, ThreadState: Unstarted
Thread Name: Second Thread, ThreadState: Unstarted

In Main() just before exiting the Main()

```
Thread Name: Main Thread, ThreadState: Running
Thread Name: First Thread, ThreadState: Unstarted
Thread Name: Second Thread, ThreadState: Unstarted

Fun1() writes: 1
Fun2() writes: 10
Fun1() writes: 2
Fun2() writes: 9
Fun1() writes: 3
Fun2() writes: 8
Fun1() writes: 4
Fun2() writes: 7
Fun1() writes: 5

In Fun1()
Thread Name: Main Thread, ThreadState: Background, Stopped, WaitSleepJoin
Thread Name: First Thread, ThreadState: Running
Thread Name: Second Thread, ThreadState: WaitSleepJoin

Fun2() writes: 6

In Fun2()
Thread Name: Main Thread, ThreadState: Background, Stopped, WaitSleepJoin
Thread Name: First Thread, ThreadState: Stopped
Thread Name: Second Thread, ThreadState: Running

Press any key to continue
```

The important thing to note here is the sequence of execution and the thread states at different points during the execution of the program. The two threads (firstThread and secondThread) didn't get started even when the Main() method was exiting. At the end of firstThread, the Main() thread has stopped while the secondThread is in the Sleep state.

Thread Demonstration Example - Thread Priority

When two or more threads are executing simultaneously they share the processor time. In normal conditions, the operating system tries to distribute the processor time equally amongst the threads of a process. However, if we wish to influence how processor time is distributed, we can also specify the priority level for our threads. In .NET we do this using the System.Threading.ThreadPriority enumeration, which contains Normal, AboveNormal, BelowNormal, Highest and Lowest. The default priority level of a thread is, to no one's surprise, Normal. A thread with a higher priority is given more time by Operating System than a thread with a lower priority. Consider the program below with no priority setting:

```
Imports SystemImports System.ThreadingModule Test
    Public Sub Main()
        Dim firstThread As New Thread(New ThreadStart(AddressOf Fun1))
        Dim secondThread As New Thread(New ThreadStart(AddressOf Fun2))
        firstThread.Name = "First Thread"
        secondThread.Name = "Second Thread"
        firstThread.Start()
        secondThread.Start()
```

```
    End Sub
    Public Sub Fun1()
        Dim i, j As Integer
        For i = 1 To 10
            Dim t As Integer = New Random().Next(20)
            For j = 0 To t
                Dim s As New String(New Char() {})
            Next
            Console.WriteLine(Thread.CurrentThread.Name + _
                        ": created: " + t.ToString() + " empty strings")
        Next
    End Sub
    Public Sub Fun2()
        Dim i, j As Integer
        For i = 20 To 11 Step -1
            Dim t As Integer = New Random().Next(20)
            For j = 0 To t
               Dim s As New String(New Char() {})
            Next
            Console.WriteLine(Thread.CurrentThread.Name + _
                        ": created: " + t.ToString() + " empty strings")
        Next
    End Sub
End Module
```

Here we have asked the runtime to create an almost similar number of objects in the two thread methods (Fun1() and Fun2()). One possible output of the program is:

```
First Thread: created: 18 empty strings
First Thread: created: 5 empty strings
First Thread: created: 5 empty strings
First Thread: created: 5 empty strings
First Thread: created: 5 empty strings
First Thread: created: 5 empty strings
First Thread: created: 5 empty strings
First Thread: created: 5 empty strings
First Thread: created: 5 empty strings
First Thread: created: 5 empty strings
Second Thread: created: 16 empty strings
Second Thread: created: 5 empty strings
Second Thread: created: 5 empty strings
Second Thread: created: 5 empty strings
Second Thread: created: 5 empty strings
Second Thread: created: 5 empty strings
Second Thread: created: 5 empty strings
Second Thread: created: 5 empty strings
Second Thread: created: 5 empty strings
Second Thread: created: 5 empty strings
Press any key to continue
```

Now we will change the priority of secondThread to AboveNormal:

```
Imports SystemImports System.ThreadingModule Test
    Public Sub Main()
        Dim firstThread As New Thread(New ThreadStart(AddressOf Fun1))
        Dim secondThread As New Thread(New ThreadStart(AddressOf Fun2))
        firstThread.Name = "First Thread"
        secondThread.Name = "Second Thread"
        secondThread.Priority = ThreadPriority.AboveNormal
```

```
        firstThread.Start()
        secondThread.Start()
    End Sub
    Public Sub Fun1()
        Dim i, j As Integer
        For i = 1 To 10
            Dim t As Integer = New Random().Next(20)
            For j = 0 To t
                Dim s As New String(New Char() {})
            Next
            Console.WriteLine(Thread.CurrentThread.Name + _
                        ": created: " + t.ToString() + " empty strings")
        Next
    End Sub
    Public Sub Fun2()
        Dim i, j As Integer
        For i = 20 To 11 Step -1
            Dim t As Integer = New Random().Next(40)
            For j = 0 To t
                Dim s As New String(New Char() {})
            Next
            Console.WriteLine(Thread.CurrentThread.Name + _
                        ": created: " + t.ToString() + " empty strings")
        Next
    End Sub
End Module
```

Here we have made two changes. We have increased the priority of the secondThread to AboveNormal. We have also increased the range for random numbers so that the second thread would be required to create a greater number of objects. On compiling and executing the program, we get output like:

```
Second Thread: created: 14 empty strings
Second Thread: created: 18 empty strings
Second Thread: created: 18 empty strings
Second Thread: created: 18 empty strings
Second Thread: created: 27 empty strings
Second Thread: created: 27 empty strings
Second Thread: created: 27 empty strings
Second Thread: created: 27 empty strings
Second Thread: created: 27 empty strings
Second Thread: created: 27 empty strings
First Thread: created: 13 empty strings
First Thread: created: 13 empty strings
First Thread: created: 13 empty strings
First Thread: created: 13 empty strings
First Thread: created: 13 empty strings
First Thread: created: 13 empty strings
First Thread: created: 13 empty strings
First Thread: created: 13 empty strings
First Thread: created: 13 empty strings
First Thread: created: 13 empty strings
Press any key to continue
```

Consider the above output. Although the second thread is creating more objects, it still finishes before the first thread. The reason simply is that now the priority level of second thread is higher than that of the first thread.

Thread Demonstration Example - Thread Execution Control

The Thread class also provides methods for controlling the execution of different threads. You can start, stop (called abort), suspend and resume suspended threads just by calling a single method on the Thread object. Consider the following demonstration application:

```
Imports SystemImports System.ThreadingModule Test
    Public Sub Main()
        Dim firstThread As New Thread(New ThreadStart(AddressOf Fun1))
        firstThread.Start()
        Console.WriteLine("Thread started")
        Thread.Sleep(150)
        firstThread.Suspend()
        Console.WriteLine("Thread suspended")
        Thread.Sleep(150)
        firstThread.Resume()
        Console.WriteLine("Thread resumed")
        Thread.Sleep(150)
        firstThread.Abort()
        Console.WriteLine("Thread aborted")
    End Sub
    Public Sub Fun1()
        Dim i As Integer
        Try
            For i = 1 To 20
                Dim t As Integer = New Random().Next(20, 50)
                Console.WriteLine("Thread 1: slept for: " + _
                                  t.ToString() + " milliseconds")
                Thread.Sleep(t)
            Next
        Catch ex As ThreadAbortException
            Console.WriteLine("Thread 1 aborted in iteration number: {0}", i)
        End Try
    End Sub
End Module
```

Here we have started, suspended, resumed and aborted the thread (firstThread) with a constant difference of 150 milliseconds. Remember that when a thread is aborted the runtime throws the ThreadAbortException in the thread method. This exception allows our thread to perform some cleanup work before it's termination, e.g. closing the opened database, network or file connections. When we execute the above program, we see the following output:

```
Thread started
Thread 1: slept for: 35 milliseconds
Thread 1: slept for: 29 milliseconds
Thread 1: slept for: 49 milliseconds
Thread 1: slept for: 36 milliseconds
Thread 1: slept for: 33 milliseconds
Thread 1: slept for: 30 milliseconds
Thread suspended
Thread resumed
Thread 1: slept for: 44 milliseconds
Thread 1: slept for: 48 milliseconds
Thread aborted
Press any key to continue
```

Note that there is no thread activity between thread suspend and resume. A word of caution: If you try to call the Suspend(), Resume() or Abort() methods on a non-running thread (aborted or un-started), you will get the ThreadStateException.

Using Join() to wait for running threads

Finally, you can make a thread wait for other running threads to complete by calling the Join() method. Consider this simple code:

```
Imports SystemImports System.ThreadingModule Test
    Public Sub Main()
        Dim firstThread As New Thread(New ThreadStart(AddressOf Fun1))
        Dim secondThread As New Thread(New ThreadStart(AddressOf Fun2))
        firstThread.Start()
        secondThread.Start()
        Console.WriteLine("End of Main()")
    End Sub
    Public Sub Fun1()
        Dim i As Integer
        For i = 1 To 5
            Console.WriteLine("Fun1() writes: {0}", i)
        Next
    End Sub
    Public Sub Fun2()
        Dim i As Integer
        For i = 10 To 6 Step -1
            Console.WriteLine("Fun2() writes: {0}", i)
        Next
    End Sub
End Module
```

In the above code, the thread of the Main() method will terminate quickly after starting the two threads. The output of the program will look like this:

```
Thread 1 writes: 1
Thread 2 writes: 10
Ending Main()
Thread 1 writes: 2
Thread 1 writes: 3
Thread 1 writes: 4
Thread 1 writes: 5
Thread 2 writes: 9
Thread 2 writes: 8
Thread 2 writes: 7
Thread 2 writes: 6
Thread 2 writes: 5
Press any key to continue
```

But if we like to keep our Main() thread alive until the first thread is alive, we can apply Join() method to it.

```
Imports SystemImports System.ThreadingModule Test
    Public Sub Main()
        Dim firstThread As New Thread(New ThreadStart(AddressOf Fun1))
        Dim secondThread As New Thread(New ThreadStart(AddressOf Fun2))
        firstThread.Start()
        secondThread.Start()
        firstThread.Join()
```

```
        Console.WriteLine("End of Main()")
    End Sub
    Public Sub Fun1()
        Dim i As Integer
        For i = 1 To 5
            Console.WriteLine("Fun1() writes: {0}", i)
        Next
    End Sub
    Public Sub Fun2()
        Dim i As Integer
        For i = 15 To 6 Step -1
            Console.WriteLine("Fun2() writes: {0}", i)
        Next
    End Sub
End Module
```

Here we have inserted the call to the Join() method of firstThread and increased the loop counter for secondThread. Now the Main() method thread will not terminate until the first thread is alive. One possible output of the program is:

```
Thread 1 writes: 1
Thread 1 writes: 2
Thread 1 writes: 3
Thread 1 writes: 4
Thread 1 writes: 5
Ending Main()
Thread 2 writes: 15
Thread 2 writes: 14
Thread 2 writes: 13
Thread 2 writes: 12
Thread 2 writes: 11
Thread 2 writes: 10
Thread 2 writes: 9
Thread 2 writes: 8
Thread 2 writes: 7
Thread 2 writes: 6
Press any key to continue
```

Author's Note: Since our threads are doing such little work, you might not get the exact output. You may get the Main() thread exiting at the end of both the threads. To see the real effect of threads competition, increase the loop counters to hundreds in the examples of this lesson.

Thread Synchronization

So far we have seen the positive aspects of using multiple threads. In all of the programs presented so far, the threads of a program were not sharing any common resources (objects).

The threads were only using the local variables of their corresponding methods. But what happens when multiple threads try to access the same shared resource? The problem is like she and I share the same television remote control. I want to switch to the sports channel, record the cricket match and switch off the television while she wants to switch to the music channel and record the music. When it is my turn, I switch to sports channel but,

unfortunately, my time slice ends. She takes the remote and switches to the music channel, but then her time slice also ends. Now I continue from my last action and start recording and then switch off the television. What would be the end result? I would have ended up with a recording of a live concert instead of the cricket match. The situation would be worse at her side, she would be attempting to record from a switched off television! The same happens with multiple threads accessing a single shared resource (object).

The state of the object may get changed during consecutive time slices without the knowledge of the thread. Still can't get the point? Let's take a more technical example; suppose thread 1 gets a DataRow from a DataTable and starts updating its column values. At the same time, thread 2 starts and also accesses the same DataRow object to update its column values. Both the threads save the Data Row back to the table and physical database. But which Data Row version has been saved to the database? The one updated by thread 1 or the one updated by thread 2? We can't predict the actual result with any certainty. It can be the one update by thread 1 or the one update by thread 2 or it may be the mixture of both of these updates… Who will like to have such a situation?

So what is the solution? Well the simplest solution is not to use shared objects with multiple threads. This might sound funny, but this is what most the programmers practice. They avoid using shared objects with multiple threads executing simultaneously. But in some cases, it is desirable to use shared objects with multiple threads. .NET provides a locking mechanism to avoid accidental simultaneous access by multiple threads to the same shared object.

The VB.NET Locking Mechanism
Consider the following code:

Imports System Imports System.Text Imports System.Threading

```
Module Test
    Dim text As New StringBuilder()
    Public Sub Main()
        Dim firstThread As New Thread(New ThreadStart(AddressOf Fun1))
        Dim secondThread As New Thread(New ThreadStart(AddressOf Fun2))
        firstThread.Start()
        secondThread.Start()
        firstThread.Join()
        secondThread.Join()
        Console.WriteLine("Text is: {0}{1}", vbCrLf, text.ToString())
    End Sub
    Public Sub Fun1()
        Dim i As Integer
        For i = 1 To 20
            Thread.Sleep(10)
            text.Append(i.ToString() + " ")
        Next
    End Sub
    Public Sub Fun2()
        Dim i As Integer
        For i = 21 To 40
            Thread.Sleep(2)
            text.Append(i.ToString() + " ")
        Next
```

```
    End Sub
End Module
```

Both the threads are appending numbers to the shared string builder (System.Text.StringBuilder) object. As a result in the output, the final string would be something like:

```
Text is:
21 1 22 2 23 3 24 4 25 5 26 6 27 7 28 8 29 9 10 30 31 11
32 12 33 13 34 14 35 15 36 16 37 17 38 18 39 19 40 20
Press any key to continue
```

The final text is an unordered sequence of numbers. To avoid threads interfering with each others results, each thread should lock the shared object before it starts appending the numbers and release the lock when it is done. While the object is locked, no other thread should be allowed to change the state of the object. This is exactly what the VB.NET SyncLock keyword does. We can change the above code to provide the locking functionality:

Imports System Imports System.Text Imports System.Threading

```
Module Test
    Dim text As New StringBuilder()
    Public Sub Main()
        Dim firstThread As New Thread(New ThreadStart(AddressOf Fun1))
        Dim secondThread As New Thread(New ThreadStart(AddressOf Fun2))
        firstThread.Start()
        secondThread.Start()
        firstThread.Join()
        secondThread.Join()
        Console.WriteLine("Text is: {0}{1}", vbCrLf, text.ToString())
    End Sub
    Public Sub Fun1()
        Dim i As Integer
        SyncLock text
            For i = 1 To 20
                Thread.Sleep(10)
                text.Append(i.ToString() + " ")
            Next
        End SyncLock
    End Sub
    Public Sub Fun2()
        Dim i As Integer
        SyncLock text
           For i = 21 To 40
                Thread.Sleep(2)
                text.Append(i.ToString() + " ")
            Next
        End SyncLock
    End Sub
End Module
```

Note that now each thread is locking the shared object 'text' before working on it. Hence, the output of the program will be:

```
Text is:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
```

```
Press any key to continue
```

Or,

```
Text is:
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38
39 40 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Press any key to continue
```

You will see the first output when the firstThread will succeed to lock the 'text' object first while the second output shows that the secondThread has succeeded in locking the 'text' object first.

Threads may cause Deadlock

Another issue regarding thread synchronization is deadlock. Consider there are two shared objects A and B. Thread 1 first locks A and then B

```
' code for Thread 1SyncLock A
        SyncLock (B)
                ' do some work
        End SyncLock
End SyncLock
```

While the second thread locks B and then A

```
' code for Thread 2
SyncLock B
        SyncLock A
                ' do some work
        End SyncLock
End SyncLock
```

Suppose the two threads start at the same time and following execution sequence is generated by the Operating System

- Thread 1 takes lock on A
- Thread 2 takes lock on B
- Thread 1 waits for the lock on B held by Thread 2...
- Thread 2 waits for the lock on A held by Thread 1...

Thread 1 will not leave the lock on A until it has got the lock on B and has finished its work. On the other hand, Thread 2 will not leave the lock on B until it has got the lock on A and has finished its work, which will never happen! Hence the program will be deadlocked at this stage. How to avoid deadlock is a matter of extensive research. These kinds of deadlocks also occur in database servers and operating systems. Hence, one must be very careful when using multiple locks. This kind of bugs are really very hard to detect and correct as there is no exception thrown and no error is reported at runtime; the program just stops responding, leaving you in an abysmal situation. To experience a deadlock yourself, compile and execute the following program.

```
Imports System
Imports System.Text
Imports System.Threading
Module Test
    Dim text As New StringBuilder()
    Dim doc As New StringBuilder()
    Public Sub Main()
        Dim firstThread As New Thread(New ThreadStart(AddressOf Fun1))
```

```
        Dim secondThread As New Thread(New ThreadStart(AddressOf
Fun2))
        firstThread.Start()
        secondThread.Start()
    End Sub
    Public Sub Fun1()
        Dim i As Integer
        SyncLock text
            Thread.Sleep(10)
            For i = 1 To 20
                text.Append(i.ToString() + " ")
            Next
            SyncLock doc
                doc.Append(text.ToString())
                For i = 1 To 20
                    doc.Append(i.ToString() + " ")
                Next
            End SyncLock
        End SyncLock
    End Sub
    Public Sub Fun2()
        Dim i As Integer
        SyncLock doc
            Thread.Sleep(10)
            For i = 1 To 20
                doc.Append(i.ToString() + " ")
            Next
            SyncLock text
                text.Append(doc.ToString())
                For i = 1 To 20
                    text.Append(i.ToString() + " ")
                Next
            End SyncLock
        End SyncLock
    End Sub
End Module
```

Here we have used two shared objects (text and doc). Both the threads are attempting to lock the two objects in the opposite order. If in a particular run, both the threads lock their first object at the same time, a deadlock is bound to occur. You might not find a deadlock in the first run. But on executing the program again and again, you will surely come up against a deadlocked situation. In this case, the program will hang and stop responding.

Food for thought: Exercise 14

1. What is the difference between multitasking and multithreading?
2. What is the difference between a process and a thread?
3. Why and when should one use multiple threads (multi-threading) instead of multitasking?
4. What is the difference between the Abort(), Suspend() and Sleep() methods of a Thread class?
5. Why is the ThreadAbortException thrown in the thread method when the Abort() method is called for that thread? Is it an error to call the Abort() method?
6. What are the advantages and disadvantages of accessing shared objects with multiple threads?
7. Architectural Issue: Those of you who have some experience of the Java programming language must have observed the difference between Java and VB.NET's approach to

implementing threads. In Java, the user inherits the java.lang.Thread class and overrides its Run() method while in VB.NET we create an object of the Thread class and pass it a delegate (function pointer) to the entry point for the thread. Compare the two approaches.

Solution to Last Issue's Exercise (Exercise 13)

1. What do you understand by the Disconnected Data Architecture of ADO.NET?
The disconnected data architecture of ADO.NET provides a local buffer to store the data fetched from the database. Changes are made to data locally in the DataSet. It connects to the database only when it needs to fetch some data or update the changes to the database, and then disconnects again.

Since most of the time the DataSet remains disconnected from the database, the architecture is referred to as the disconnected data architecture. The DataSet gives to the DataTables collection to store the tables fetched from the database and the Relations collection to keep track of the relationships between these tables. Each table contains collections of its Rows and Columns. Each column can be used to hold values of specific data types that directly map to the database types (e.g., types in the System.Data.SqlTypes namespace map to MS-SQL Server types). Also different constraints can be applied to each column to make sure that only valid data is entered into the field.

2. When using a DataSet, you make changes (INSERT, UPDATE and DELETE) locally to the dataset. What is the use of it if the updates are not reflected back to the database immediately?
Well, it depends on the nature of the application. If multiple users are using the database and the database needs to be updated every time, you must not use the DataSet. For this, .NET provides the connection oriented architecture. In scenarios where an instant update of the database is not required, the DataSset provides optimal performance by making the changes locally and connecting to database later to update a whole batch of data. This also reduces the network bandwidth required if the database is accessed through network.

3. What are the advantages and disadvantages of using the disconnected data access architecture of ADO.NET?
Disconnected data access is suited most to read only services. In common practice clients are often interested in reading and displaying data. In this type of situation, disconnected data access excels as it fetches all of the data in a single go and stores it in the local buffer (DataSet). This local storage of data eliminates the need to stay connected to the database and fetching single records at a time. On the down side, the disconnected data access architecture is not designed to be used in a networked environment where multiple users are updating data simultaneously as each of them needs to be aware of current state of the

database at any time (e.g., an Airline Reservation System).

4. When using the disconnected architecture, you can get the rows that have been modified in the current session and which have not yet been saved in the back end database. We have not discussed how this is done in the lesson. Find out the method using MSDN and other resources on the internet.

To check an individual row for its state, you can use the RowState property of the DataRow class. The type of RowState property is the DataRowState enumeration, whose values include Added, Modified, Deleted, Unchanged and Detached. For example, you can check if the row has been modified or not like this:

```
' DataRow someRow... defined earlier
If someRow.RowState = DataRowState.Modified Then
        ' take necessary actions
End If
```

Or if you are presenting a data grid to add, modify, delete and view data; you may use the RowStateFilter property of the DataView class which is of type DataViewRowState (an enumeration). This property returns the collection of all the rows which are modified, added or deleted.

Download Code

5. In the examples regarding updating tables, we only considered the single table 'Article'. Modify the program so that the user can modify (INSERT, UPDATE, DELETE) records from both the 'Article' and 'Author' table.

Download Code

What's Next...

Next time we will explore files and stream handling in .NET. We will cover

- Working with the Windows File System
- Working with Drives (getting a list of all the drives on the system, getting system drive and reading other properties of the drive)
- Working with Files (creating, copying, moving, deleting and reading various information about files)
- Working with Directories or Folders (creating, copying, moving, deleting and reading various properties of folders)
- Streams: An introduction
- An overview of different types of streams
- Exploring file streams (an overview of different types of file streams)
- Reading and writing data to files.
- Serialization and De-serialization in .Net and VB.Net
- How can we serialize our objects?

- Demonstrating the serialization and de-serialization of objects

Working With The File System & Streams

Lesson Plan

In this lesson we will learn how we can manipulate the Windows file system and different types of streams using VB.NET and .NET. We will start out by looking at how we can manipulate physical drives, folders and files, and perform different operations on them. In the second half of the lesson, we will explore the different types of streams used in .NET and see how we can read data from and write data to files. Finally we will learn about the concept of serialization of objects and how we can implement serialization in VB.NET. The lesson also contains a supplementary topic on Asynchronous I/O.

Working with the File System

This section is further divided into three parts. In the first part, we will see how we can get the software system's environment information, e.g. the path to the Windows System folder & the Program Files folder, current user name, operating system version, etc. In the second part, we will learn how to perform common file operations such as copying, moving, deleting, renaming and more. In the last part of this section, we will explore directory (or Folder) manipulation techniques.

Obtaining the Application's Environment Information - The System.Environment class

The System.Environment class is the base class used to obtain information about the environment of our application. The description of some of its properties and methods is presented in the following table:

Member	Description
CurrentDirectory	Gets or sets the directory name from which the process was started
MachineName	Gets the name of computer on which the process is currently executing
OSVersion	Returns the version of current Operating System
UserName	Returns the user name of the user executing this process
Version	Returns a System.Version object representing the complete version information of the common language runtime (CLR)
Exit()	Terminates the current process, returning the exit code to the Operating System
GetFolderPath()	Returns the complete path of various standard folders of the windows operating system like Program files, My documents, Start Menu and etc.
GetLogicalDrives() Returns an array of type string containing the list of all drives present in the current system.	

Demonstration Application - Environment Information

Let's make a demonstration application that uses the above mentioned properties and methods to display environment information. It is a windows application project that contains a list box named 'lbx' which is used to display the information, a button named 'btnGo' which will be used to start fetching the information and a button named 'btnExit' which terminates the application. The main logic is present inside the Go button's event handler:

```
    Private Sub btnGo_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles btnGo.Click
        Dim os As OperatingSystem = Environment.OSVersion
        Dim OSid As PlatformID = os.Platform
        Dim drives() As String = Environment.GetLogicalDrives()
        Dim drivesString As String = ""
        Dim drive As String
        For Each drive In drives
            drivesString += drive + ", "
        Next
        drivesString = drivesString.TrimEnd(" "c, ","c)
        lbx.Items.Add("Machine Name:
            " + Environment.MachineName)
        lbx.Items.Add("Operating System: " + Environment.OSVersion.ToString())
        lbx.Items.Add("Operating System ID:  " + OSid.ToString())
        lbx.Items.Add("Current Folder:          " + Environment.CurrentDirectory)
        lbx.Items.Add("CLR Version:             " + Environment.Version.ToString())
        lbx.Items.Add("Present Drives:          " + drivesString)
    End Sub
```

Here we have simply retrieved the environment information from the public properties and methods and added them to the list box. A few important things to note here are:

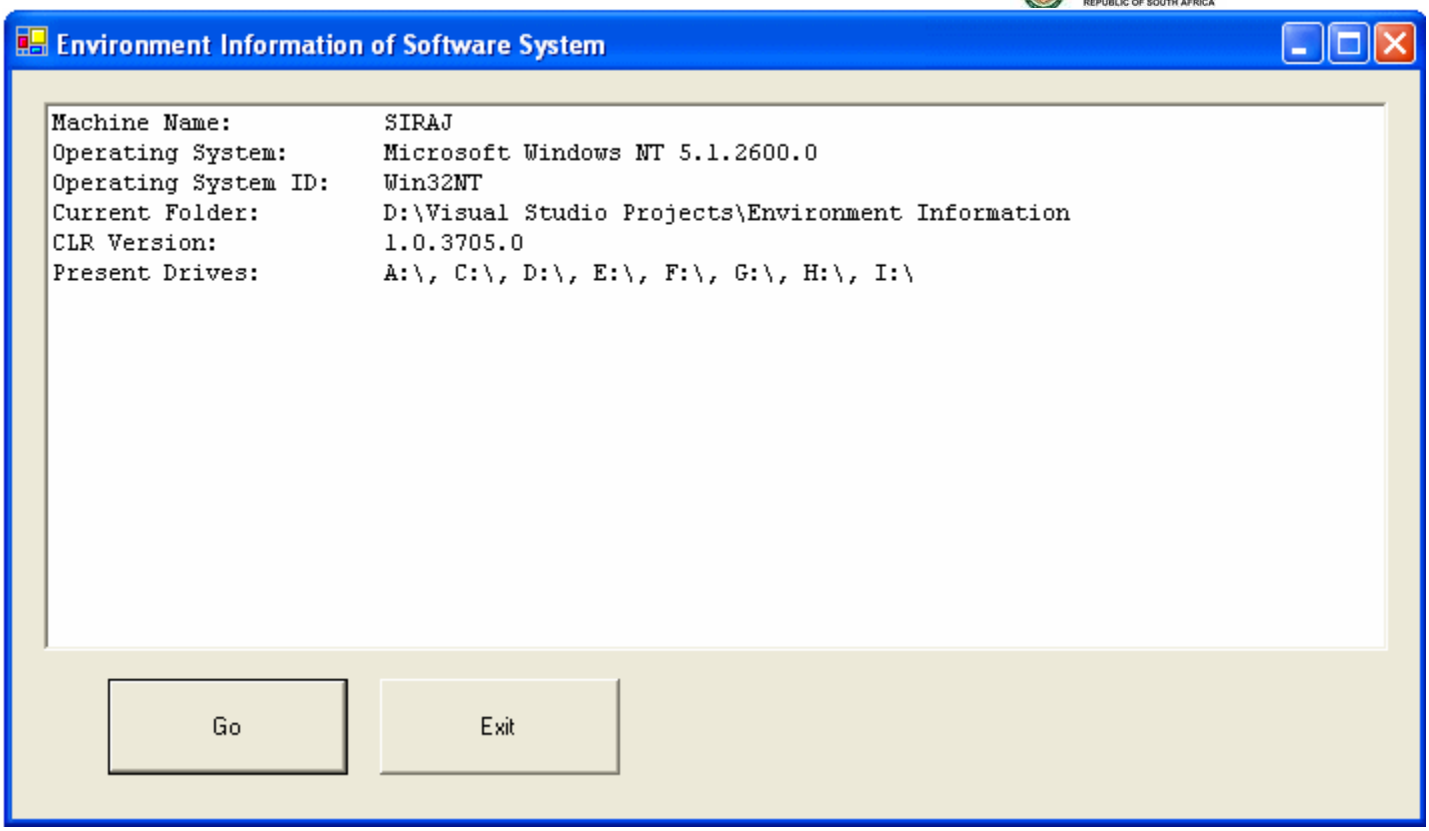
- Environment.GetLogicalDrives() returns an array of strings, with each string representing the drive name.
- Environment.Version returns an object of type System.Version which contains the detailed information about the current version of the common language runtime (CLR).

The event handler for the exit button simply calls the Exit() method of the Environment class to terminate the current process.

```
    Private Sub btnExit_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles btnExit.Click
        Environment.Exit(0)
    End Sub
```

Here we have passed zero to the Exit() method. This value will be returned to the Operating System and can be used to check whether the program terminates successfully or not.

When I executed this program to my system, I got the following result:



Obtaining the paths of various Windows Standard folders - Environment.GetFolderPath()

The method Environment.GetFolderPath() can be used to get the complete paths of various windows standard folders on the current machine. The only argument passed to the method is a value from the System.Environment.SpecialFolder enumeration. Some of the more common members of this enumeration are:

Member	Description
ProgramFiles	The program files folder where programs are usually installed
CommonProgramFiles	The Common Files folder of Program Files.
DesktopDirectory	The folder representing the desktop of user
Favorites	The Favorites folder to store favorite links
History	The History folder to store history files
Personal	The My Documents folder
Programs	The folder representing the Programs menu of Start Menu
Recent	The Recent folder
SendTo	The Send To folder
StartMenu	The Start menu folder

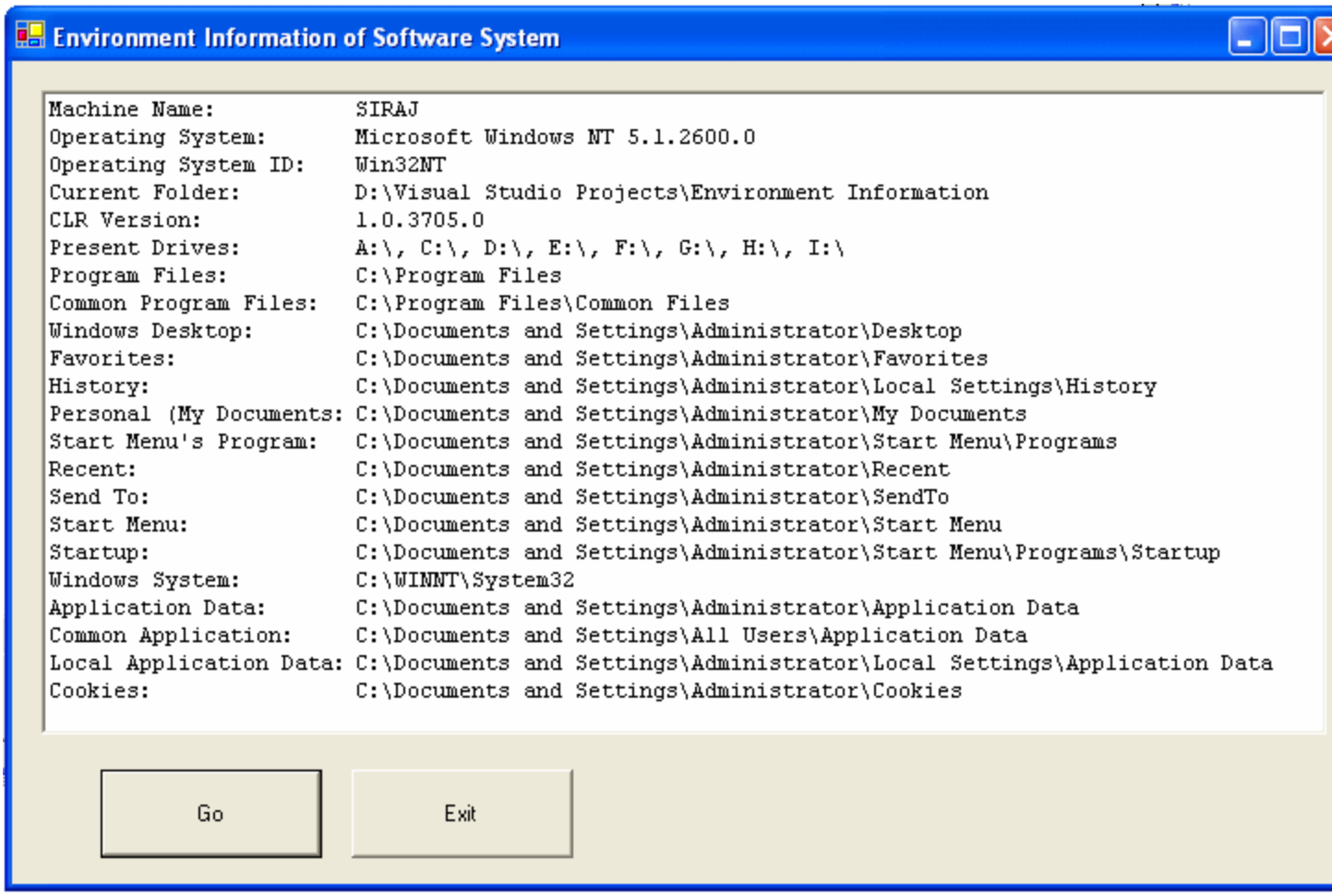
Startup	Folder of the Startup menu on the Start>>Programs menu
System	The System folder of Windows folder
ApplicationData	The Application Data folder
CommonApplicationData	The Common Application Data folder
LocalApplicationData	The Local Application Data folder
Cookies	The folder used to store cookies

Let's use these in a simple program to understand their functionality. We can modify the previous program to include these results by changing the btnGo_Click method to:

```
    Private Sub btnGo_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles btnGo.Click
        Dim os As OperatingSystem = Environment.OSVersion
        Dim OSid As PlatformID = os.Platform
        Dim drives() As String = Environment.GetLogicalDrives()
        Dim drivesString As String = ""
        Dim drive As String
        For Each drive In drives
            drivesString += drive + ", "
        Next
        drivesString = drivesString.TrimEnd(" "c, ","c)
        lbx.Items.Add("Machine Name:            " + Environment.MachineName)
        lbx.Items.Add("Operating System:        " + Environment.OSVersion.ToString())
        lbx.Items.Add("Operating System ID:     " + OSid.ToString())
        lbx.Items.Add("Current Folder:          " + Environment.CurrentDirectory)
        lbx.Items.Add("CLR Version:             " + Environment.Version.ToString())
        lbx.Items.Add("Present Drives:          " + drivesString)
        lbx.Items.Add("Program Files:           " + _
            Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles))
        lbx.Items.Add("Common Program Files:    " + _
            Environment.GetFolderPath(Environment.SpecialFolder.CommonProgramFiles))
        lbx.Items.Add("Windows Desktop:         " + _
            Environment.GetFolderPath(Environment.SpecialFolder.DesktopDirectory))
        lbx.Items.Add("Favorites:               " + _
            Environment.GetFolderPath(Environment.SpecialFolder.Favorites))
        lbx.Items.Add("History:                 " + _
            Environment.GetFolderPath(Environment.SpecialFolder.History))
        lbx.Items.Add("Personal (My Documents:  " + _
            Environment.GetFolderPath(Environment.SpecialFolder.Personal))
        lbx.Items.Add("Start Menu's Program:    " + _
            Environment.GetFolderPath(Environment.SpecialFolder.Programs))
        lbx.Items.Add("Recent:                  " + _
            Environment.GetFolderPath(Environment.SpecialFolder.Recent))
        lbx.Items.Add("Send To:                 " + _
        Environment.GetFolderPath(Environment.SpecialFolder.SendTo))
        lbx.Items.Add("Start Menu:              " + _
            Environment.GetFolderPath(Environment.SpecialFolder.StartMenu))
        lbx.Items.Add("Startup:                 " + _
            Environment.GetFolderPath(Environment.SpecialFolder.Startup))
        lbx.Items.Add("Windows System:          " + _
            Environment.GetFolderPath(Environment.SpecialFolder.System))
        lbx.Items.Add("Application Data:        " + _
            Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData))
        lbx.Items.Add("Common Application:      " + _
```

```
Environment.GetFolderPath(Environment.SpecialFolder.CommonApplicationData))
        lbx.Items.Add("Local Application Data:  " + _

Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData))
        lbx.Items.Add("Cookies:                 " + _
            Environment.GetFolderPath(Environment.SpecialFolder.Cookies))
    End Sub
```

When I executed the program with the above changes on my system, I got the following result:



Download the sourcecode for this application.

Manipulating Files using System.IO.File and System.IO.FileInfo classes

We can manipulate files and perform different operations on them using the System.IO.File and System.IO.FileInfo classes. The System.IO.File class exposes static methods to perform various operations on files. On the other hand, the object of type System.IO.FileInfo class represents a single file through which we can get/set different properties of a file. Let us practice them one by one:

System.IO.File class

A review of shared methods of the File class is presented in the following table:

Member	Description
Copy()	Copies the specified file to the specified target path
Create()	Creates the specified file
Delete()	Deletes the specified file
Exists()	Returns Boolean value indicating whether the specified file exists
GetAttributes()	Returns an object of type System.IO.FileAttributes which contain different information regarding file like whether its is hidden or not
GetCreationTime()	Returns an object of type DateTime that represents the date time of the creation of this file
GetLastAccessTime()	Returns an object of type DateTime that represents the date time of the last access to this file
GetLastWriteTime()	Returns an object of type DateTime that represents the date time of the last write action to this file
Move()	Moves the specified file to the specified path.
Open()	Opens the specified file and returns the System.IO.FileStream object for this file
OpenRead()	Opens the specified file for reading purpose and returns a read only System.IO.FileStream object for this file
OpenWrite()	Opens the specified file for reading purpose and returns a read/write System.IO.FileStream object for this file
SetAttributes()	Accepts an object of type System.IO.FileAttributes which contain different information regarding file and set these attributes to the file.

Most of the above methods are very straight forward and it is difficult to show them working in a sample application and its output. So we will consider some of them individually to demonstrate how we can use them in our applications.

Creating a file using the Create() method

Suppose we wish to create a file named "VBDotNET.txt" on the root folder of C drive. We can write the following statement to do this:

```
File.Create("C:\VBDotNET.txt")
```

Author's Note: To compile the program containing the above and following statements in this section, you need to add the System.IO namespace in the source file of your program like

```
Imports System.IO
```

Copying and Moving a file using Copy() and Move() methods

If we want to copy a file to C:\my programs folder, we can use the following statement:

```
File.Copy("C:\VBDotNET.txt", "c:\my programs\VBDotNET.txt")
```

Similarly you can use the Move() method to move a file. Also you can use the overloaded form of the Copy() and Create() methods that take a Boolean value to indicate whether you wish to overwrite this file if the file with the same name exists in the target path. E.g.,

```
File.Copy("C:\VBDotNET.txt", "c:\my programs\VBDotNET.txt", True)
```

Checking the existence of the file using Exists() method

This method can be used to check the existence of the file

```
If Not File.Exists("C:\VBDotNET.txt") Then
  File.Create("C:\VBDotNET.txt")
End If
```

Getting Attributes of a file using GetAttributes() method

We can check the attributes of a file using the GetAttributes() method

```
Dim attrs As FileAttributes = File.GetAttributes("c:\VBDotNET.txt")
lbx.Items.Add("File 'c:\ VBDotNET.txt'")
lbx.Items.Add(attrs.ToString())
```

When I executed the program, I got informed that this is an archive file. Similarly you can set the attributes of the file by using the FileAttributes enumeration

System.IO.FileInfo class

The System.IO.FileInfo class is also used to perform different operations on files. Unlike the File class, we need to create an object of the FileInfo class to use its services. A review of some more important methods and properties of the FileInfo class is presented in the following table:

Member	Description
CreationTime	Gets or sets the time of creation for this file
Directory	Returns a DirectoryInfo object that represents the parent directory (folder) of this file
DirectoryName	Returns the name of the parent directory (in string) of this file
Exists	Returns Boolean value indicating whether the specified file exists
Extention	Returns the extention (type) of this file (e.g., .exe, .vb, .aspx)
FullName	Returns the full path and name of the file (e.g., C:\VBDotNET.txt)
LastAccessTime	Returns an object of type DateTime that represents the date time of the last access to this file
LastWriteTime	Returns an object of type DateTime that represents the date time of the last write action to this file
Length	Returns the size (number of bytes) in a file.
Name	Returns the name of the file (e.g., VBDotNET.txt)
CopyTo()	Copies this file to the specified target path
Create()	Creates this file
Delete()	Deletes this file
MoveTo()	Moves this file
Open()	Opens this file with various read/write and sharing privileges
OpenRead	Opens this file for reading purpose and returns a read only System.IO.FileStream object for this file
OpenWrite()	Opens this file for reading purpose and returns a read/write System.IO.FileStream object for this file

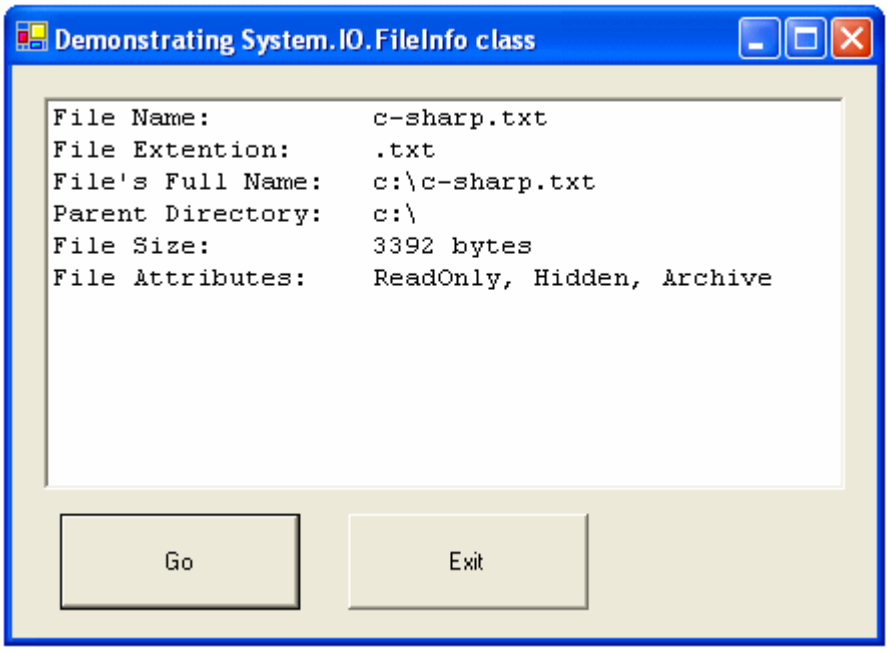
OpenText()	Opens this file and returns a System.IO.StreamReader object with UTF8 encoding that reads from an existing text file.

A quick and simple example
Although almost all the above properties and methods are understandable just by reading their name; we still need to create a simple example to demonstrate the functionality of the FileInfo class. In the following example, we will simply perform different operations on a file and display the result in a list box named 'lbx' **Author's Friendly Note:** I think I have made the worst use of my time in learning programming languages when I read something. I thought that "It is so easy, I have understood it to 100% and I don't need to implement a program for this". Remember most humans just can't learn even Console.WriteLine() without actually writing it in the IDE, compiling and executing the program.

We have written the following code on the 'Go' button's event handler:

```
    Private Sub btnGo_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles btnGo.Click
        Dim aFile As New FileInfo("c:\c-sharp.txt")
        lbx.Items.Add("File Name:          " + aFile.Name)
        lbx.Items.Add("File Extention:     " + aFile.Extension)
        lbx.Items.Add("File's Full Name:   " + aFile.FullName)
        lbx.Items.Add("Parent Directory:   " + aFile.DirectoryName)
        lbx.Items.Add("File Size:          " + aFile.Length.ToString() + " bytes")
        lbx.Items.Add("File Attributes:    " + aFile.Attributes.ToString())
    End Sub
```

We have simply used the properties of the FileInfo class to retrieve and print some information about a file. When I executed this program on my system, I got the following output:



Note: Before executing this program, I changed the attributes of file 'C:\C-Sharp.txt' to Readonly, Hidden and Archive using Windows Explorer.

Manipulating Directories (folders) using System.IO.Directory and System.IO.DirectoryInfo classes

Similar to the File and FileInfo classes we can perform several operations on directories using the Directory and DirectoryInfo classes. Again it is worth-noting that the System.IO.Directory class contains static methods while the System.IO.DirectoryInfo class contains instance members to perform various tasks on directories.

System.IO.Directory class

A review of shared methods of the Directory class is presented in the following table:

Member	Description
CreateDirectory()	Creates the specified directory
Delete()	Deletes the specified directory
Exists()	Returns a Boolean value indicating whether the specified directory exists
GetCreationTime()	Returns an object of type DateTime that represents the date time of the creation of the specified directory
GetDirectories()	Returns an array of strings containing the names of all the sub-directories of the specified directory.
GetFiles()	Returns an array of strings containing the names of all the files contained in the specified directory.
GetFileSystemEntries()	Returns an array of strings containing the names of all the files and directories contained in the specified directory.
GetParent()	Returns an object of type DirectoryInfo representing the parent directory of the specified directory
Move()	Moves the specified directory and all its contents (files and directories) to the specified path.

Creating, deleting and checking for the existence of directories

Some code that demonstrates how to perform the above operations is shown below.

```
    Private Sub btnGo_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles btnGo.Click
        lbx.Items.Add("Directory 'C:\Faraz' exists:       " + _
            Directory.Exists("C:\Faraz").ToString())
        lbx.Items.Add("Creating Directory 'C:\Faraz':     " + _
            Directory.CreateDirectory("C:\Faraz").ToString())
        lbx.Items.Add("Directory 'C:\Faraz' exists:       " + _
            Directory.Exists("C:\Faraz").ToString())
        lbx.Items.Add("Parent Directory of 'Faraz' is:    " + _
            Directory.GetParent("C:\Faraz").ToString())
        lbx.Items.Add("Deleting Directory 'C:\Faraz'...   ")
        Directory.Delete("C:\Faraz", True)
        lbx.Items.Add("Directory 'C:\Faraz' exists:       " + _
            Directory.Exists("C:\Faraz").ToString())
    End Sub
```

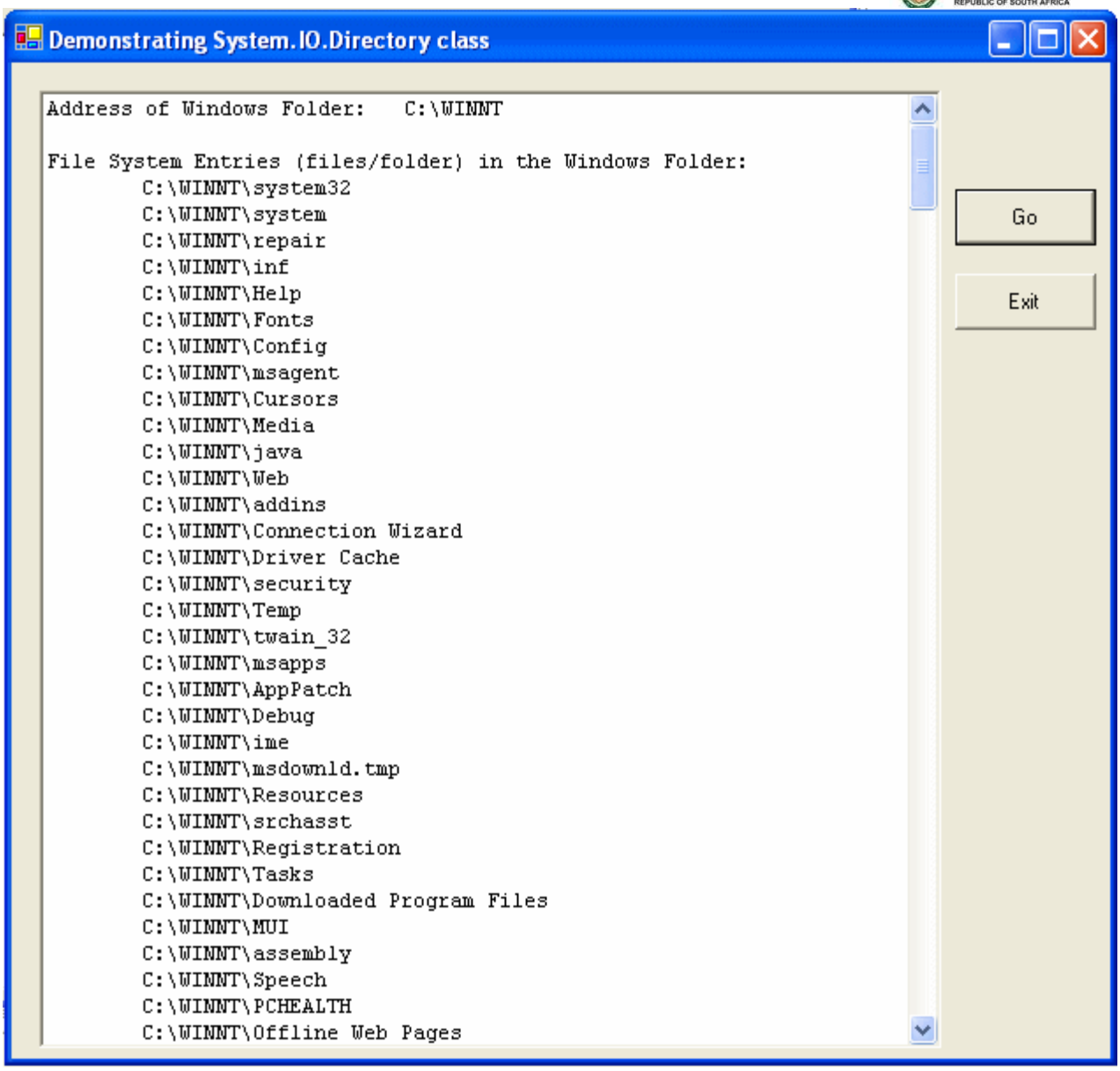
Again, this code simply calls the various shared methods of the Directory class to perform these operations. One thing to note here is that we have passed the path-string and a true value to the Directory.Delete() method. Passing the true value as the second parameter tells the runtime environment (CLR) to remove the directory recursively i.e. not only deletes the files in this directory but also delete the files and directories contained in its sub-directories and so on.

Getting the contents (files and sub-directories) of a directory

The Directory class exposes three methods to retrieve the contents of a directory. Directory.GetDirectories() returns a list of all the sub-directories of the specified directory, Directory.GetFiles() returns a list of all the files in the specified directory and Directory.GetFileSystemEntries() returns a list of all the files and sub-directories contained in the specified directory. Let's get a list of the contents of the Windows folder of your system.

```
    Private Sub btnGo_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles btnGo.Click
        ' get the path of Windows Folder's System Folder
        Dim winFolder As String =
Environment.GetFolderPath(Environment.SpecialFolder.System)
        ' Separate the Windows Folder
        winFolder = winFolder.Substring(0, winFolder.LastIndexOf("\"))
        Dim fileSystemEntries() As String = Directory.GetFileSystemEntries(winFolder)
        Dim files() As String = Directory.GetFiles(winFolder)
        Dim directories() As String = Directory.GetDirectories(winFolder)
        ' show windows folder path        lbx.Items.Add("Address of Windows Folder:
" + winFolder)
        ' show files/folder in windows folder        lbx.Items.Add("")
        lbx.Items.Add("File System Entries (files/folder) in the Windows Folder: ")
        Dim fileSystemEntry As String
        For Each fileSystemEntry In fileSystemEntries
            lbx.Items.Add("    " + fileSystemEntry)
        Next
        ' show files in windows folder
        lbx.Items.Add("")
        lbx.Items.Add("Files in the Windows Folder: ")
        Dim aFile As String
        For Each aFile In files
            lbx.Items.Add("    " + aFile)
        Next
            ' show folder in windows folder
        lbx.Items.Add("")
        lbx.Items.Add("Directories in the Windows Folder: ")
        Dim aDirectory As String
        For Each aDirectory In directories
            lbx.Items.Add("    " + aDirectory)
        Next
    End Sub
```

And when I executed the above program on my system, I got the following result:



System.IO.DirectoryInfo class

The System.IO.DirectoryInfo class is also used to perform different operations on directories. Unlike the Directory class, we need to create an object of the DirectoryInfo class to use its services. A review of some of the important methods and properties of the DirectoryInfo class is presented in the following table:

Member	Description
Exists	Returns a Boolean value indicating whether the specified directory exists.
Extention	Returns the extention (type) of this directory
FullName	Returns the full path and name of the directory (e.g., C:\Faraz)

Name	Returns the name of the directory (e.g., Faraz)
Parent	Returns the full path and name of the parent directory of this directory.
Create()	Creates a directory with the specified name
Delete()	Deletes the directory with the specified name
GetDirectories()	Returns an array of type DirectoryInfo that represents all the sub-directories of this directory.
GetFiles()	Returns an array of type FileInfo that represents all the files contained in this directory.
GetFileSystemInfos()	Returns an array of type FileSystemInfo that represents all the files and folders contained in this directory.
MoveTo()	Moves this directory and all its contents (files and directories) to the specified path.
Refresh()	Refreshes this instance of DirectoryInfo.

Demonstration application for the DirectoryInfo class

Here we have changed the previous example so that it now uses the DirectoryInfo class instead of the Directory class. The output of this program will remain the same as that of the previous one. The modified code is:

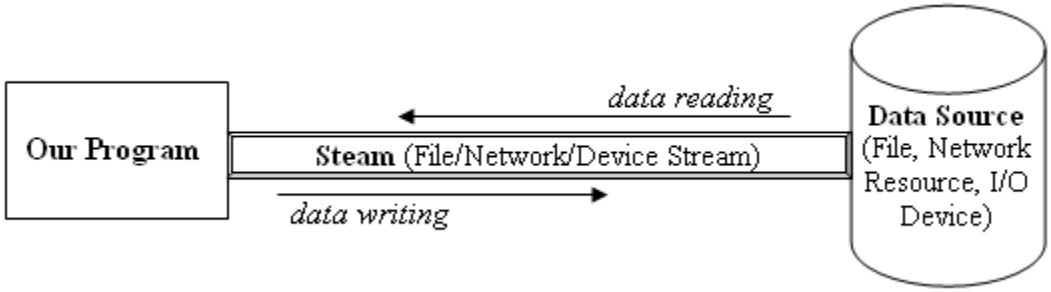
```
    Private Sub btnGo_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles btnGo.Click
        ' get the path of Windows Folder's System Folder
        Dim winFolder As String =
Environment.GetFolderPath(Environment.SpecialFolder.System)
        ' Separate the Windows Folder
        winFolder = winFolder.Substring(0, winFolder.LastIndexOf("\"))
        Dim winFolderObj As New DirectoryInfo(winFolder)
        Dim fileSystemInfos() As FileSystemInfo = winFolderObj.GetFileSystemInfos()
        Dim fileInfos() As FileInfo = winFolderObj.GetFiles()
        Dim directoryInfos() As DirectoryInfo = winFolderObj.GetDirectories()
        ' show windows folder path
        lbx.Items.Add("Address of Windows Folder:   " + winFolderObj.FullName)
        ' show files/folder in windows folder
        lbx.Items.Add("")
        lbx.Items.Add("File System Entries (files/folder) in the Windows Folder: ")
        Dim fileSystemInfoObj As FileSystemInfo
        For Each fileSystemInfoObj In fileSystemInfos
            lbx.Items.Add("    " + fileSystemInfoObj.FullName)
        Next
        ' show files in windows folder
        lbx.Items.Add("")
        lbx.Items.Add("Files in the Windows Folder: ")
        Dim fileInfoObj As FileInfo
        For Each fileInfoObj In fileInfos
            lbx.Items.Add("    " + fileInfoObj.FullName)
        Next
        ' show folder in windows folder
        lbx.Items.Add("")
        lbx.Items.Add("Directories in the Windows Folder: ")
        Dim directoryInfoObj As DirectoryInfo
        For Each directoryInfoObj In directoryInfos
            lbx.Items.Add("    " + directoryInfoObj.FullName)
        Next
    End Sub
```

As you might have noticed, the only difference between this and the previous code is that here we are using the DirectoryInfo object's methods and properties instead of the Directory

class' static methods. Again the result of the program will be same as that of the previous one, printing the names of files/folders in the Windows folder. Hence you have seen how easy and straight forward it is to perform different operations on files and folders and to get information about file system and the an application's environment in .NET. One last word of caution is that you must be very careful when manipulating files and directories in your application. In the examples in this lesson, we haven't attempted to catch any exceptions that might be thrown because of the absence of specified files and the privileges on different files. It is always a better practice to write the file access code in Try...Catch blocks.

Streams

With in context with regards programming, various definitions of stream exist. Some say 'A stream is an object used to read and write data to and from a data source (e.g., file memory, network, device, etc)'. Some say 'A stream is an abstraction of a sequence of bytes, like a file'. I perceive a stream as a data channel having two ends; one is attached to a data source while the other is attached to a reader or writer of data. The data source can be a file, data in memory or data on another PC. We use File Streams, Memory Streams and Network Streams to read/write data to and from these data sources. Hence, the basic functionality of any stream is to allow data to be read from and written to a data source.



An overview of the different types of streams

There are many different data sources. Data can be read from files stored on a disk, on a remote PC, in memory or from some other I/O devices. As developers, we need a simple clean interface to access data from these many data sources using simple methods like Read() and Write(). We don't want to go into the lower level details, such as how a data source is accessed and how data is retrieved and saved to the data source, and in which format.

Streams provide exactly these features. .NET provides different classes to serve as different types of stream. The base class of all the streams in the .NET framework is System.IO.Stream. If you want to access data in a file you may use System.IO.FileStream. If you want to access data in memory, you may use System.IO.MemoryStream. Also if you want to connect to a remote PC, you may use System.Net.Sockets.NetworkStream. The best thing about the Stream architecture in .NET is that you don't need to worry about how data is actually read from a local file system, network socket or memory; all you need to do is to instantiate the stream object by defining the data source to connect to and then call the simple Read() and Write() methods. We have been using Console.WriteLine() and Console.ReadLine() methods right from the start. In fact, the System.Console class represents the input, output and error stream to the console window. By calling the Write() method on the Console stream, we can send the data (bytes) to the console window.

The System.Stream class - the base of all streams in the .Net framework

The System.Stream is an abstract class which all other streams in the .NET framework

inherit. It exposes some properties and methods overridden by the concrete stream classes (like FileStream, NetworkStream, etc). A review of some of its more interesting properties and methods is provided in the following table:

Member	Description
CanRead	Returns a Boolean value indicating whether the stream can read from the data source.
CanWrite	Returns a Boolean value indicating whether the stream can write to the data source.
Length	Returns the length or number of bytes in the current stream.
Position	Gets/Sets the current position of the stream. Any read/write operation on the stream is carried out at the current position.
Close()	Closes the stream.
Flush()	Writes all the data stored in the stream buffer to the data source.
Seek()	Sets the current position of the stream.
Read(buffer as byte(), offset as Integer, count as Integer) As Integer	Reads the specified number of bytes from the current position of the stream into the supplied array of bytes and returns the number of bytes actually read from the stream
ReadByte()	Reads a single byte from the current position of the stream and returns the byte casted into an int. The '-1' return value indicates the end of the stream of data.
Write(buffer as byte(), offset as Integer, count as Integer)	Writes the specified number of bytes at the current position of the stream from the supplied array of bytes.
WriteByte()	Writes a single byte at the current position of the stream.

Different types of file streams - Reading and Writing to files

The major topic of this section is about file streams, which are used to read from and write to files. There are various classes in the .NET framework that can be used to read and write files. You can simply use System.IO.FileStream to read/write bytes to the file. Alternatively you may use the System.IO.BinaryReader and System.IO.BinaryWriter classes to read binary data as primitive data types. And you can also use the System.IO.StreamReader and System.IO.StreamWriter classes to read/write text files. We will demonstrate each of these one by one.

Using System.IO.FileStream to read and write data to files

The System.IO.FileStream class inherits the System.IO.Stream class to provide core stream functionality to read and write data to files. It implements all of the abstract members of the Stream class to work with files. Before using any of the stream operations, we need to import the System.IO namespace in our project

```
Import System
Import System.IO
```

We can instantiate the FileStream class in many different ways. We may use any of the File.Open(), File.OpenRead() and File.OpenWrite() methods of the System.IO.File class.

```
Dim fs As FileStream = File.Open("C:\VBDotNET.txt", FileMode.Open)
```

You may also use the FileInfo class' Open(), OpenRead() and OpenWrite() methods.

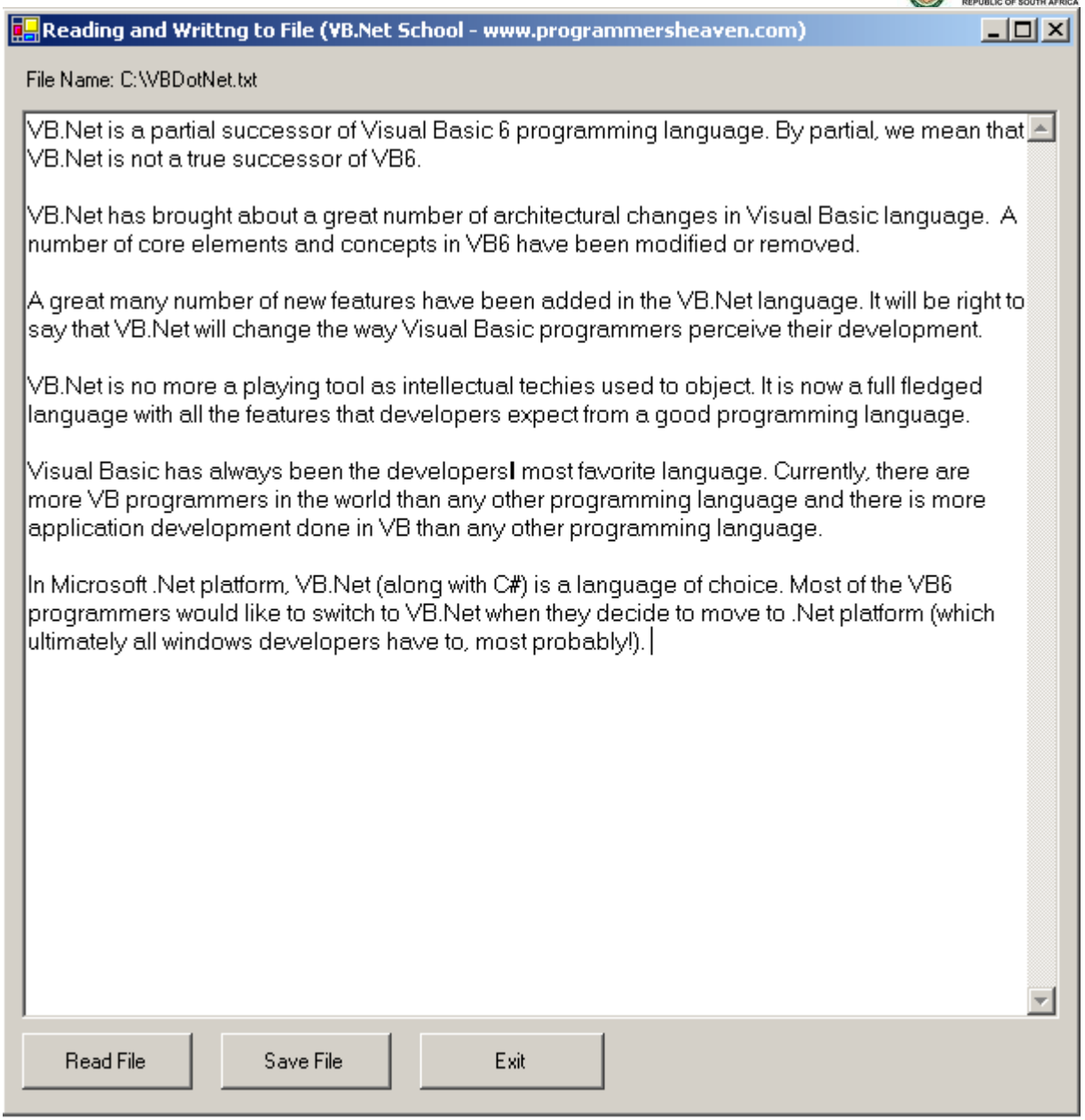
```
Dim objFileInfo As new FileInfo("C:\VBDotNet.txt ")
Dim fs As FileStream = objFileInfo.Open(FileMode.Open)
```

Finally you may use any of the number of overloaded constructors of the FileStream class. When creating any file stream, we may define four parameters.

- A string representing the path and name of the file
- A FileMode enumeration instance that defines how the operating system should open the file. The possible values include Open, OpenOrCreate, Append, Create and others. If FileMode is Open, it will attempt to open the existing file. If FileMode is OpenCreate it will attempt to open the existing file; if no file exists, it will create the new one. In Append file mode, the new data will be written at the end of the existing file.
- A FileAccess enumeration instance defines which operations are allowed during the file access. The possible values include Read, Write and ReadWrite. If FileAccess is Read, you can only read from the file stream.
- A FileShare enumeration instance that defines the sharing options for the file. The possible values include None, Read, ReadWrite, Write. If FileShare is None, no other stream can open this file until you have got this stream open. If FileShare is Read, other streams can open and only read from this file.

Opening and reading from a file

Now we've got a good grasp of the theory of file streams, let's do something practical. In practice, handling a file stream is as easy as anything else in VB.NET. Let's create a windows application that reads a file into a text box, allows the user to change the text and saves it back to the file. The application will finally look like:



The code behind it is very simple. The event handler for the Read File button (btnLoadFile) is:

```
    Private Sub btnLoadFile_Click(ByVal sender As System.Object,
        ByVal e As System.EventArgs) Handles btnLoadFile.Click
        Dim fileName As String = "C:\VBDotNet.txt"
        lblFileName.Text = "File Name: " + fileName
        ' Open existing file in Read only mode without allowing any sharing
        Dim fs As New FileStream(fileName, FileMode.Open, FileAccess.Read,
FileShare.None)
        ' create an array of bytes of the size of file
        ' and read file contents to it.
        Dim byteText(fs.Length) As Byte
        fs.Read(byteText, 0, byteText.Length)
        ' convert bytes array to string and display in the text box
```

```
        txtFileText.Text =
System.Text.Encoding.ASCII.GetString(byteText)
        ' close the file stream so that other streams may use it
        fs.Close()
    End Sub
```

We have used the file stream to open the existing file (FileMode.Open) in read only mode (FileAccess.Read) and without allowing any other stream to use it while our application is using this file (FileShare.None). We then created an array of bytes with length equal to the length of the file and read the file into it.

```
fs.Read(byteText, 0, byteText.Length)
```

Here we have specified that we want to read the file contents into the byteText array, that the file stream should start filling from the '0' index and that it should read byteText.Length (size of file) bytes from the file.

After reading the contents, we need to convert the byte array to a string so that it can be displayed in the text box. Finally, we have closed the stream (and thus the file) so that other streams may access it.

Similarly, the event handler for Save File button (btnSaveFile) is:

```
    Private Sub btnSaveFile_Click(ByVal sender As System.Object,
        ByVal e As System.EventArgs) Handles btnSaveFile.Click
        Dim fileName As String = "C:\VBDotNet.txt"
        ' Open existing file in Read only mode without allowing any sharing
        Dim fs As New FileStream(fileName, FileMode.Open, FileAccess.Write,
FileShare.None)
        ' covert the text (string) in the text box to bytes array
         ' and make the byteText reference to point to that byte array
        Dim byteText() As Byte = System.Text.Encoding.ASCII.GetBytes(txtFileText.Text)
        ' write the byte array to file from the start to end
        fs.Write(byteText, 0, byteText.Length)
        ' close the file stream so that other streams may use it
        fs.Close()
    End Sub
```

Here we have first converted the text in the text box to a byte array and then written it to the file using FileStream class' Write() method:

```
fs.Write(byteText, 0, byteText.Length)
```

In the above line we have told the FileStream class to write the bytes in the byteText array to the associated file. We have asked it to start writing from the first byte of the array and write the complete array to the file.

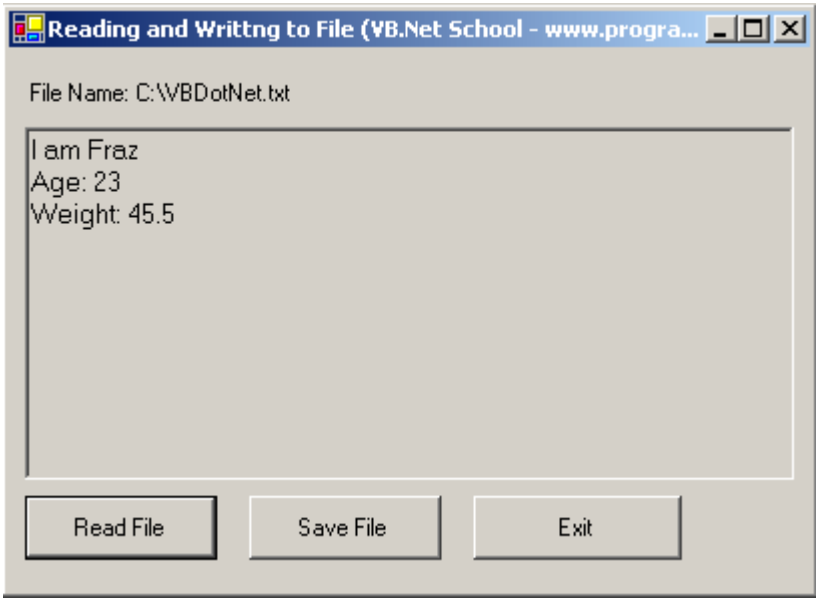
Download the sourcecode for this application.

Using BinaryReader and BinaryWriter to read and write primitives to files

The problem with using the FileStream class is that we can only read and write bytes to a file. We have to explicitly convert other types of data (int, double, bool, string) to bytes before writing to the file (and vice versa for reading). The .NET framework class library provides two classes that allow us to read and write primitive data types to a file. We can

use System.IO.BinaryReader to read primitive data types from a file and System.IO.BinaryWriter to write primitives to the file. An important point about the BinaryReader and BinaryWriter classes is that they need a stream to be passed in their constructor. These classes use the stream to read and write primitives.

Let's create an application that writes different primitives to a file and then read them back. The application will finally look like this:



The text box in the application is read only. At first, the Read File button is also disabled and the user needs to select the Save File button to save the file first, and then read the file back to the text box. The contents written to the file are hard coded in the Save File button's click event handler which is:

```
    Private Sub btnSaveFile_Click(ByVal sender As System.Object,
         ByVal e As System.EventArgs) Handles btnSaveFile.Click
        Dim fileName As String = "C:\VBDotNet.txt"
        ' Open existing file in Read only mode without allowing any sharing
        Dim fs As New FileStream(fileName, FileMode.Open, FileAccess.Write,
FileShare.None)
        ' Open the Writer over this file stream
        Dim writer As New BinaryWriter(fs)
        ' write different types of primitives to the file
        writer.Write("I am Fraz" + vbCrLf)
        writer.Write("Age: ")
        writer.Write(23)
        writer.Write(vbCrLf + "Weight: ")
        writer.Write(45.5)
        ' close the file stream so that other streams may use it
        writer.Close()
        fs.Close()
        btnLoadFile.Enabled = True
    End Sub
```

Here we have first created the file stream and used it to instantiate the BinaryWriter class. We have then written different primitives to the stream using BinaryWriter's Write()

method, which has many overloaded versions to write different types of primitives. Finally we have closed the two streams and enabled the Load File button.

The event for the Load File button is:

```
    Private Sub btnLoadFile_Click(ByVal sender As System.Object,
         ByVal e As System.EventArgs) Handles btnLoadFile.Click
        Dim fileName As String = "C:\VBDotNet.txt"
        lblFileName.Text = "File Name: " + fileName
        ' Open existing file in Read only mode without allowing any sharing
        Dim fs As New FileStream(fileName, FileMode.Open, FileAccess.Read,
FileShare.None)
        ' Open the Reader over this file stream
        Dim reader As New BinaryReader(fs)
        ' read different types of primitives to the file
        Dim name As String = reader.ReadString()
        Dim ageString As String = reader.ReadString()
        Dim age As Integer = reader.ReadInt32()
        Dim wtString As String = reader.ReadString()
        Dim weight As Double = reader.ReadDouble()
        ' concatenate primitives into single string and display in the text box
        txtFileText.Text = name + ageString + age.ToString() + wtString +
weight.ToString()
        ' close the file stream so that other streams may use it
        reader.Close()
        fs.Close()
    End Sub
```

Here we create the BinaryReader class' object using the FileStream object. We then read the primitives previously written to the file. After reading all the data, we concatenate the primitives to a single string and display it in the text box. Finally we close the two streams. The important point to note here is that the primitives are read in the same order they were written.

Download the sourcecode for this application.

Using StreamReader and StreamWriter to read and write text files

The classes StreamReader and StreamWriter are used to read and write text files. They have got useful methods like ReadLine() and ReadToEnd() to facilitate the reading and writing of text files. More than that, these streams can use different text encodings (like ASCII and Unicode) for reading and writing the files.

Download the sourcecode for this application.

Author's Note: You might have noticed that we haven't gone into the details of specific classes in this section. The reason is that these classes are very similar to each other. They all provide the same functionality and that is to read/write data from/to files. They have a number of common and similar methods and some of them do not even serve any purpose. What you need to learn is which class should be used in which scenario. You can always see the description of individual methods of these classes in MSDN. Once again, I will suggest not to leave the topic without practice just because it looks simple and easy. You should spend some hours playing with various streams for better understanding.

Serialization and De-serialization

Serialization is the process of writing objects to the stream while De-serialization is the process of reading objects from the stream. Up until now, we have seen how to read/write primitive types to the streams but we haven't read/written any explicit (user defined) type to the stream. There are certain points that must be clear before actually implementing the serialization.

- The purpose of serializing or writing an object to a stream is to save its state. The state of an object is determined by its instance variables. Hence serializing an object means writing all of its member (or instance) variables (also called an object graph) to the stream. Methods or static members are not serialized or written to the stream.

- You can serialize an object yourself by simply writing all of its member variables to the stream. When de-serializing, you would have to read all the member variables in the same sequence in which they were written. However this process of serializing and de-serializing has two major disadvantages:

1. It is a tedious job to write all the member variables yourself and it might become hectic if your class contains a lot of variables and if your class contains other user defined objects.
2. It is not the standard procedure. The person who is willing to de-serialize the object you serialized previously, would have to be aware of the sequence in which you wrote the member variables and must follow that sequence.

- The .NET framework takes care of these issues and provides binary and SOAP formatters using which you can serialize your object just by calling their Serialize() and Deserialize() methods.

- There is a serious security issue connected with serialization. You might not want certain classes to be serialized to the stream or you might not want to serialize all of your member variables to the stream. For example, a web-application may not allow the UserInfo object to be serialized or its Password field to be serialized.

- All the classes in .NET are un-serializable by default, that is, they can not be written to a stream. You have to explicitly mark your class to be serializable using the <Serializable()> attribute.
- You can optionally mark a particular member variable (or field) as Non-Serialized by using the <NonSerialized> attribute to prevent the CLR from writing that field when serializing the object.

Author's Note: We haven't covered attributes in our VB.NET School up till this issue. Attributes are a fantastic feature of VB.NET that allows you to provide extra information about certain entities (like assemblies, classes, methods, properties and fields). The beauty of attributes lies in their power and equal amount of simplicity. If you are interested to learn about attributes, you will find MSDN very helpful.

Implementing Serialization and Deserialization - A simple example

Let's create a console application that contains a serializable class. The application will serialize its object to a file and then deserialize it again from the file to another object. We will use a simple class that calculates the sum of two integer variables. The complete source code of the program is:

```
Imports System
Imports System.IO     ' for FileStream
Imports System.Runtime.Serialization.Formatters.Binary  ' for BinaryFormatter
Module Test
    Public Sub Main()
        Dim addition As New Addition(3, 4)
        Dim fs As New FileStream("C:\VBDotNET.txt", FileMode.Create)
        Dim binForm As New BinaryFormatter()
        Console.WriteLine("Serializing the object....")
        binForm.Serialize(fs, addition)
        fs.Position = 0 ' move to the start of file
        Console.WriteLine("DeSerializing the object....")
        Dim sum As Addition = CType(binForm.Deserialize(fs), Addition)
        Dim res As Integer = sum.Add()
        Console.WriteLine("The sum of 3 and 4 is: {0}", res)
    End Sub
End Module
<Serializable()> _
Class Addition
    Private num1 As Integer
    Private num2 As Integer
    Private mResult As Integer

    Public Sub New()
    End Sub
    Public Sub New(ByVal num1 As Integer, ByVal num2 As Integer)
        Me.num1 = num1
        Me.num2 = num2
    End Sub
    Public Function Add() As Integer
        mResult = num1 + num2
        Return mResult
    End Function
    Public ReadOnly Property Result() As Integer
        Get
            Return mResult
        End Get
    End Property
End Class
```

The Addition class is very simple and has three private members. Note that we have marked the class with the <Serializable()> attribute. Also note that we have included the appropriate namespaces.

```
Imports System
Imports System.IO     ' for FileStream
Imports System.Runtime.Serialization.Formatters.Binary  ' for BinaryFormatter
```

In the Main() method we have created an instance of the Addition class. We have then created a file stream and serialized the object to this file using the BinaryFormatter class (We will come to the BinaryFomatter later in the lesson).

```
Dim fs As New FileStream("C:\VBDotNET.txt", FileMode.Create)
Dim binForm As New BinaryFormatter()
Console.WriteLine("Serializing the object....")
binForm.Serialize(fs, addition)
```

This is all we need to do on our part when serializing an object. Deserializing is again similar but before deseializing we need to set the file pointer position to the start of the file

```
fs.Position = 0 ' move to the start of file
```

Now we can deserialize the object from the stream using the same BinaryFormatter instance:

```
Console.WriteLine("DeSerializing the object....")
Dim sum As Addition = CType(binForm.Deserialize(fs), Addition)
```

The BinaryFormatter class' Deserialize() method takes the file stream as its parameter. Then reads the object graph from it and returns an object of type System.Object. We have to explicitly cast it to the desired class. Once we have got the object, we call the Add() method of the class which uses the private members of the class to compute the sum and print the result on the console

```
Dim res As Integer = sum.Add()
Console.WriteLine("The sum of 3 and 4 is: {0}", res)
```

When you compile and execute the code, you will see the following output:

```
Serializing the object....
DeSerializing the object....
The sum of 3 and 4 is: 7
Press any key to continue
```

If you comment just the <Serializable()> attribute over the Addition class definition, you will get an exception when the Serialize() method of the BinaryFormatter is called. Try it!

```
<Serializable()> _
Class Addition
                ...
```

Formatters in Serialization

A formatter describes the format in which an object is serialized. The formatter should be standard and both the serializing and deserializing parties must use the same or a compatible formatter. In .NET, a formatter needs to implement the System.Runtime.Serialization.IFormatter interface. The two common formatters are the Binary Formatter (System.Runtime.Serialization.Formatters.Binary.BinaryFormatter) and the SOAP Formatter (System.Runtime.Serialization.Formatters.Soap.SoapFormatter). The SOAP (Simple Object Access Protocol) is a standard protocol over the internet and has got the support of Microsoft, IBM and other industry giants. The Binary Formatter is more optimized for a local system.

Preventing certain elements from Serializing - The <NonSerialized()> attribute

You can prevent the CLR from serializing certain fields when serializing an object. There may be different reasons for that. You might decide on it for security purposes or if you want to save disk space by not writing some long irrelevant fields. For this you simply need

to mark the field with the <NonSerialized()> attribute. For example, let us change the result field to <NonSerialized()>. The complete source code of the modified program is:

```
Imports System
Imports System.IO     ' for FileStream
Imports System.Runtime.Serialization.Formatters.Binary  ' for BinaryFormatter
Module Test
    Public Sub Main()
        Dim addition As New Addition(3, 4)
        addition.Add()
        Console.WriteLine("The value of result is: {0}", addition.Result)
        Dim fs As New FileStream("C:\VBDotNET.txt", FileMode.Create)
        Dim binForm As New BinaryFormatter()
        Console.WriteLine("Serializing the object....")
        binForm.Serialize(fs, addition)
        fs.Position = 0 ' move to the start of file
        Console.WriteLine("DeSerializing the object....")
        Dim sum As Addition = CType(binForm.Deserialize(fs), Addition)
        Console.WriteLine("The value of result is: {0}", sum.Result)
        Console.WriteLine("The sum of addition is: {0}", sum.Add())
    End Sub
End Module
<Serializable()> _
Class Addition
    Private num1 As Integer
    Private num2 As Integer

    <NonSerialized()> _
    Private mResult As Integer

    Public Sub New()
    End Sub
    Public Sub New(ByVal num1 As Integer, ByVal num2 As Integer)
        Me.num1 = num1
        Me.num2 = num2
    End Sub
    Public Function Add() As Integer
        mResult = num1 + num2
        Return mResult
    End Function
    Public ReadOnly Property Result() As Integer
        Get
            Return mResult
        End Get
    End Property
End Class
```

Note the <NonSerialized()> attribute over the mResult field. Now consider the Main() method of the program. We first create an instance of the Addition class with two numbers, then call its Add() method to compute the result and print it.

```
Dim addition As New Addition(3, 4)
addition.Add()
Console.WriteLine("The value of result is: {0}", addition.Result)
```

We then serialize the object and deserialize it back to another object, just like in the previous example:

```
Dim fs As New FileStream("C:\VBDotNET.txt", FileMode.Create)
Dim binForm As New BinaryFormatter()
Console.WriteLine("Serializing the object....")
```

```
binForm.Serialize(fs, addition)fs.Position = 0 ' move to the start
of file
Console.WriteLine("DeSerializing the object....")
Dim sum As Addition = CType(binForm.Deserialize(fs), Addition)
```

Now consider the next steps. First we print the value of the result variable using the Result property.

```
Console.WriteLine("The value of result is: {0}", sum.Result)
```

If the value of Result is not serialized, the above line should print zero. Finally we print the result of the addition

```
Console.WriteLine("The sum of addition is: {0}", sum.Add())
```

The above line should print the sum of 3 and 4 if the variables num1 and num2 were serialized.

When we compile and execute the program, we get the following result.

```
The value of result is: 7
Serializing the object....
DeSerializing the object....
The value of result is: 0
The sum of addition is: 7
Press any key to continue.
```

In the output, we can see that the value of the result field after deserializtiaon is zero, suggesting that the field result is not serialized with the object. The value of result (3+4=7) after calling the Add() method suggests that the fields num1 and num2 did get serialized with the object. Hence, we can prevent some of our fields from serializing with the object.

Getting notified when Deserializing - the IDeserializationCallBack interface

When we don't want some of our fields to serialize with the object, we may want to perform some work on the object when deserializing so that we may prepare non-serialized fields. For example, we may want to compute the result variable of the Addition class when deserializing the object. For this, we need to implement the IDeserializationCallBack interface. The interface only contains one method

```
Sub OnDeserialization(ByVal sender As Object)
```

This method is always called in the implementing class when the object is deserialized. Let's change the previous application so that the result variable retains its value even if it is not serialized. The modified source code is:

```
Imports SystemImports System.IO   ' for FileStream
Imports System.Runtime.Serialization  ' for IDeserializationCallBack
Imports System.Runtime.Serialization.Formatters.Binary     ' for BinaryFormatter
Module Test
    Public Sub Main()
        Dim addition As New Addition(3, 4)
        addition.Add()
        Console.WriteLine("The value of result is: {0}", addition.Result)
        Dim fs As New FileStream("C:\VBDotNET.txt", FileMode.Create)
        Dim binForm As New BinaryFormatter()
```

```
        Console.WriteLine("Serializing the object....")
        binForm.Serialize(fs, addition)
        fs.Position = 0 ' move to the start of file
        Console.WriteLine("DeSerializing the object....")
        Dim sum As Addition = CType(binForm.Deserialize(fs), Addition)
        Console.WriteLine("The value of result is: {0}", sum.Result)
    End Sub
End Module
<Serializable()> _
Class Addition
    Implements IDeserializationCallback
    Private num1 As Integer
    Private num2 As Integer
    <NonSerialized()> _
    Private mResult As Integer
    Public Sub New()
    End Sub
    Public Sub New(ByVal num1 As Integer, ByVal num2 As Integer)
        Me.num1 = num1
        Me.num2 = num2
    End Sub
    Public Function Add() As Integer
        mResult = num1 + num2
        Return mResult
    End Function
    Public ReadOnly Property Result() As Integer
        Get
            Return mResult
        End Get
    End Property
    Public Sub OnDeserialization(ByVal sender As Object) Implements
IDeserializationCallback.OnDeserialization
        mResult = num1 + num2
    End Sub
End Class
```

Note that this time the class Addition inherits the IDeserializationCallback interface and provides the definition of the OnDeserialization method in which it computes the sum of num1 and num2 and stores it in the result field:

```
Public Sub OnDeserialization(ByVal sender As Object) Implements
IDeserializationCallback.OnDeserialization
    mResult = num1 + num2
End Sub
```

The Main() method is very similar to the one in the previous program but this time we should not get a zero value in the result field after deserialization if the OnDeserialization method is called. When we compile and execute the above program, we see the following result:

```
The value of result is: 7
Serializing the object....
DeSerializing the object....
The value of result is: 7
Press any key to continue
```

The result shows that the OnDeserialization() method is actually called when deserialization is performed as we did not get the zero value of the result fields after deserialization.

Supplementary topic: Asynchronous Reading and Writing with Streams

Introduction

Up until now we have only used synchronous reading and writing to streams. Now we will see asynchronous reading and writing to streams. The first obvious question is what asynchronous and synchronous read/write means? Just consider our previous procedure of reading and writing to the stream. We used to call the Read() and Write() methods. For example, we call the Read()method by specifying the amount of data to be read to the supplied array

```
Dim byteText(fs.Length) As Byte
fs.Read(byteText, 0, byteText.Length)
SomeOtherMethod()
```

When we call the Read() method, our program (or the current thread) is blocked until the data has been read to the supplied array and SomeOtherMethod() is only called when the complete data has been read into the array. This is called a synchronous read, i.e. we are actually waiting till the data is read. The same thing happens with Write(), and this is called a synchronous write. In an asynchronous read and write we just issue the command to read or write through the System.IO.Stream class' BeginRead() and BeginWrite() methods. Once we call BeginRead() or BeginWrite(), two things start simultaneously:

- The current thread starts executing the statements following BeginRead() or BeginWrite() without waiting for the read or write to be completed.
- The Common Language Runtime (CLR) starts reading or writing the data and informs our program when it is complete.

So it looks nice. But how does the CLR inform our program that the read or write has been completed? Well asynchronous operations are always implemented in VB.NET using delegates, be it Events, Threads or Asynchronous I/O. So the BeginRead() and BeginWrite() methods take a delegate of type System.AsyncCallback. The delegate AsyncCallback is defined in the System namespace as:

```
Public Delegate Sub AsyncCallback(ByVal ar As IAsyncResult)
```

This means that the delegate AsyncCallback can reference any method that has a no return type and takes a parameter of type System.IAsyncResult. The type IAsyncResult can be used to supply information about the asynchronous operation. Most of the time, we don't bother about this object. A sample method that an AsyncCallback delegate can reference is:

```
Public Sub OnWriteCompletion(ByVal ar As IAsyncResult)
        Console.WriteLine("Write Operation Completed")
End Sub
```

A demonstration application

Let's now create a simple console application that demonstrates the use of an Asynchronous read/write to streams. The read/write operations in this application will be asynchronous. The source code of the program is:

```
Imports SystemImports System.IO
     ' for FileStreamImports System.ThreadingModule Test
```

```
    Public Sub Main()
        Dim fs As New FileStream("C:\VBDotNet.txt", FileMode.Open)
        Dim fileData(CInt(fs.Length)) As Byte
        Console.WriteLine("Reading file...")
        fs.Read(fileData, 0, fileData.Length)
        fs.Position = 0
        Dim callbackMethod As New AsyncCallback(AddressOf OnWriteCompletion)
        fs.BeginWrite(fileData, 0, fileData.Length, callbackMethod, Nothing)
        Console.WriteLine("Write command issued")
        Dim i As Integer
        For i = 1 To 10
            Console.WriteLine("Count Reaches: {0}", i)
            Thread.Sleep(10)
        Next
        fs.Close()
    End Sub
    Public Sub OnWriteCompletion(ByVal ar As IAsyncResult)
        Console.WriteLine("Write Operation Completed...")
    End Sub
End Module
```

In the above code block, we have defined a delegate instance 'callbackMethod' of type AsyncCallback in the Main() method and made it reference the OnWriteCompletion() method. We have created a file stream, which reads data to the array 'fileData' and moved the file pointer position to the start of the file. We have then started writing to the file using the BeginWrite() method, passing it the same byte array 'fileData' and the delegate to the callback method 'OnWriteCompletion()'.

```
Dim callbackMethod As New AsyncCallback(AddressOf OnWriteCompletion)
fs.BeginWrite(fileData, 0, fileData.Length, callbackMethod, Nothing)
```

We have then printed numbers in a loop introducing a delay of 10 milliseconds in each iteration. Finally we have closed the stream. Since we have called the BeginWrite() method to write the contents of the file, the main method thread should not block and wait until the writing is complete, it should continue and print the numbers in the loop. When the writing to the file is complete, the OnWriteCompletion() method should get called, printing the message on the console. For test purposes, I copied all the text in this lesson to the 'VBDotNET.txt' file and executed the program to get the following result:

```
Reading file...
Write command issued
Count Reaches: 0
Count Reaches: 1
Count Reaches: 2
Write Operation Completed...
Count Reaches: 3
Count Reaches: 4
Count Reaches: 5
Count Reaches: 6
Count Reaches: 7
Count Reaches: 8
Count Reaches: 9
Press any key to continue
```

Here you can see in the output that after issuing the write command, the main method

thread did not get blocked, but continued to print the numbers. Once the write operation was completed, it printed the message while the loop continued to iterate.

Issues Regarding Asynchronous Read/Write

When using asynchronous read/write, certain issues should be considered:

- An Asynchronous read/write is designed for I/O tasks that take a relatively longer time to complete. It is an overkill to use asynchronous operations for reading and writing small files. This is the reason we have used a relatively larger file in the above demonstration.
- In the background, BeginRead() and BeginWrite() both create a separate thread and delegate the reading/writing task to this new thread. Hence you can also implement your own BeginRead() and BeginWrite() methods, for example, for the StreamReader and StreamWriter classes that do not contain these methods.

Important points regarding the use of Streams

- The most important thing which must be kept in mind when reading streams is that it is extremely important to close the stream your program created. If you don't close the stream, the files opened by your program will not be accessible to any other process until your application is closed. These types of bugs are really hard to debug.
- It is always good practice to open a file or stream in a Try...Catch block as the file specified in the code may not be available during the execution of the code.
- It is always a good trick to close the streams in the finally block as it is always guaranteed that the code in finally block will be executed regardless of the presence of exceptions. This way you would be dead sure that the opened stream will be closed.

Food for thought: Exercise 15

1. We have seen in the lesson that the File and FileInfo classes are very similar. The same is the case with the Directory and DirectoryInfo classes. Why does the System.IO namespace contain two classes with such similar functionality? Why didn't they provide only one of these classes?
2. What is the difference between binary and text files?
3. What is the significance of the StreamReader and the StreamWriter classes when we have the BinaryReader and the BinaryWriter classes? We can always read the string using the BinaryReader class' ReadString() method, so why should one use the StreamReader class?
4. What is the concept of serialization and de-serialization?
5. Why is using the Asynchronous read and write always the best practice?

We have published the answers here.

Solution of Last Issue's Exercise (Exercise 14)

1. What is the difference between multitasking and multithreading?

Multitasking is the process of sharing system resources among different applications (processes), while multithreading is the process of sharing system resources among different threads (execution unit) within an application. In multitasking, different processes share the processor's time while in multithreading, different threads share the processor time allotted to their parents process.

2. What is the difference between a process and a thread?

A process is an independently executing application. A thread is an execution unit within a process. A process may have multiple threads. All the threads in a process work for that process. A thread does not have separate memory space and other resources but shares the memory space and resources allotted to the parent process. Inter-thread communication is relatively a lot easier and common than the Inter-process communication.

3. Why and when should one use multiple threads (multi-threading) instead of multitasking?

When you need to perform different tasks closely related to your application (that use objects and resources held by your process) and which are closely related to the core logic of your application; you should go for multithreading. On the contrary, when you need to perform multiple tasks independent of each other simultaneously, you should go for multitasking.

4. What is the difference between the Abort(), Suspend() and Sleep() methods of the Thread class?

The Abort() method permanently terminates the execution of the thread. The Suspend() method pauses the execution of the thread until the Resume() method is called. The Sleep() method pauses the running of the thread for a specified amount of time.

5. Why is the ThreadAbortException thrown in the thread method when the Abort() method is called for that thread? Is it an error to call the Abort() method?

No, not at all, aborting the thread is not an error. The Abort() method throws the ThreadAbortException just to inform the executing thread that it is being asked to terminate and it should start performing the clean up code so that it may be terminated safely. The Abort() method is extremely useful when you provide your user with the option to Cancel the current activity. For example, you are searching for some text in a file in a thread and the user presses the Cancel button to stop searching. In this case, the Abort() method of the thread is called which throws the ThreadAbortException in the search method, which closes the file connection and gets terminated.

6. What are the advantages and disadvantages of accessing shared objects with multiple threads?

Using shared objects allows your threads to communicate with each other. It saves having multiple copies of the same data to be used by different threads. On the downside, the use of shared objects gives rise to issues like Thread synchronization, locking, race conditions and deadlocks. But usually the situation does not get so bad as to create problems like deadlocks in our programs. Programmers usually know the flow of their program execution and safely avoid such problems when using multithreading with shared objects.

7. Architectural Issue: Those of you who have some experience in Java programming languages must have observed the difference between Java and VB.NET's approach of implementing threads. In Java, the user inherits the java.lang.Thread class and overrides its Run() method while in VB.NET, we create an object of the Thread class and pass to it a delegate (function pointer) to the entry point for the thread. Compare the two approaches.

Java implements the Abstract Factory Design Pattern to provide the thread functionality. The abstract class Thread contains an abstract method called run(). Anyone wishing to create a thread simply inherits from this class and overrides its run() method and later calls the start() method to start the thread.

On the contrary, .NET uses its delegates to implement the threads. The delegates are internally converted to function pointers. Since function pointers are generally more efficient than object creation, VB.NET threads are more efficient at least in theory. Also in VB.NET, the method to be executed by a thread can be the part of any class and thus can use the functionality and resources of existing classes and logic more easily.

While in Java, the method is contained in the class inherited from the Thread class. Many people inherit their working class from the Thread class and add the run() method inside it to use the resources and logic in the class. But this is a very limited solution as in Java you can only perform single-inheritance. Hence most of the time, we need to create a separate class for each thread which sometimes is a hectic job.

On the other side, the Java design is much more object oriented than the VB.NET one. In java, you implement multithreading using the regular Object Oriented concepts of classes, inheritance and polymorphism. In VB.NET you need to learn about and use an extra constructs (delegates) to implement multithreading.

THE END

The solution of Food for thought for lesson 15 is:

1. We have seen in the lesson that the File and FileInfo classes are very similar. The same is the case with the Directory and DirectoryInfo classes. Why does the System.IO namespace contains two classes with such similar functionality? Why did they not provide only one of these classes?

The File class is intended to be used when you want to perform just a single operation like creating or copying a file. The FileInfo class is intended to be used when you want to perform a series of operations with the file like creating the file, setting its attributes and copying it to another location. The same is true for Directory and DirectoryInfo classes.

2. What is the difference between binary and text files?

The difference between binary and text files is the way they write data to the files. In binary files, the binary representation of values is written to the file. For example, the integer '4929067' which takes 4 bytes in memory will also take 4 bytes in the file. In the case of a text file, each value is written as a series of characters (ASCII or Unicode). For example, the integer '4929067' will be written as text and will take 7 bytes in ASCII encoding and 14 (7 x 2) bytes in Unicode encoding. Binary files are more efficient for reading and writing of data for machines while text files are more human readable.

3. What is the significance of the StreamReader and the StreamWriter classes when we have the BinaryReader and the BinaryWriter classes? We can always read the string from BinaryReader class ReadString() method so why should one use the StreamReader class?

The StreamReader and StreamWriter classes are also useful if you want to read/write the data line by line using their ReadLine() and WriteLine() methods. The StreamReader also exposes a useful ReadToEnd() method. Also the StreamReader class is more useful when reading files with different types of encoding. Finally, to cut a long story short, the StreamReader and StreamWriter classes are designed to work with text files while BinaryReader and BinaryWriter classes are designed to work with binary files.

4. What is the concept of serialization and de-serialization?

The serialization is the process of writing objects to streams. De-serialization on the other hand, is the process of reading objects from the stream.

5. Why is using Asynchronous read and write is not always the best practice?

The asynchronous read and write is more suited when we have large amounts of data to read and write and we don't want our program to be "hung up" waiting for I/O to complete. The use of asynchronous I/O does not provide performance benefit when performing the I/O

of short time durations. It is an overkill to apply asynchronous I/O for small amount of data.

About the Author:

Faraz Rasheed is a student of BSCS in the department of Computer Science, University of Karachi, Pakistan. He is also part of Operation Badar - an IT educational movement in Pakistan. He is an international student member of ACM (Association for Computing Machinery) and INETA (International .Net Association). He has keen interest in Object Oriented Analysis and Design (OOAD) and development in programming languages like VB.NET, Java, VC++ and VB.Net. He can be accessed via farazrasheed@acm.org or frazrasheed@hotmail.com

About the Author:

Faraz Rasheed is the Software Engineer at yEvolve Pvt. Ltd (http://www.yevolve.com). He is also. He is also a part of Operation Badar - an IT educational movement in Pakistan and international student member of ACM (Association for Computing Machinery). He has a keen interest in Object Oriented Analysis and Design (OOAD) and development in programming languages like C#, Java, VC++ and VB.Net. He can be accessed via farazrasheed@acm.org or frazrasheed@hotmail.com